Measuring and Characterizing System Behavior Using Kernel-Level Event Logging

Karim Yaghmour and Michel R. Dagenais
Département de génie électrique et de génie informatique
Ecole Polytechnique de Montréal
C.P. 6079, Succ. Centre-Ville
Montréal, Québec, CANADA, H3C 3A7
karym@opersys.com, michel.dagenais@polymtl.ca

Abstract

Analyzing the dynamic behavior and performance of complex software systems is difficult. Currently available systems either analyze each process in isolation, only provide system level cumulative statistics, or provide a fixed and limited number of process group related statistics. The Linux Trace Toolkit (LTT) introduced here provides a novel, modular, and extensible way of recording and analyzing complete system behavior. Because all significant system events are recorded, it is possible to analyze any desired subset of the running processes, and for instance distinguish between the time spent waiting for some relevant event (data from disk or another process) versus time spent waiting for some unrelated process to use up its time slice.

Despite the extensive information gathered, experimental results show that the LTT time and memory overhead is minimal ($< 2.5\%$ when observing core kernel events). Moreover, due to the LTT and Linux kernel modularity and open source code availability, the system is easily extended both in terms of system events gathered, and of later post-processing and graphical presentation.

1 Introduction

System performance measurement, for understanding and optimization, may be performed at different levels according to varying needs. For example, system administrators need to understand what is using up all the resources and where the bottlenecks are; this may help determine which hardware upgrade would be most beneficial, or who is abusing the system. On the other hand, the application user or developer wants to understand where all the elapsed time is spent; this may include several related processes as well as system services such as network and file systems.

At the system level, easily available performance data usually includes the average load, number of interrupts, number of packets sent and received, and number of disk blocks read and written. At user level, it is possible to obtain per process and per process group the elapsed time, system and user CPU time, number of input/output operations, instruction counts and portion of CPU time spent in each function.

This information is sufficient to analyze the overall system performance or single CPU intensive processes. However, a different approach is required to analyze complex networked multi-process software systems, running on multi-processor systems, and taking into account the aggressive disk caching policy of modern operating systems. An excellent example of such a complex performance problem is the recent discussions on the Linux kernel mailing lists about the lower than expected performance on very high-end web servers. After tedious ad hoc manual code instrumentation and repeated experiments, they eventually focused on two perceived problems: coarse grain locking in the TCP/IP stack and the thundering herd problem (several daemons were awakened when new data arrived while a single daemon could use the data anyway). Such complex and specialized applications motivate the need for a comprehensive yet low-overhead, modular, and extensible event monitoring system for detailed performance analysis.

In section 2, existing profiling and measurement tools are reviewed and discussed. Section 3 details the architecture of the new Linux Trace Toolkit

(LTT) presented here. Section 4 presents the overhead caused by LTT. Section 5 presents cases where LTT has been used to analyze system behavior and compares LTT to existing analysis tools. Section 6 discusses possible future directions.

2 Related work

As discussed in the previous section, there are two broad categories of existing profiling tools. The first is aimed at the detailed analysis of individual applications and the second is aimed at an overview of the system's behavior.

In the first category, we find tools such as DCPI [9], Morph [11], Path Profiler [6], Quantify [1] and GProf [12], to name a few. DCPI provides a detailed analysis of different processes running on a system down to pipeline stalls. In order to provide its highly detailed data, DCPI uses a very high frequency interrupt. Similarly, Morph uses the clock interrupt to gather data in order to optimize applications off-line. Both systems fail to provide their user with information on the interactions of the different processes. Neither enable the user to understand the dynamics of the observed system. Path Profiler is based on work done by J. Larus and T. Ball [8] and, contrary to DCPI and Morph, is an instrumentational approach to data sampling. Path Profiler is much like GProf but is much richer in detail. Here, the problem is the overhead. By the authors' own evaluation, the overhead is around 30%. On a normal running system, this is often not tolerable. As reported in [6], Quantify uses techniques similar to Path Profiler to provide profiling information, but its capabilities remain confined to analyzing one process at a time. Moreover, its overhead is unpublished.

All these profiling systems provide detailed analysis of one or many processes, but fail to provide information on system dynamics. Yet, two of these systems bare a significant contribution that is used by LTT and that provides an efficient way to collect a high volume of data without hindering system performance. DCPI and Morph use a combination of kernel modifications, driver, and daemon to collect the necessary data for their operation. This is a departure from the traditional practice of using new system calls or adding entries to the */proc* directory and provides a wealth of opportunities for profiling and measuring systems.

On the other end of the spectrum, we find such tools as UNIX ps and top and the Windows NT Performance Monitor [15]. The former often use the content of the */proc* directory to present the user with system statistics. The later uses system calls that enable the user to read into kernel counters. Both are based on sampling or on crude event counting. Both lack the possibility to track the order in which events occurred or their details. They cannot, therefore, be relied on to offer a correct appreciation of the underlying system's dynamics.

In a category of their own, we find specialized tools that enable the user to track key system events while preserving their order of occurrence and some details. These tools have primarily been used for debugging or for security auditing. They have not been designed for measurement or characterization. For instance, WindView [2], DejaView [3] and the Hyperkernel Trace Utility [4] are primarily designed to help embedded system designers understand the dynamics of their systems in order to clarify synchronization and resource usage problems. The security auditing tools are limited by their own purpose. That is, their usage cannot be generalized to purposes other than security auditing. Neither category is intended to provide the user with system performance analysis. Moreover, the debugging tools, which are far more elaborate than the security oriented tools mentioned, are all proprietary. Therefore, they cannot be improved and extended in the same way by the user community.

