Universität Hamburg
Department Informatik
Scientific Computing, WR

Performance analysis using Great Performance Tools and Linux Trace Toolkit next generation

Seminar Paper

Seminar: Performance analysis in Linux

Heye Vöcking
Matr.Nr. 6139373
9voeckin@informatik.uni-hamburg.de

Tutor: Julian Kunkel

April 3, 2012

Contents

Abstract Todays systems are getting more and more complex. Especially through the trend to software running on multi-core setups. In order to find bugs and improve performance of the software we develop, we need good tools for tracing the programs execution efficiently. By monitoring, recording, and analyzing the programs behavior we can diagnose problems, tune for more performance, and understand the interaction of our software with libraries and the operating system better. In this work we want to present the Great Performance Tools and the Linux Trace Toolkit next generation.

1 Introduction

When we develop software we often encounter bugs or code that performs poorly. To debug and tune our code we can observe the programs execution. There are several more or less efficient ways to accomplish this. Tracing is one of them which can be performed using LTTng, another way is profiling which is possible with gperftools.

1.1 Overview

At first we will mention some tools available for tracing as well as discuss and compare the Great Performance Tools (section 4) and the Linux Trace Toolkit next generation (section 5) in detail. We will show what features they offer, describe how to install and integrate them, show how to use them, and give an insight of how they work. The areas of application are discussed in section 6.

1.2 Motivation

There is more to developing software than just to make it work. Even though the processing power of computers is still rising a developer should write efficient code and use the advantage of multi-core systems. Efficiency is a crucial subject because it saves time and energy during runtime. Another important part is software safety. To avoid dead locks, segmentation faults, or memory the programmer needs a good understanding of the software he writes and the libraries he uses. In order to Here we want to show what tools can help developers to write efficient code.

1.3 Related Tools

gprof (1982) gprof extended the 1979 released Unix tool “prof”, by providing a complete call graph analysis. [gpr12]

EDGE (1999) The Mentor Embedded EDGE Developer Suite is an IDE built upon the Eclipse framework and provides tools that give an insight into the kernel and a debugger as well as some other features. [Men99]

QNX Momentics Tool Suite (2001) This tool suite can be used to get a view of real-time interactions and memory profiles as well as porting software from single- to multi-core systems. [QNX01]

Valgrind (2002) Originally designed to be a free memory debugging toll for Linux on x86, Valgrind is now a generic framework for creating dynamic analysis tools. [Val12]

1.3.1 Tools discussed in this work

In this work we are going to discuss and compare the Great Performance Tools (GPerfTools) and the Linux Trace Toolkit next generation (LTTng). Both projects where started in 2005, available for free on the Internet, and by now both are community driven.

2 Related Work

Peter Smith manages to achieve a 30% speed gain by fixing poorly performing code that he located by profiling his 3D application with GPerfTools and kCachegrind as described in his work [Smi11].

In his article [Tou11] Toupin describes how to use tracing to diagnose or monitor single and multicore systems.

The author and maintainer of LTTng, Mathieu Desnoyers, has written his PhD. thesis [Des09] on software tracing with LTTng

Romik Guha Anjoy and Soumya Kanti Chakraborty evaluated in their Master Thesis [RGA10] the efficiency of LTTng for employing it within the telecommunication company *Ericsson*

3 History

Performance analysis is no recent problem. Tools for analyzing performance have been around since the early 1970s. They where usually based on timer interrupts to detect “hot spots” in the program code. The first profiler driven analysis tool was prof in 1979. In 1994 instrumentation by inserting profiling code in programs during compile time came up with the ATOM platform. Instrumentation of the Linux kernel using a method called “Kprobes” is implemented since July 2002 (Kernel 2.5.26) and still present today (Kernel 3.3). In early 2008 “Kernel Markers” where developed to replace Kprobes with no success, because the so called “Tracepoints” found their way into the Kernel not even a year later and managed to replace the Kernel Markers completely since December 2009. Tracepoints are still popular and used for example in Systemtap and LTTng which is discussed in section 5. [Mö12]

4 Great Performance Tools in depth

The Google Performance Tools project was launched in March 2005 [SR08]. In 2011 Craig Silverstein, in the gperftools project better known as “csilvers”, who was the main developer, decided to step down. As a part of that Google decided to make the project completely community run, therefore with the beginning of version number 2 (released in February 2012), the tools changed their name from “google-perftools” to “gperftools”, where the “g” now stands for “great”. The gperftools are now maintained by David Chappelle who was the only other active developer in the past. [CS12]

4.1 Field of application

4.1.1 Supported operating systems

The gperftools can be used in Linux, Mac, as well as in Windows environments.

4.1.2 Supported programming languages

Supported programming languages are C and C++. But the gperftools can be used with any language that can call C code.

4.2 Install / Setup

Setup in a Linux environment is fairly straight forward. You should be able to find the corresponding packages using your package manager. In Arch Linux for example you can use a tool called yaourt to install packages from the AUR (Arch User Repository). The command “yaourt -S google-perftools” will download, compile, and install the gperftools as well as all required packages. Additionally you can install perl5, dot, and gv if you want to visualize the collected data with pprof.

4.3 How it works and how to use it

The gperftools are a set of open-source tools for profiling, leak checking, and memory allocation distributed under BSD license. They offer:

1. `cpu_profiler`, a profiler for monitoring the performance of functions
2. `tcmalloc`, a thread caching memory allocator as a fast replacement for
3. `heap_checker`, a heap checker for leak detection
4. `heap_profiler`, a profiler for monitoring heap allocations `CMalloc`

We will discuss these below.