Another approach is used in SimOS [13] which simulates the hardware on which an operating system runs in order to retrieve information regarding its behavior and how applications interact. This, though, remains a simulated system and cannot be used on production systems. In this regard, LTT is not unique in the information it provides. Yet, it is the only tool available for a mainstream operating system that has been designed from the ground up in order to provide the user with not only a view of the dynamic behavior of the system, but also a characterization of its behavior and a measurement of the different latencies suffered by and because of the different processes running. Moreover, it is an open-source project and can, therefore, be modified and extended under the GNU General Public License [5].

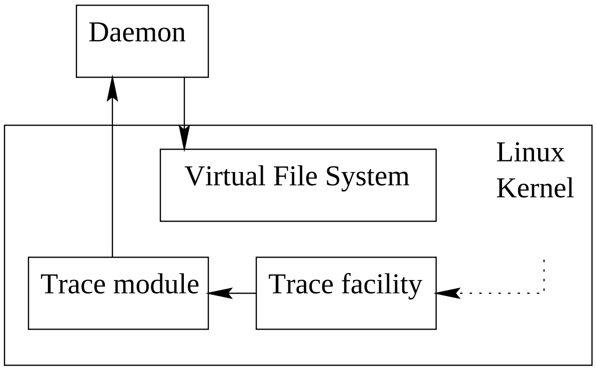

Figure 1: LTT architecture.

3 Toolkit architecture

LTT is composed of independent software modules. Each module has been designed in order to facilitate extension while minimizing performance overhead. Figure 1 represents the architecture used. Note that, for reasons of simplicity, only the architectural parts of the Linux kernel relating to LTT's functionality are presented. Details for the Linux architecture can be found in [10, 14]. Details for a general UNIX system architecture can be found in [16, 7]. The arrows indicate the flow of information through the different modules making up LTT. The primary source of information being the instrumented kernel. The source of this primary information is not attributed to a kernel component in particular since many components convey trace information. Therefore, not all the modules producing information are illustrated, to simplify presentation, but they are all discussed in detail below.

Basically, events are forwarded to the trace module via the kernel trace facility. The trace module, visible in user space as an entry in the */dev* directory, then logs the events in its buffer. Finally, the trace daemon reads from the trace module device and commits the recorded events into a user-provided file.

Section 3.1 discusses the kernel trace facility. Section 3.2 discusses the instrumentation of the kernel. Section 3.3 discusses the trace module. Section 3.4 discusses the trace daemon. Finally, section 3.5 discusses the data analysis and presentation software that come with LTT.

3.1 Kernel trace facility

The kernel trace facility is an extension to the core kernel facilities. Its function is to provide a unique entry point to all the other kernel facilities that would like an event to be traced. However, it does not log the events. Rather, it forwards the trace request to the trace module.

To achieve such functionality, the trace module has to register itself with the trace facility upon system startup, if it was compiled as part of the kernel. When compiled and loaded as a separate module, the registration will take place when the module is loaded. When registering, the trace module provides the trace facility with a call-back function that is to be called whenever an event occurs. If no trace module is registered, then the traced events are ignored. Furthermore, it provides the trace module with the possibility to configure the way the instruction pointer values are recorded upon the occurrence of a system call. For instance, since most system calls are done from a loaded library rather than from the code belonging to the running application, it is a desirable feature to specify either the number of call depths on the stack or an address range from which the instruction pointer should come from. Once set, the kernel will browse the stack to find an instruction pointer matching the desired constraints, whenever a system call occurs.

In summary, the kernel trace facility acts as a link between the trace module and the different kernel facilities.

3.2 Kernel instrumentation

The kernel instrumentation consists of the different events being traced in the kernel. There are different types of events. Each type of event has its own data set and, as such, the length of a trace entry varies according to the type of event recorded. Within a given type, there can be different subtypes. This allows various degrees of detail. Figure 2 lists for every type of event existing subtypes and the recorded data.

Given the number of modifications to the kernel source code and the number of files modified, a set of macros has been used instead of a direct call to the services of the trace facility. During compilation of the kernel, if the tracing is disabled in the configuration, the added code will have no effect. If

Event type	*Event subtype*	*Event detail*
Trace start	None	Trace module specific
System call entry	None	System call ID and instruction pointer
System call exit	None	None
Trap entry	None	Trap ID and instruction pointer
Trap exit	None	None
Interrupt entry	None	Interrupt ID
Interrupt exit	None	None
Scheduling change	None	Incoming task, outgoing task and outgoing task's state
Kernel timer	None	None
Bottom half	None	Bottom half ID
Process	Create kernel thread	Thread start address and PID
	Fork	PID of created process
	Exit	None
	Wait	PID waited on
	Signal	Signal ID and destination PID
	Wakeup	Process PID and state before wakeup
File system	Buffer wait start	None
	Buffer wait end	None
	Exec	File name
	Open	File name and descriptor
	Close	File descriptor
	Read	File descriptor and quantity read
	Write	File descriptor and quantity written
	Seek	File descriptor and offset
	Ioctl	File descriptor and command
	Select	File descriptor and timeout
	Poll	File descriptor and timeout
Timer	Expired	None
	Set itimer	Type and time
	Set timeout	Time
Memory	Page allocate	Size order
	Page free	Size order
	Swap in	Page address
	Swap out	Page address
	Page wait start	None
	Page wait end	None
Socket communication	Socket call	Call ID and socket ID
	Socket create	Socket type and ID of created socket
	Socket send	Type of socket and quantity sent
	Socket receive	Type of socket and quantity received
Inter-process communication	System V IPC call	Call ID and entity ID
	Message queue create	Message queue ID and creation flags
	Semaphore create	Semaphore ID and creation flags
	Shared memory create	Shared memory ID and creation flags
Network	Incoming packet	Protocol type
	Outgoing packet	Protocol type

Figure 2: Kernel events traced.

tracing is selected, then the macros will be replaced by the portion of code that sets the desired values and calls the trace facility.