4.3.1 CPU Profiler

The CPU profiler collects information during runtime, namely the number of function calls with caller and callee methods and classes. There are multiple ways to use the CPU profiler:

1. Through linking `-lprofiler` into your executable.
2. Through `$LD_PRELOAD` by using `% env LD_PRELOAD="/usr/lib/libprofiler.so" /path/to/binary`. But doing so is not recommended.

Note that CPU profiling has to be turned on manually before execution. This is done by defining the environment variable `$CPUPROFILE` to the file you want to save the collected information to. Surround the code you want to profile with `ProfilerStart("profile name")` and `ProfilerStop()`. Several more advanced functions are available like `ProfilerFlush()` and `ProfilerStartWithOptions()`.

Using the environment variables
`$CPUPROFILE_FREQUENCY`
and
`$CPUPROFILE_REALTIME`
you can control the behavior of the CPU profiler during runtime.

See section 4.4 for analysis of the generated profiles. [Ghe08]

4.3.2 TCMalloc

`tcmalloc` acts as a replacement for the standard `malloc` provided in clib. It is designed to provide an efficient memory management for multi-threaded applications, it reduces memory overhead especially for small objects. The heap checker as well as the heap profiler depend on `tcmalloc`. For brevities sake we are not looking further into how tcmalloc works in the inside, because we are more focusing on performance analysis. But the interested reader might want to take a look it this [SG07].

There are multiple ways to use tcmalloc:

1. Through linking `-ltcmalloc` into your executable. (The heap leak checker depends on tcmalloc so in order to use it you have to link `-ltcmalloc` into your binary)

2. Through `$LD_PRELOAD` by using `% env LD_PRELOAD="/usr/lib/libtcmalloc.so" /path/to/binary`. But doing so is not recommended.

4.3.3 Heap Leak Checker

As the name says, the heap leak checker helps finding memory leaks during execution. So it records every allocation and checks on exit if all allocated memory has been freed. If not all memory has been freed it reports a leak.

In order to run a program with the heap leak checker you have to use tcmalloc, see section 4.3.2 and you have to define the `$HEAPCHECK` environment variable with the mode you want to use e.g. `HEAPCHECK=normal`.

There are 4 different modes of leak checking:

1. `minimal` ignores all allocations before `main()`.

2. `normal` reports all objects allocated during runtime that are still “alive” at the end of execution.

3. `strict` is basically like `normal` but also reports memory leaked in global destructors.

4. `draconian` is tracking every bit of memory. If not all allocated memory has been freed it reports a leak.

5. `as-is` offers a very flexible mode, where you can choose all options precisely by setting environment variables.

6. `local` will activate leak checking not for the whole program, but explicitly. In order to do so, you create a `HeapLeakChecker` object where you want to start leak checking, and end it with a call to `NoLeaks()`. In order to turn leak checking on you

still have to start the program with `HEAPCHECK=local`.

You can also explicitly turn off leak checking for certain objects or code pieces. To do this, bracket the code you want to ignore with the following heap-checking construct:

```
...
{
  HeapLeakChecker::Disabler disabler;
  <code you want to ignore>
}
...
```

This will lead the heap leak checker to ignore all objects and any objects reachable from these objects, including any routines called from the bracketed region to be ignored.

It is also possible to turn of leak checking for single objects and everything reachable by following pointers inside these objects. To do this simply pass the pointer of the object you want to ignore to `IgnoreObject()`. By passing the pointer to `UnIgnoreObject()` the effect can be undone.

Via environment variables the leak checker can be tuned when run in `as-is` mode. For a complete overview of all available options have a look at [Lif07].

The leak report is printed on the command-line, but it can also be visualized by pprof, see section 4.4 for details. [Lif07]

4.3.4 Heap Profiler

The heap profiler can be used, to find out more about the memory management of a certain program. This is useful for analyzing the program heap, finding memory leaks and make out places that allocate a lot of memory. The profiler keeps track of all allocations done by `malloc`, `calloc`, `realloc`, and `new`.

To use the heap profiler `tcmalloc` must be used. Take a look at section 4.3.2 to see how to include `tcmalloc` in your application. You also have to set the environment variable `$HEAPPROFILE` to the path where you want to save your profile to.

In your code you can call `HeapProfilerStart("prefix")` to start heap profiling. The profiled data will be written in to files called `prefix.XXXX.heap`, where `XXXX` is a whole number starting at `0000` and increasing with every periodic dump. With `HeapProfileDump()` or `GetHeapProfile()` you can examine the profile. Whenever the profiler is running `IsHeapProfilerRunning()` will return true. To stop profiling call `HeapProfilerStop()`.

It is possible to manually check for leaks using the profiler, but it is easier to use the heap leak checker discussed in section 4.3.3 for this task.

During execution the heap profiler will write periodical outputs to the specified path. These can be analyzed using the pprof tool, see section 4.4 for details. [Ghe12]

4.3.5 Profiler Insides

The profiler collects information about the program stack. In order to do so it grabs information provided in the program stack during execution. In the following source code we want to show the basic idea behind this.

```
main() {
  foo();
}

foo() {
  bar();
}

bar() {
  // Code inserted for profiling the
  // program stack
  int size = 10;
  void* result[size];
  int sizes[size];
  int depth = GetStackFrames(
            result, sizes, size, 1);
}
```

The method `bar()` and the call to `GetStackFrames()` will be skipped, so in this case the result returned by `GetStackFrames()` will be 2. The array `result` will contain two elements

```
result[0] = foo
result[1] = bar
```

The two elements are instruction pointers that point at the program stack where `foo` and `bar` are located.