3.3 Trace module

The trace module is a key element of the architecture. The performance of the trace process largely depends on it. In theory, the goal of the trace module is simple: store the incoming event descriptions and deliver them efficiently to the trace daemon. In practice, its implementation is much more elaborate. There are many reasons for this.

First, the trace module must retrieve additional information for each event. This additional information consists of the time at which the event occurred and the CPU identifier. Since the absolute time is large (an 8 byte pair consisting of seconds and microseconds), only the time difference between the current event and the time at which the last buffer switch occurred is recorded. The time of the last buffer switch is necessary to support a double buffering scheme explained below. Given the size of the buffers and the frequency of event occurrences, 4 bytes suffice. Note that the trace module uses the *do_gettimeofday* kernel call in Linux in order to obtain the absolute time at which an event occurred. Under Pentium type PCs, this call uses the processor's Time Stamp Counter (TSC) which enables microsecond precision on timestamps.

Second, the trace module must be configurable. The following configuration options are possible:

- Set event data buffer size.
- Set event mask.
- Set event details mask.
- Record CPU ID.
- Track a given PID.
- Track a given process group.
- Track a given GID.
- Track a given UID.
- Set call depth at which the instruction pointer is to be fetched for a system call.
- Set an address range from which the instruction pointer is to be fetched for a system call.

The event mask is used to determine whether an event is to be logged. The event details mask is used to determine whether the details of a given event are to be recorded. Tracking a PID, process group, GID, or UID, will result in the logging of only the events occurring during the execution of a process fitting the right description. The call depth and address range for a system call have previously been discussed. Configuration is done through the *ioctl* system call.

Third, the trace module must be reentrant since an event can occur from two different levels of priority at the same time. For instance, the trace module could be dealing with a system call event when an interrupt occurs. Given the fact that interrupts have priority over any other event and that different interrupts have different priorities, the processing of the system call will be suspended and an interrupt event will call upon the trace module. To solve this issue a kernel lock is used during the critical part of the event treatment. This though raises another concern. Holding a kernel lock for an extended period of time can be costly given the frequency at which some events can occur. This is not the case with LTT, since the kernel lock used is held only for the time of the logging of the event into the module's buffer. Furthermore, since the logging procedure does not call upon any other procedure, there can be no priority-inversion problem. Each event logging is interruptible once it has released the kernel lock by any higher priority event.

In order to be accessible through the virtual filesystem as a device, the trace module provides a basic set of file operations. These are:

- *open*, upon which a pointer to the task structure of the caller is kept in order to be used by the trace module when the trace buffer is full.
- *ioctl*, to provide the daemon with a way to configure tracing.
- *release*, for the device to be released when the daemon dies or when the device is closed.
- *fsync*, to reset the driver.

To efficiently deal with the large quantity of data generated, the trace module uses double-buffering. That is, a write buffer is used to log events until it reaches its limit. At which time, the daemon is notified using a SIGIO signal. Once the write buffer has been filled, the trace module assigns it as the read buffer and uses the previous read buffer as the new write buffer. Of course data can be lost if the daemon is not rescheduled before the new write buffer fills. Since the size of the buffers used by the module is configurable, it is up to the daemon to configure the module properly. Moreover, even if the daemon has opened the trace module, events do not get logged until the daemon uses the start command via *ioctl*. Logging may be discontinued with the stop command.

3.4 Daemon

The primary function of the daemon is to retrieve the information accumulated by the trace module and store it, typically in a file. It provides the user with a number of options to control the tracing process. In addition to giving the user access to the options available from the trace module, the daemon lets the user specify the tracing duration.

Once the daemon is launched, it opens and configures the trace module, and sets a timer if a time duration was specified. Otherwise, the user must kill the daemon process manually to stop the trace. In normal operation, the daemon sleeps awaiting SIGIO signals to read from the trace module, or timer/kill events to end tracing.

Akin to the trace module, the daemon uses double-buffering. Here, though, the intent is not on preventing the loss of events but on reducing the impact on the system, due to frequent reading and writing, and reducing the pollution of the trace, due to the daemon using system resources. Therefore, when it receives a SIGIO signal from the trace module, the trace daemon reads the content of the module's buffer and appends it to the content of an internal buffer. Once this buffer is full, it is committed to file and, while doing so, a second buffer is used to record the incoming data. This second buffer will be used as the first one was. When tracing is finished, the trace module and the trace data file are closed. Note that even though the daemon writes to a file, this does not necessarily mean that information gets written to disk. In fact, given the different caching mechanisms used by Linux, the information is written to disk somewhat later in big chunks by the *kupdate* kernel thread.

Although the data collection method described above provides the detailed dynamics of the system, there is one more piece of information missing, the system's state prior to the trace start. To this end, the trace daemon will go through the */proc* directory recording for each process the following characteristics: process ID (PID), name (as given to *exec*) and parent's PID (PPID). This is done following the configuration of the trace module and the start of the trace. The information retrieved is stored in a file that will later be used by the analysis software.

3.5 Data analysis and presentation software

Unlike the other LTT components, the data analysis and presentation software is typically run off-line. It uses both the initial processes state and the trace data files created by the daemon to recreate the dynamic behavior of the system in the observed time interval.