The `sizes` array will also contain two elements

```
sizes[0] = 16
sizes[1] = 16
```

Which are the corresponding frame sizes of the functions.

4.4 Post-Processing with pprof

The program called “pprof” was written by Google Inc. in the programming language go. It is under the BSD license and its source is located inside the gperftools repository.

It can be used to analyze the collected data. It offers a number of different output formats: Raw text, postscript via gv, dot, pdf, gif, source-code listings, and disassembly. Note that you might need to install dot, gv, and ps2pdf. To reduce cluttering some edges might be dropped but this as well as the granularity of the output can be controlled at the command-line.

4.4.1 Analyzing CPU Profiles

After a program has been run with the CPU profiler enabled the output file can be analyzed using pprof. When calling pprof with the `-gv` option, a call graph annotated with timing information will be displayed. In figure 1 you can see an example of approximating the value of a gaussian distribution using the direct approximation (original_exp), a gradient, the exponential function, and a lookup table. The sizes of the nodes reflect

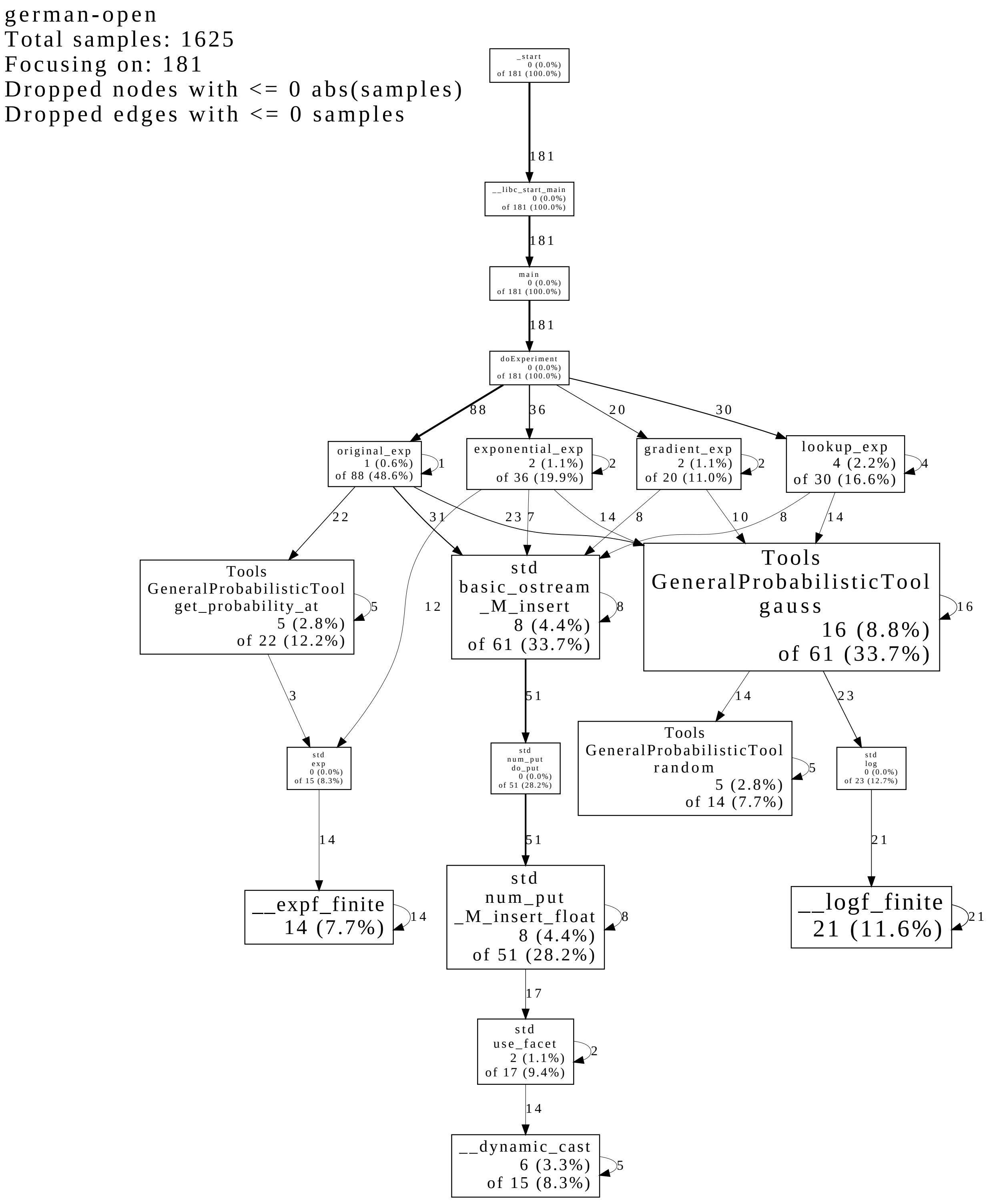

german-open
Total samples: 1625
Focusing on: 181
Dropped nodes with <= 0 abs(samples)
Dropped edges with <= 0 samples
_start 0 (0.0%) of 181 (100.0%)
181
__libc_start_main 0 (0.0%) of 181 (100.0%)
181
main 0 (0.0%) of 181 (100.0%)
181
doExperiment 0 (0.0%) of 181 (100.0%)
88
36
20
30
original_exp 1 (0.6%) of 88 (48.6%)
1
exponential_exp 2 (1.1%) of 36 (19.9%)
2
gradient_exp 2 (1.1%) of 20 (11.0%)
2
lookup_exp 4 (2.2%) of 30 (16.6%)
4
22
31
23
7
14
8
10
8
14
12
Tools GeneralProbabilisticTool get_probability_at 5 (2.8%) of 22 (12.2%)
5
std basic_ostream _M_insert 8 (4.4%) of 61 (33.7%)
8
Tools GeneralProbabilisticTool gauss 16 (8.8%) of 61 (33.7%)
16
3
51
14
23
std exp 0 (0.0%) of 15 (8.3%)
std num_put do_put 0 (0.0%) of 51 (28.2%)
Tools GeneralProbabilisticTool random 5 (2.8%) of 14 (7.7%)
5
std log 0 (0.0%) of 23 (12.7%)
14
51
21
__expf_finite 14 (7.7%)
14
std num_put _M_insert_float 8 (4.4%) of 51 (28.2%)
8
__logf_finite 21 (11.6%)
21
17
std use_facet 2 (1.1%) of 17 (9.4%)
2
14
__dynamic_cast 6 (3.3%) of 15 (8.3%)
5

Figure 1: Output produced by pprof