The initial state file is used to create the process tree as it was before the trace started. The trace file is then used as the trace event database. This is accomplished using the *mmap* system call and a collection of primitives that enable the extraction of information regarding the events from the trace. These functions provide the following services: get an event's ID, its time of occurrence, the process to which it belongs and the human-readable string describing it. Also, they enable forward or backward browsing of the events trace. An important functionality they provide is to determine whether an event is a control event or not. By control event, we mean an event that results in the transition of control from or to the kernel. A system call, for instance, always marks a control transition to the kernel. A trap, on the other hand, might not mark a transition since traps can occur during the execution of kernel code. It is not the objective of this paper to present the complete conditions under which control transitions occur under Linux, but these conditions have been formalized in the course of the development of LTT in order to facilitate the reconstruction of the event graph and the computation of the different performance measures.

An important component of those primitives is the trace analysis procedure. This procedure reads the entire trace cumulating results about the behavior of the system during the trace. This is where the core of the trace processing takes place. Here, the time spent scheduled, the time spent executing process code, the time spent in system calls, the number of occurrences of the different events, etc. are computed.

These primitives serve as the basis for the analysis software which is usable either from the command line or from a GUI. The GUI front-end enables the user to browse the trace in both graphical form and list form. Both interfaces enable the presentation of the cumulated system statistics and the list of events. They also enable the user to select the events to present and the events to ignore. Moreover, the graphical front-end provides a search menu which simplifies event searching.

The possibility to view the trace in graphical form is interesting since it enables the user to easily view interactions that are often deemed complex. He can then follow the flow of control and clearly identify the different transitions.

4 Toolkit overhead

The main novelty of LTT, besides the extensibility provided by the modular open source architecture, is the extent of available information while remaining a non-invasive low overhead monitoring tool. Thus, this section concentrates on the study of the overhead caused by LTT.

To this end, section 4.1 deals with quantifying the overhead caused by LTT. Section 4.2 discusses the ramifications of the observed overhead. Section 4.3 presents LTT's memory footprint. Finally, section 4.4 discusses possible improvements.

The experimental results show that LTT is able to provide unique data sets with very little overhead on the observed systems. Typically, an observed system incurs less than 2.5% overhead when monitoring core system events.

The experiments were performed on an Intel Pentium II 350MHz processor with 128MB of main memory. The Linux distribution used is RedHat 6.0. Of course, the default kernel is replaced by the LTT modified Linux kernel. Unless stated otherwise, all jobs were run from a plain command-line outside any graphical environment. The daemon is configured to use 1,000,000 bytes buffers and the trace module is configured to use 50,000 bytes buffers. Therefore, the daemon will receive a signal from the trace module every time 50,000 bytes of trace are generated and it will commit data to file every 1,000,000 bytes of trace data. The times for these to occur vary according to the job traced and are presented below.

4.1 Overall system overhead

In order to evaluate the overhead caused by LTT to a running system, a number of modifications had to be made in order to isolate the impact caused by each component to the overall architecture. The following configurations were tested:

1. Original 2.2.13 kernel (base configuration).
2. Modified 2.2.13 kernel. Events are ignored by the kernel trace facility.
3. Modified 2.2.13 kernel. Events are logged by the trace module but the daemon is not running.
4. Modified 2.2.13 kernel. Events are logged and reported, but the daemon is not writing the data to a file.
5. Modified 2.2.13 kernel. Events are logged, reported and written.
6. Same as above, except that event details are only recorded for core kernel events [1].

On each system, a batch of jobs was issued and the elapsed time to complete every job recorded. Thereafter, job completion times could be compared and provide us with insight on which component was slowing down the system, if any. To this end, the following jobs were issued using shell scripts:

1. Complete compilation of a Linux kernel, using make. This job is CPU and file intensive.
2. Creation of a tar archive using a large number of files (800MB). This job is file intensive.
3. Compression of an archive of the Linux kernel source (50MB), using bzip2. This job is CPU intensive.
4. While running the KDE environment, the following tasks were started twice in parallel:
    (a) *netscape* loaded different web pages from 14 different sites.

[1] Incidently, the core kernel events are the events presented in Figure 2 that have no subtypes.

Conf.	*Compile*	*Archive*	*Compress*	*Desktop*
1	240.2	357.4	141.2	246.8
2	240.8	358.0	141.4	249.2
Δ	0.25 %	0.17 %	0.14 %	0.97 %
3	243.6	359.1	141.1	252.2
Δ	1.42 %	0.48 %	-0.07 %	2.19 %
4	245.7	358.1	141.6	252.9
Δ	2.29 %	0.20 %	0.28 %	2.47 %
5	246.9	365.5	141.4	258.3
Δ	2.79 %	2.27 %	0.14 %	4.66 %
6	246.3	363.6	141.6	252.9
Δ	2.54 %	1.74 %	0.28 %	2.47 %

Figure 3: Job completion times in seconds according to configuration.

(b) *acroread* opened 7 different documents.

(c) *staroffice* opened 8 different documents.

(d) *gnuplot* drew 4 functions 14 times.

This job is I/O and operating system[2] intensive. It represents the extreme case of user desktop usage. Note that for this job, the daemon is configured to use 2,000,000 bytes buffers and the trace module is configured to use 250,000 bytes buffers.