their time consume. Each node contains the function name (first line), the time spend in the function itself (second line), the total time which is composed of the cumulated time of the functions called and the time displayed in the second line. The time is provided in units where one unit is equivalent to one sample. For example: 88 units at 100 samples per second equate to 0.88 seconds spend in the corresponding function.

The edges are drawn from the caller to the callee, located at the arrow of the connecting edge. The weight of the edges reflect the time spend in the function that is being called. Some edges and nodes might be dropped to reduce clutter, so the sum might not always add up to 100 percent. [Ghe08]

4.4.2 Analyzing Memory Leaks

To help track down memory leaks pprof can be used to display a call graph of the reported leak. [Lif07]

4.4.3 Analyzing Heap Profiles

The pprof tool can also be used to analyze the profiles generated by the heap profiler. Calling pprof with the `-gv` option will open a `gv` window displaying a weighted directed graph showing what portions of the code allocated how much memory.

To find gradual memory leaks you can use pprof and compare two profiles. The memory usage in the first profile will be subtracted from the second profile. [Ghe12]

5 Linux Trace Toolkit next generation in depth

The LTTng was written by Mathieu Desnoyers after he started to maintain the predecessor called “Linux Trace Toolkit” since 2005. It offers both user space, as well as kernel space tracing. It claims to trace with a very low impact on the system. It is able to output the recorded data directly to disk or to the network. An overview over the whole LTTng architecture can be achieved by looking at figure 2.

5.1 Field of application

5.1.1 Supported operating systems

LTTng is only available for Linux.

5.1.2 Supported architectures

LTTng has at the moment full support for x86-32, x86-64, PowerPC 32/64, ARMv7 OMAP3, MIPS, sh, sparc64, s390 with a timestamp precision of typically $\sim 1ns$ (cycle counter). Other Linux architectures are supported as well, but LTTng might only have a limited timestamp precision on these. LTTng also supports arbitrary architecture endianness. [LTT12]

5.1.3 Supported programming languages

Supported programming languages are C and C++. But LTTng can be used with any language that can call C code, for example it can be used with Java or Erlang through the LTTng VM adaptor.