The results of these tests are presented in Figure 3 and discussed in section 4.2. For each job-configuration pair, except the ones belonging to the first configuration, the percentage difference between its completion time and the one for the base configuration is given. This facilitates further analysis. The times given are in seconds and represent the average time over 10 runs[3]. Note that due to the complexity of modern UNIX systems the times reported vary. Figure 4 presents the standard deviation for each of the times reported. Figure 5 presents the size of the traces generated and the rate of generation of the traces. These results will be discussed in section 4.3.

[2]The OS has to deal with scheduling many competing processes besides having to manage inter-process communication to the X server via sockets.

[3]Each test was actually run 11 times, the results from the first run being discarded.

Conf.	*Compile*	*Archive*	*Compress*	*Desktop*
1	0.4	1.6	0.4	0.9
2	0.6	1.9	0.5	1.9
3	0.5	1.7	0.3	1.6
4	0.5	2.0	0.5	1.1
5	1.1	1.6	0.5	3.9
6	0.5	1.3	0.5	1.8

Figure 4: Standard deviations of the measured job completion times in seconds according to configuration.

Job	*Conf.*	*One run* (MB)	*Rate* (MB/s)	*Read Freq.* (Hz)	*Write Freq.* (Hz)
Compile	5	33.4	0.135	2.83	0.14
	6	25.0	0.101	2.12	0.11
Archive	5	27.9	0.076	1.59	0.08
	6	16.5	0.045	0.94	0.05
Compress	5	1.76	0.012	0.25	0.01
	6	1.14	0.008	0.17	0.01
Desktop	5	139	0.538	2.26	0.28
	6	62.9	0.249	1.04	0.13

Figure 5: Size of event trace in megabytes, according to the job, configuration, and frequency of invocation of the trace daemon given the use of 50,000 bytes buffers for non-X applications and 250,000 bytes buffers for X applications (desktop job).

4.2 Components of system overhead

In order to fully understand the impact of the different components of LTT's overhead, the following discussion is broken up along the different processing steps, from the point where the data is generated inside the kernel to the point where it is available on disk.

It is interesting to note that instrumenting the core kernel events yields an impact below 2.5%, as can be seen by comparing the sixth configuration with the first from Figure 3, regardless of the type of job run. Of course jobs that do not call upon kernel facilities as the compression example are much less disturbed by the tracing.

4.2.1 Kernel instrumentation

The impact of instrumenting the kernel can be seen by comparing the second configuration to the original configuration. As the percentages show, the impact of instrumenting the kernel is very small, if not insignificant. The largest impact occurs during the desktop job and is due to the system resource intensive nature of the job. Otherwise, there does not seem to be any noticeable slowdown. Therefore, it is fair to say that using a traced kernel has minimal or no effect on the system's performance as long as the trace daemon has not instructed the trace module to record the events in its buffers.

4.2.2 Trace logging

The impact of logging the events generated by the kernel into the trace module's buffers can be seen by comparing the third configuration to the second configuration. Here, the impact varies according to the type of job the system is running. In the cases where there's only one process running at all times, like the archiving and compression, the overhead is negligible. In the other cases, the kernel compilation (*gcc* and *make* do, in fact, have children) and the desktop trace, the results suggest that the copying of events from the kernel to the trace module within a kernel lock *(spin_lock_irq_save)* causes scheduling contention problems.

4.2.3 Trace reading

The impact of the daemon's reading the data from the trace module to its buffers can be seen by comparing the fourth configuration with the third. The overhead of the daemon copying the data is quite small but seems correlated with the overhead of the trace logging.

4.2.4 Trace committing

The impact of the daemon writing the trace data to disk can be seen by comparing the fifth configuration with the fourth. Jobs where many processes contend for scheduling suffer the most significant overhead since they have to share the CPU with the trace daemon which has to write large quantities of data to file. The compilation involves several temporary files which are erased before they actually hit the disk, and is thus more CPU intensive than disk bound. The archiving job sees its overhead increase because it now has to contend with another task that not only needs to be scheduled, but also uses the same resource, the disk.

4.3 Space overhead

When compiled with trace support, a modified kernel increases in size by 4KB. This includes the trace module and the modifications to the kernel. Given the current resources, this increase does not cause any problem. When tracing is activated, the configurable double buffer space is added.

The amount of disk space used by the generated traces, as can be seen in Figure 5, varies depending on the job run and can be quite large. The size of the generated trace is proportional to the number of events occurring. For CPU intensive jobs (compression), the results show that the amount of data generated in minimal. For jobs that use operating system resources intensively but are not in graphical mode (compilation and archiving), the quantity of data generated is below 10 MB per minute (0.167 MB/s) .

The largest traces are generated when in the graphical environment. Here, the amount of data generated per minute is above 30 MB (0.500 MB/s) in the worst case. This is due to the constant interactions between the different applications required for the graphical display through the X server. When only core kernel events are logged, the quantity of data generated decreases to 15 MB (0.249 MB/s) per minute. Therefore 50% of the trace is composed of detailed non-core events.

4.4 Discussion

As seen in the previous section, some aspects of LTT use significant system resources, most importantly disk accesses and disk space. In order to reduce the overhead caused by reading the trace and writing it to disk, several mechanisms may be examined. One solution would be to take advantage of the *mmap* system call available in Linux. This would enable the daemon to map the memory used by the trace module's buffers directly in its address space and, thus, avoid any data copying from kernel space to user space by feeding the mapped buffer as the input for the *write* system call, in order to commit the trace to file. Another solution would be to feed the traces directly to the file system or to the block device driver (disk driver) from within the kernel without using a daemon. This, though, raises other issues which go beyond the scope of our work.

Trace size reduction both reduces disk accesses and disk space requirements simultaneously. The types and structures used to record the traced events have already been optimized for size. Nonetheless, sophisticated compression algorithms use trace regularity to achieve significant size reductions at the cost of some CPU time. Using available compression tools, a compression ratio of approximately 10 was obtained on the sample traces generated. This means that the 30MB/minute trace obtained in Figure 5 could be significantly reduced to approximately 3MB/minute. Tests would need to be run to determine the cost in CPU time of the chosen compression method.

5 Toolkit usage and comparison

This section covers the usage of LTT in characterizing system behavior. Moreover, LTT's results are compared to the results given by conventional Unix tools.

First, section 5.1 presents example traces generated and analyzed using LTT. Then, sections 5.2 and 5.3 present real case studies where LTT has been used in order to reconstruct a system's dynamic behavior and understand its performance. Last, section 5.4 compares LTT's capabilities with that of conventional Unix tools.

5.1 Trace examples

In order to illustrate the type of data LTT generates and, inherently, the reason why it obtains its level of detail and accuracy, Figure 6 presents a sample trace where a process can be seen waiting to output to the hard disk. The first column presents the event type, the second the moment at which the event occurred[4], the third the PID of the process running when the event occurred and the fourth the details of the event[5]. The sequence of events is simple, the process tries to write to a file but has to wait for I/O. It is unscheduled until the hard disk is ready. Thereafter, a verification is made to make sure no more waiting is necessary and control is returned to the process.

[4]This usually consists of a seconds and microseconds pair but due to space constraints only the microseconds are presented here.

[5]Not all the details are given due to space constraints.

Syscall entry	(678777)	1021	SYSCALL : write
File system	(678779)	1021	WRITE : 3
File system	(679107)	1021	START I/O WAIT
Sched change	(679151)	0	IN : 0; OUT : 1021
	...		
IRQ entry	(691806)	0	IRQ : 14, IN-KERNEL
Process	(691818)	0	WAKEUP PID : 1021
IRQ exit	(691823)	0	
Sched change	(691824)	1021	IN : 1021; OUT : 0
File system	(691826)	1021	START I/O WAIT
File system	(691827)	1021	END I/O WAIT
Syscall exit	(691936)	1021	
Syscall entry	(691941)	1021	SYSCALL : sigreturn

Figure 6: Sample trace: Process waits for I/O.

Though the example is simplistic, it demonstrates the level of detail attainable using LTT. *gdb*, for instance, would have been fairly inadequate in helping us to figure out this sequence of events.

Figure 7 shows an example trace graph drawn by LTT. At the top of the window, we can see a menu and a toolbar. Three thumbnails are used to present the data about the trace. The first presents the graph. The second contains performance data. The last contains the raw list of events. The graph thumbnail is separated in two parts. The left side holds a list of all the processes that were active during the trace. The right side holds a graph that shows the flow of events in time. Vertical lines mark a transition in control, whereas horizontal lines signify time flow. The graph in Figure 7 shows how *minilogd* got scheduled right after the system clock. *minilogd* then did a *newstat* system call. The kernel did some work and gave control back to *minilogd* which then called on *poll*. The kernel did not find anything for *minilogd* and, consequently, scheduled the idle task since no other task needed the CPU.

5.2 Characterizing a normal workstation

In order to characterize a normal workstation, a set of applications must be chosen to represent the common usage of a system as a workstation. Thereafter, measures can be made using LTT and using conventional means. The accuracy of the conventional tools can then be assessed by comparing to the more accurate detailed data available in the trace.

The following applications where monitored for a period of 30 seconds: *X server*, *netscape*, *staroffice*, *x11amp* and a script running *ps* every 5 seconds.

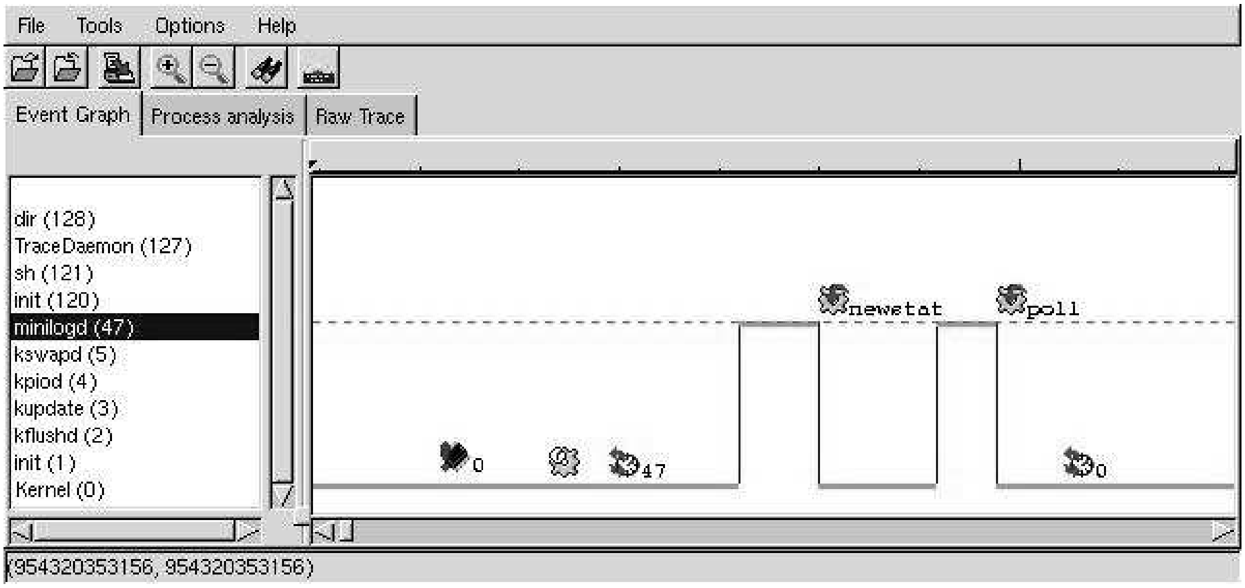

Figure 7: An example trace graph.