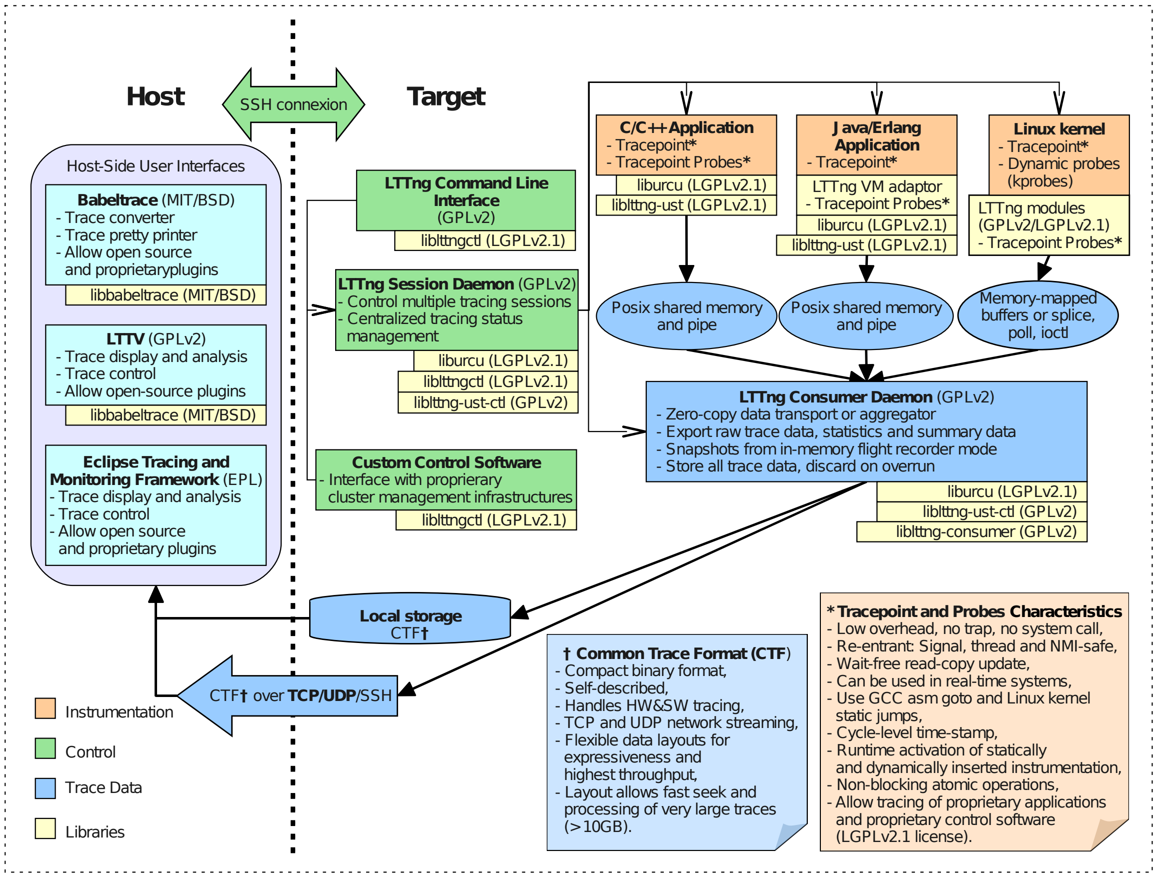

Host
SSH connexion
Target
Host-Side User Interfaces
Babeltrace (MIT/BSD)
- Trace converter
- Trace pretty printer
- Allow open source and proprietaryplugins
libbabeltrace (MIT/BSD)
LTTV (GPLv2)
- Trace display and analysis
- Trace control
- Allow open-source plugins
libbabeltrace (MIT/BSD)
Eclipse Tracing and Monitoring Framework (EPL)
- Trace display and analysis
- Trace control
- Allow open source and proprietary plugins
LTTng Command Line Interface (GPLv2)
liblttngctl (LGPLv2.1)
LTTng Session Daemon (GPLv2)
- Control multiple tracing sessions
- Centralized tracing status management
liburcu (LGPLv2.1)
liblttngctl (LGPLv2.1)
liblttng-ust-ctl (GPLv2)
Custom Control Software
- Interface with proprierary cluster management infrastructures
liblttngctl (LGPLv2.1)
C/C++Application
- Tracepoint*
- Tracepoint Probes*
liburcu (LGPLv2.1)
liblttng-ust (LGPLv2.1)
Java/Erlang Application
- Tracepoint*
LTTng VM adaptor
- Tracepoint Probes*
liburcu (LGPLv2.1)
liblttng-ust (LGPLv2.1)
Linux kernel
- Tracepoint*
- Dynamic probes (kprobes)
LTTng modules (GPLv2/LGPLv2.1)
- Tracepoint Probes*
Posix shared memory and pipe
Posix shared memory and pipe
Memory-mapped buffers or splice, poll, ioctl
LTTng Consumer Daemon (GPLv2)
- Zero-copy data transport or aggregator
- Export raw trace data, statistics and summary data
- Snapshots from in-memory flight recorder mode
- Store all trace data, discard on overrun
liburcu (LGPLv2.1)
liblttng-ust-ctl (GPLv2)
liblttng-consumer (GPLv2)
Local storage CTF†
CTF† over TCP/UDP/SSH
Instrumentation
Control
Trace Data
Libraries
† Common Trace Format (CTF)
- Compact binary format,
- Self-described,
- Handles HW&SW tracing,
- TCP and UDP network streaming,
- Flexible data layouts for expressiveness and highest throughput,
- Layout allows fast seek and processing of very large traces (>10GB).
* Tracepoint and Probes Characteristics
- Low overhead, no trap, no system call,
- Re-entrant: Signal, thread and NMI-safe,
- Wait-free read-copy update,
- Can be used in real-time systems,
- Use GCC asm goto and Linux kernel static jumps,
- Cycle-level time-stamp,
- Runtime activation of statically and dynamically inserted instrumentation,
- Non-blocking atomic operations,
- Allow tracing of proprietary applications and proprietary control software (LGPLv2.1 license).

Figure 2: LTTng 2.0 architecture [LTT12]

5.2 Install / Setup

The installation of LTTng 2.0 is fairly simple, given you are using Ubuntu. You can simply install it from the PPA (Personal Package Archives) and reboot. That's it.

5.3 How it works

LTTng consists of three parts: The kernel part which controls kernel tracing, the user space command-line application called *lttctl*, and the user space daemon called *lttd* which waits for the data to write it on the disk. Moreover LTTng is modular, it consists of 5 modules:

1. *ltt-heartbeat* which generates periodic events for monotonic increasing timebase.
2. *ltt-facilities* is a collection of event types.
3. *ltt-statedump* generates events to describe kernels state at start time.
4. *ltt-core* handles a number of LTTng control events and therefore controls the previously mentioned *ltt-heartbeat*, *ltt-facilities*, and *ltt-statedump*.
5. *ltt-base* which is a built in kernel object that contains symbols and data structures.

[DD06]

All these modules work together to observe and persist the data while tracing.

5.3.1 Tracing

Tracing is an efficient way to monitor the execution of a program and record the observed data. We observe in particular operating system kernel events [YD00] such as:

- System Calls
- Interrupt Requests
- Scheduling Activities
- Network Activities

Events can have an arbitrary number of arguments, therefore the event sizes might differ. All events are ordered accurately, yet across multiple cores or CPUs, except if the hardware makes it impossible.

The data recorded saves the event as well as its attributes and the timestamp. There are several ways to do this.

- Static tracepoints are located at source code level and therefore hardwired in a compiled binary file of a program or in the kernel. They have to be activated prior to execution.
- Dynamic breakpoints can be inserted without recompiling via a system call, trap, or dynamic jump, and added in the Linux kernel via kprobe.

A breakpoint interrupt causes big overhead, so LTTng uses Tracepoints to record raw data which is analyzed in a post-processing step to keep the overhead during execution low. [Tou11] [RGA10]

5.3.2 Tracepoints

A tracepoint is a small piece of code basically similar to this:

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

Tracepoints are already available in the Linux kernel and in many Linux applications. But in order to use them they have to be activated prior to execution and connected to a probe. A probe is the function called by an activated tracepoint during execution. See figure 3 for a digram of the tracing architecture. With custom probes tracing can be bound to a condition. In order to keep the overhead of tracepoints to a minimum, they are implemented very carefully so the magnitude of an inactive tracepoint is practically zero. In the kernel they use code modification and in user space the evaluation on booleans is fairly fast. When active a the tracepoints overhead is comparable to a C function call. [Tou11]. [LTT12]

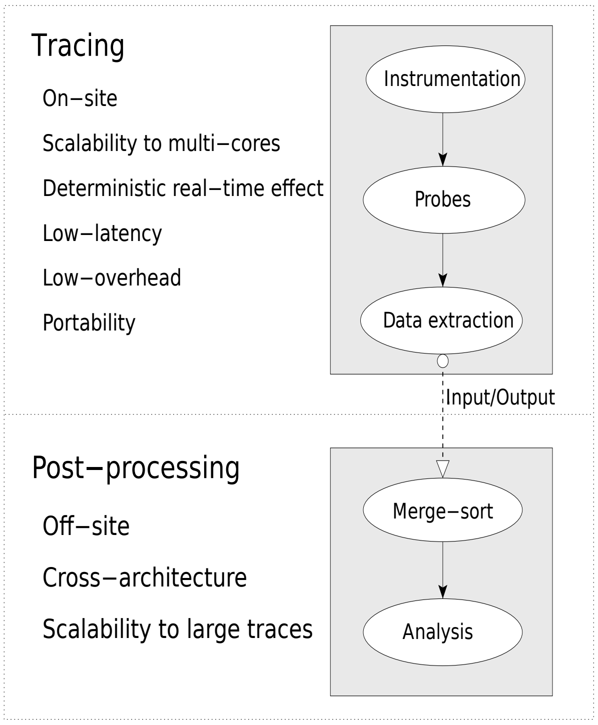


Figure 3: Trace Architecture Diagram [Des09] Probes are called by the instrumented portions of a program or the kernel. They collect the data, hand it over to the lttd which takes care of the data persistence. After the execution is over the collected data is prepared for the analysis and displayed in an appropriate viewer.

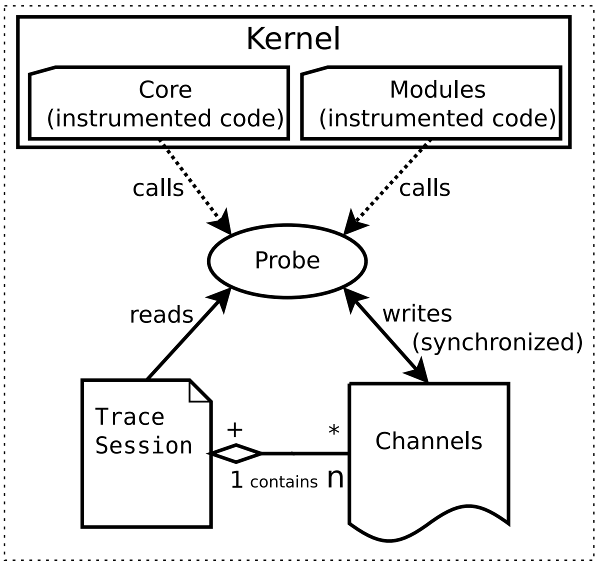


Figure 4: Tracer Components Overview [Des09] The notation used for the edge connecting Trace session and Channels indicates, that a trace session can contain multiple channels.

5.3.3 Probes, Trace Sessions, and Channels

A probe can be connected to more than one tracepoint. When an active tracepoint is reached, it calls the probe it is connected to. The probe then reads the trace session and writes the events into a channel. The probes are not implemented through traps or system calls and are therefore very efficient.

A trace session can be attached to several channels and keeps the trace configuration which consists of the event filters to be applied and a boolean value which states if the session is active or not.

A channel is used as an abstract I/O device. Data written into a channel is at first stored in a buffer and when needed exported by *lttd* to the disk or to the network.

An overview of the different trace components can be seen in figure 4.

5.3.4 Storage and Trace Modes

In order to minimize the performance impact on the system and ensure an effective use of memory bandwidth, LTTng is designed to take a zero copy approach. That means that no data is moved between memory locations during tracing phases. All recorded data is sent into channels which store the data in buffers and depending on the trace mode store the data independent from tracing. The data in the channels can be written on disk or streamed over the network. Furthermore it is highly optimized to keep it as compact as possible.

The following Tracing modes are available:

write-to-disk the recorded data is written, whenever the circular buffer is full.

Therefore no trace data is lost.

flight recorder the recorded data is not written when the circular buffer is full, but whenever the recording does not impact performance or after the trace has ended. The flight recorder mode activation is per-channel. Some data might be overwritten in this mode.

Still under development are stream to remote disk and live monitoring, remote or local. Since I/O operations are costly they are not done by the probes, but by threads specialized for I/O operations which can persist the data while tracing is in progress or after the trace session has been ended. [RGA10]

5.4 Post-Processing and Analysis

The trace dump must not necessarily be viewed in the same environment where the data has been recorded. The trace output itself is a binary file which is self describing and therefore portable.

5.4.1 Babeltrace

Babeltrace offers libraries for parsing traces to human readable text logs and vice versa. It is distributed by EfficiOS which is run by Mathieu Desnoyers, the author and maintainer of LTTng. Babeltrace works with logs in CTF (Common Trace Format) and is command-line based, so it offers no GUI. For a visual representation of a trace the LTT Viewer can be used.