Application	*ps*	*User*	*Running*	*Wait I/O*
X server	8.6	12.79	15.28	0.04
netscape	3.25	15.43	17.17	1.85
staroffice	2.7	4.73	5.52	2.60
x11amp	1.15	1.62	2.19	0
ps script	0.43	0.04	0.08	0

Figure 8: Percentage of activity for each monitored application for the observed workstation.

During that period of time, LTT's trace daemon was recording events and *ps* was called upon every 5 seconds. The applications where used in a typical fashion. *netscape* was used to view 2 web sites. A document was being modified in *staroffice*. *x11amp* was playing an mp3 file. Figure 8 presents the results of the 30 seconds run. Column *ps* contains the average value of the percentages of CPU usage reported by the many runs of the *ps* utility during the test. Column *User* shows the percentage of the time the system was executing actual application code in user mode, as reported by LTT. Column *Running* shows the percentage of time the said application was scheduled, as reported by LTT. Column *Wait I/O* shows the percentage of time during which the said application was waiting for I/O.

As the results show, the conventional monitoring tool, in this case *ps*, fails to provide the observer with an accurate appreciation of the system's behavior. In fact, *ps* reports that *netscape*'s CPU usage was 3.25% whereas the real figure is 15.43%, misleading an observer into believing that the said application's CPU usage is more than 10% lower than its actual value. Moreover, there is no way to find out the amount of time during which the process is not running because it does not need to run or the amount of time during which it is waiting on an I/O resource. Also, it is important to know how much time was spent executing kernel code and how much time the system was idle. For the workstation, 59.06% was spent in kernel mode and the idle task was scheduled for 49.31% of the sample time.

5.3 Characterizing a small server

Here a small FTP server is set up to provide service to a client connected through Parallel Port IP using a null modem printer cable. The set of applications observed is *X server*, *ftp daemon*, KDE window manager *kwm*, and a script running *ps* every 5 seconds. The client requests 2 files for download from the server and the operation is monitored for 60 seconds. Figure 9 presents the results for the run. Note that there are 2 ftp entities, one for each copy of the daemon serving a corresponding file download.

As for the workstation, the results show that *ps* does not provide a clear profile of the observed system. Whereas for the workstation the time spent execut-

Application	*ps*	*User*	*Running*	*Wait I/O*
X server	7.2	0.48	3.45	0
ftp #1	13.1	0.29	10.92	0.26
ftp #2	10.8	0.32	11.77	0.63
kwm	1.63	1.33	21.2	0
ps script	3.6	0.38	2.25	0

Figure 9: Percentage of activity for each monitored application for the observed server.

ing application code was similar in most cases to the time reported by *ps*. For the server no such correlation can be found. Note that the KDE window manager has been scheduled as running for 21.2% of the duration of the sample, yet only 1.33% of the CPU time was in user mode. As for kernel code execution and time spent idle, the server spent 94.71% of its CPU time in kernel mode and the idle task was scheduled for 18.72% of the sample time.

5.4 Conventional tools comparison

The main interest of LTT being the fact that it provides information previously unavailable, it is important to compare LTT to existing analysis software. To this end, the following paragraphs compare LTT to *gdb*, *ps*, *gprof* and *time*.

Section 5.1 presented a simple trace example that showed that *gdb* was an inadequate tool in some circumstances. A more blatant example of the limits of *gdb* would be to try using it to figure out synchronization problems. Because it modifies an application's behavior, using the *ptrace* system call, it is often impossible to use *gdb* to reproduce synchronization problems, much less debug them. Moreover, synchronization problems, depending on their nature, can disappear when debugged using *gdb*. In the case of simple interactions, the crude workaround of inserting *printf*s in the debugged code is usually sufficient to help the developer solve these types of problems. But in complex software systems, this workaround is seldom adequate. In this regard, LTT fixes this type of problem by providing the developer with the exact sequence of events as they occurred on a live system. Thereafter, tracking a synchronization problem amounts to tracing the conflicting events such as IPC and socket communication and analyzing the sequence of their occurrences. Furthermore, LTT does not modify the behavior of the observed system since

```
int i, j;
void fct_A(void)
{ for(i = 0, j = 0; j < 10000000; j++)
    i++;
  printf("i's value : %d \n", i);
}
void fct_B(void)
{ for(j = 0; j < 1000000; j++)
    sched_yield();
}
int main(void)
{
  fct_A();
  fct_B();
}
```

Figure 10: Profiled source code.

events are logged in the sequence of their occurrence and the locks held to record those events are held for a very short time.

As has been demonstrated above, LTT is also very helpful in figuring out performance issues regarding an observed system. More importantly, compared with the performance data generated by conventional tools, the performance data generated by LTT matches more closely the actual behavior of the observed system of process. This has been demonstrated for *ps*. It is important to note that though *ps* is not the only tool used for performance measurement on modern Unix systems, and certainly not the most precise one, it is by far the most important because of its wide-spread usage and adoption as a legitimate way of quantifying a system's performance.