5.4.2 LTT Viewer

The LTT Viewer, or short LTTV, can be used to view recorded traces by either the kernel space tracer or the user space trace. It is written in C using glib and the GTK and independent from LTTng. Developers can easily extend the LTTV by writing plug-ins. The GUI of LTTV offers control flow charts and resource viewers. In figure 5 you can see a trace recorded by LTTng of a program which performs a one second sleep. At the position of the cursor we observe a syscall and page faults by different processes. In the “Traceset” view we can select statistics of the whole system and of our program. For example Traps, Interrupts, and Syscalls.

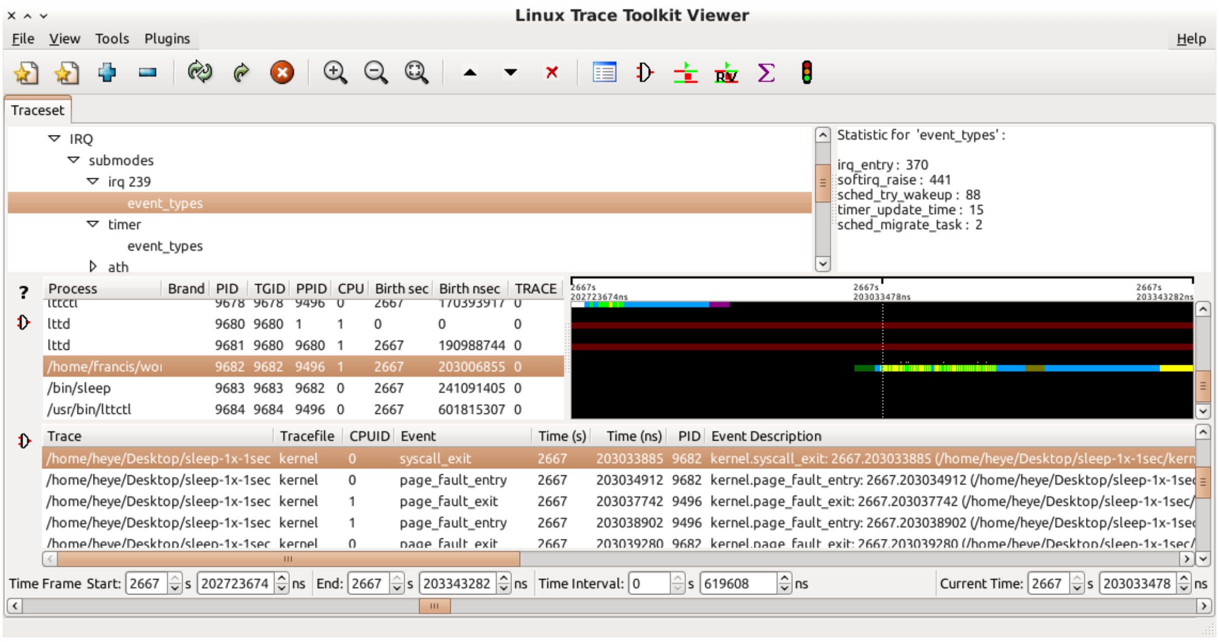
As of now, it is not possible to view traces generated with LTTng 2.0 because they are saved in CTF which is not yet supported by LTTV.

5.4.3 IDE integration

Through the TMF (Tracing and Monitoring Framework, also known as “LTTng plug-in”) traces can be viewed and analyzed directly in eclipse. TMF is part of the Linux Tools project and can be installed using the built in software manager in eclipse. It requires the LTTng trace reading library to be installed. It offers the following functions:

Control Flow visualizes the state transition of the processes.

Resources visualizes the state transition of system resources.



Statistic for 'event_types':

irq_entry: 370
softirq_raise: 441
sched_try_wakeup: 88
timer_update_time: 15
sched_migrate_task: 2

Process	Brand	PID	TGID	PPID	CPU	Birth sec	Birth nsec	TRACE
lttctl		9678	9678	9496	0	2667	170393917	0
lttd		9680	9680	1	1	0	0	0
lttd		9681	9680	9680	1	2667	190988744	0
/home/francis/wo		9682	9682	9496	1	2667	203006855	0
/bin/sleep		9683	9683	9682	0	2667	241091405	0
/usr/bin/lttctl		9684	9684	9496	0	2667	601815307	0

2667s 202723674ns 2667s 203033478ns 2667s 203343282ns

Trace	Tracefile	CPUID	Event	Time (s)	Time (ns)	PID	Event Description
/home/heye/Desktop/sleep-1x-1sec	kernel	0	syscall_exit	2667	203033885	9682	kernel.syscall_exit: 2667.203033885 (/home/heye/Desktop/sleep-1x-1sec/kern
/home/heye/Desktop/sleep-1x-1sec	kernel	0	page_fault_entry	2667	203034912	9682	kernel.page_fault_entry: 2667.203034912 (/home/heye/Desktop/sleep-1x-1sec
/home/heye/Desktop/sleep-1x-1sec	kernel	1	page_fault_exit	2667	203037742	9496	kernel.page_fault_exit: 2667.203037742 (/home/heye/Desktop/sleep-1x-1sec/
/home/heye/Desktop/sleep-1x-1sec	kernel	1	page_fault_entry	2667	203038902	9496	kernel.page_fault_entry: 2667.203038902 (/home/heye/Desktop/sleep-1x-1sec
/home/heye/Desktop/sleep-1x-1sec	kernel	0	page_fault_exit	2667	203039280	9682	kernel.page_fault_exit: 2667.203039280 (/home/heye/Desktop/sleep-1x-1sec/

Time Frame Start: 2667 s 202723674 ns End: 2667 s 203343282 ns Time Interval: 0 s 619608 ns Current Time: 2667 s 203033478 ns

Figure 5: An example trace recorded with LTTng analyzed with LTTV

Statistics provides statistics on occurrences of events.

5.5 Performance Impact

The performance impact of LTTng is reasonably small. On a busy system the CPU time added by tracing under medium and high load goes from 1.54% to 2.28%. Only under very high load the impact is noticeable with 9.46%. An overview of the impact under different scenarios can be seen in table 1. The load in the third column shows the usage of the CPU by all processes including probes and lttd. Adding up the time in the probes and the lttd column gives us the whole overhead caused by LTTng. The data rate is the amount of data outputted and processed by the lttd (LTTng Daemon in user space) every second. And Event lists the number of events recorded every second. [DD06]

6 Areas of Application

Google is using their gperftools themselves for profiling, leak checking, as well as memory allocation. The LTTng is used by a variety of companies for performance monitoring and debugging. These include *IBM*, for solving issues in distributed file systems, *Autodesk*, for real time issues in application development, *Siemens*, for internal debugging and performance monitoring. Furthermore the LTTng included in the packages of many Linux distributions. To only name a few: *Ubuntu*, *Montavista*, *Wind River*, *STLinux* and *Suse*. [RGA10] [LTT12]

7 Summary

We have mentioned several different tools for performance analysis and discussed the Linux Trace Toolkit next generation and the Great Performance Tools in detail. We explained how to use them and gave an insight

Load size	Test Series	CPU time (%)			Data rate (MiB/s)	Events/s
		load	probes	lttd		
Small	mozilla (browsing)	1.15	0.053	0.27	0.19	5,476
Medium	find	15.38	1.150	0.39	2.28	120,282
High	find + gcc	63.79	1.720	0.56	3.24	179,255
Very high	find + gcc + ping flood	98.60	8.500	0.96	16.17	884,545

Table 1: LTTng benchmarks under different loads [DD06]

to some basic functionality. The use of the gperftools is very straight forward and the collected information can be analyzed using pprof by outputting it to many different formats. There are several tools for the analysis of data collected by the LTTng.

8 Conclusion

The need for more speed drives the trend to systems that are getting more and more complex and should yet perform faster and more efficient. At the same time the complexity of software running on those system is also increasing. In order to improve comprehension of the behavior of the systems we need to use tools that help us understanding how the different parts work together we can use tools that help us. In this work we have discussed two of the many tools offered for profiling and tracing. We explained how the LTTng and the gperftools work and further showed how use them and how to analyze their output.

9 Outlook

With the demand for more performance profiling and tracing will remain a crucial task for speed improvement. When we look at the history of tracing we can see that due to the rising demand tracing is getting more and more comfortable. Also the performance impact of tracing is getting smaller and smaller. But it is not only the speed that matters but also the safety and security of software. It is hard to imagine debugging software with interacting threads by hand and since the drift to parallel programs is growing we will depend on good tracing and profiling tools in the future.

References

[CS12] Craig Silverstein, David C.: Great Performance Tools. (2012). `http://code.google.com/p/gperftools/`

[DD06] Desnoyers, Mathieu ; Dagenais, Michel R.: The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. In: *Linux Symposium* (2006)

[Des09] Desnoyers, Mathieu: *Low-Impact Operating System Tracing*, Université de Montréal, Diss., December 2009

[Ghe08] Ghemawat, Sanjay: Gperftools CPU Profiler. (2008), May. `http://gperftools.googlecode.com/svn/trunk/doc/cpuprofile.html`

[Ghe12] Ghemawat, Sanjay: Gperftools Heap Profiler. (2012), February. `http://gperftools.googlecode.com/svn/trunk/doc/heapprofile.html`

[gpr12] gprof: GNU prof. (2012). `http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html`

[Lif07] Lifantsev, Maxim: Gperftools Heap Leak Checker. (2007), July. `http://gperftools.googlecode.com/svn/trunk/doc/heap_checker.html`

[LTT12] LTTng: http://www.lttng.org. (2012)

[Men99] Mentor: *An In-Depth and Comprehensive look at The Mentor Embedded Linux Development Platform*, 1999

[Mö12] Möller, Hajo: Instrumentierung des Kernels. (2012)

[QNX01] QNX: *QNX*. `http://www.qnx.com/products/tools/`. Version: 2001

[RGA10] Romik Guha Anjoy, Soumya Kanti C.: *Efficiency of LTTng as a Kernel and Userspace Tracer on Multicore Environment*, School of Innovation, Design and Engineering Mälardalen University Västerås, Sweden, Diplomarbeit, June 2010

[SG07] Sanjay Ghemawat, Paul M.: TCMalloc : Thread-Caching Malloc. (2007), February. `http://gperftools.googlecode.com/svn/trunk/doc/tcmalloc.html`

[Smi11] Smith, Peter: Using the CPU to Improve Performance in 3D Applications. 2011. – Forschungsbericht

[SR08] Sean Rose, Riley A.: *An Evaluation of Performance Analysis Tools*, University of Ottawa, Diplomarbeit, April 2008

[Tou11] Toupin, Dominique: Using Tracing to Diagnose or Monitor Systems. In: *Software, IEEE* 28 issue:1 (2011), S. 87–91

[Val12] Valgrind: (2012). `http://www.valgrind.org`

[YD00] Yaghmour, Karim ; Dagenais, Michel R.: The Linux Trace Toolkit. In: *Linux Journal* (2000)