Another tool commonly used to measure performance is *gprof*. Contrary to *ps*, it is usually used to analyze the behavior of a single application. Here again, the data provided is statistical at best. To illustrate this, Figure 10 presents a portion of code that was profiled using *gprof*.

Using *gprof* we learn that the application spent a total of 250ms executing. 170ms were spent in *fct_A* and the rest, 80ms, were spent in *fct_B*. Using LTT, we find that the application actually spent 3.28s scheduled and that 2.34s were spent executing code belonging to the application. The rest of the time was spent running system code for the application. It is interesting to note that LTT reports that the application spent 972ms in cumulative calls to *sched_yield*. This has gone unnoticed by *gprof*, which simply reports that 80ms were spent in *fct_B*. Moreover, *time* corroborates LTT's results and reports that the application ran for 3.64s. *time*, though, reports that 2.71s were spent running system code and 0.61s were spent running user code. These times are calculated using the statics cumulated by the sampling done in the kernel. In essence, these results are similar to results reported by *ps*, which are known to lack precision.

The difference between the results given by LTT and the results given by *gprof* are due to the difference in the way data is acquired. *gprof* uses the system clock to sample the process' behavior. With LTT, we can see the timers used by *gprof* to sample the code going off and generating *sigreturn*s once they are done sampling. By observing the trace, we can see that the profiling timeouts very often occur within a call to *sched_yield*. Therefore, the system is running kernel code at that time and the time from that last sample is attributed to the system. This is why *time* attributes the wrong values to the different components of performance and it is the reason why *gprof* reports incorrect values. The same experiment was run using *gettimeofday* instead of *sched_yield* and gave very similar results.

6 Future directions

LTT has been available for some time through its home page (http://www.opersys.com/LTT). Many users have already used it for tracing and profiling purposes. In providing this project through the GPL license, the authors hope that it will benefit as many users as possible while offering advanced functionality.

The extensibility of LTT is provided by its openness and its modularity. Adding events to the list of events already being traced amounts to adding an identifier in the source code and placing the required trace instruction in the corresponding place in the kernel source code. Simplifying this process, an extension is currently being developed that enables the dynamic creation of event IDs and their automatic recognition by the trace analysis software. This will eliminate the need to modify LTT in any way to add traced events.

As said before, LTT comes with a versatile data analysis and presentation software tool. The later enables the user to view the event trace in a browsable graphical form. The trace is presented as a control graph where changes of control from/to the kernel are easily seen. It also provides a command-line interface enabling the user to access all of its functionality without requiring a resource intensive GUI.

There are many interesting future research avenues. For instance, given the precision of LTT's results, it would be interesting to implement a quality of service oriented kernel resource management that would use trace analysis feedback in making its future decisions. This would enable system administrators to fix quotas on the usage of most system resources.

Given LTT's ability to track detailed kernel events, it can easily be used as a component of security auditing and watchdog tools. Events matching a certain description could trigger logging or the execution of a program. Another usage would be to create a graph for tracking some of the monitored events. Rather than polling the content of */proc*, this graphing tool would be fed directly by the trace module, increasing the precision of the data. It could resemble the Windows NT Performance Monitor, though the underlying functionality would differ greatly.

Merging and interfacing with related open source projects is important to the authors. Prime and foremost, this includes eventually integrating the kernel patches to the main Linux kernel source tree. This would bring built-in tracing capabilities to every Linux user.

7 Conclusion

In this paper, we have presented a novel way of recording and analyzing system behavior. Our results have shown that LTT's overhead is minimal and that it provides unique data sets. These data sets have successfully been used to reconstruct the dynamic behavior of systems. The relatively low accuracy of conventional system monitoring tools was also shown, thus motivating the use of kernel tracing facilities, whenever a precise characterization is required.

The tools developed as part of the Linux Trace Toolkit are modular, extensible and openly available, making it easy to extend and customize them. Such tools will be crucial to the development of future computer systems due to the ever-increasing complexity of the software and hardware developed.

References

[1] Rational Software's Quantify, http://www.rational.com/products/quantify.

[2] WindRiver's WindView, http://www.windriver.com/products/html/windview2.html.

[3] QNX's DejaView, http://www.qnx.com.

[4] Nematron's HyperKernel Trace Utility, http://www.nematron.com/solutions/software/hyperkernel/hyperkernel.html.

[5] GNU General Public License version 2, http://www.gnu.org/copyleft/gpl.html.

[6] G. Ammons, T. Ball, and J. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, 1997.

[7] M. Bach. *The Design of the Unix Operating System*. Prentice Hall, 1986.

[8] T. Ball and J. Larus. Efficient Path Profiling. In *Proceeding of MICRO-29*, 1996.

[9] J. Anderson et al. Continuous Profiling: Where Have All the Cycles Gone. In *16th ACM Symposium on Operating Systems Principles*, 1997.

[10] M. Beck et al. *Linux Kernel Internals: Second Edition*. Addison-Wesley, 1998.

[11] X. Zhang et al. System Support for Automatic Profiling and Optimization. In *16th ACM Symposium on Operating Systems Principles*, 1997.

[12] S. Graham, P. Kessler, and M. McKusick. gprof: A call graph execution profiler. In *SIGPLAN Notices*, 1982.

[13] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. In *ACM Transactions on Modeling and Computer Simulation*, volume 7, pages 78–103, January 1997.

[14] A. Rubini. *Linux Device Drivers*. O'Reilly, 1998.

[15] David A. Salomon. *Inside Windows NT: Second Edition*. Microsoft Press, 1998.

[16] Uresh Vahalia. *Unix Internals: The New Frontiers*. Prentice Hall, 1996.