Userspace Application Tracing with Markers and Tracepoints

Jan Blunck
SUSE Linux Products GmbH
jblunck@suse.de

Mathieu Desnoyers
École Polytechnique de Montréal
mathieu.desnoyers@polymtl.ca

Pierre-Marc Fournier
École Polytechnique de Montréal
pierre-marc.fournier@polymtl.ca

Abstract

Today, multiple tracing infrastructures are available for the Linux operating system. Most of them are focused on tracing the execution of kernel code. In some cases it is helpful to also have insight into the userspace application's activity to fully understand the system behavior. A solution that is capable of tracing both userspace and kernel mode code is therefore necessary.

A new userspace tracing solution based on the *Linux Trace Toolkit Next Generation (LTTng)* [1] fills this gap and integrates seamlessly into the existing LTTng analysis applications available today. This paper describes the architecture of the LTTng Userspace Tracer and how it can be used in applications.

1. Introduction

Today's Linux distributions offer a wide range of tools for tracing. Users and developers can choose between SystemTap, OProfile, Ftrace or Ptrace just to name a few. However, none of these cover tracing of userspace applications and the execution of kernel code in parallel.

The new LTTng userspace tracing framework tries to fill this gap. It provides the sames instrumentation to userspace applications that is available by the kernel components of the LTTng framework. Through the use of tracepoints and markers it is designed for low performance impact from ground up.

The output data of the userspace tracer is generated in LTTng format. It enables the trace viewer to merge the traces from userspace with the data from the kernel tracers. Therefore we have a full tracing solution that covers events from the whole system.

1.1. Previous Work

Classic userspace tracing applications, for instance `strace`, are often based on the *ptrace()* system call. The system call provides a means by which a parent process may observe and control the execution of child processes. Its primary focus is on breakpoint debugging, system call and signal tracing. Both system call and signal tracing is nowadays obtainable at a much lower cost through kernel tracing.

Profiling oriented tools like OProfile are technically able to trace kernel code as well as the running userspace applications. However, these tools do not do application specific traces. Another limitation of profiling is that it uses sampling. Therefore it can detect general tendencies, but it cannot help with detecting problems that occur in short episodes.

IBM's DProbes [2] is a low-level system debugging facility that is able to support userspace, kernel mode and interrupt mode probes. It uses technology that has already proven itself on OS/2, namely kernel probes (*kprobes*), hardware debug registers and hardware watchpoints (*dr_alloc* and *kwatchpoints*) and userspace probes (*uprobes*). The userspace probes are RPN based. DProbes comes with its own compiler that supports a C-like language to be used for probe definition. The userspace probes are system-wide global and therefore common to all instances of an object module. A disadvantage of the DProbes userspace approach is the performance impact due to the use of traps (`INT3` on Intel's x86) for breakpoints and watchpoints.

DProbes does not seem to be available for recent versions of the Linux operating system anymore. Some of its components, like *kprobes*, have only been partially available upstream or are still being worked on.

SystemTap [3] has both statically and dynamically defined probe points available for kernel mode tracing. Available in an early prototype is support for non-symbolic probe points that uses raw virtual addresses. The userspace process is identified by process id or executable path name. SystemTap makes use of KProbes internally. Hence, it is a breakpoint-based approach.

Ftrace has a feature to generate an event from userspace that shows up in the kernel trace. This is done by writing a string to the `trace_marker` file. The syscall-based approach has the same disadvantage as the breakpoint-based approach, namely significant performance overhead.

2. Userspace Tracing Components

The LTTng Userspace Tracer (UST) consists of the following components:

- C header files necessary to statically define tracepoints
- shared library that applications can link against or preload
- tracing daemon that acts as a data sink
- control application to dynamically enable/disable tracepoints

From an application programmer's point of view, the major component of the userspace tracer is the C header. It gets included for static instrumentation of interesting points in the application to trace.

The shared library includes the code to activate and arm the markers. The trace functions are hooked up with the markers and produce the LTTng formatted output and send it to the tracer daemon.

The Userspace Tracer Library (libust) itself consists of the following major components:

- userspace RCU (read-copy-update) library [5]
- userspace markers and tracepoints
- kernel compatibility headers [6]
- ltt-relay channels port

The userspace RCU library is necessary for quick read access to shared resources like control variables. Multiple papers have been written on this topic and the concept is well understood [4]. The userspace RCU library is a re-implementation of the kernel's RCU source code. It is self-contained and can be used in libraries without any modification of the application [7].

The kernel compatibility headers are necessary to get the markers and tracepoint code ported to userspace. The Linux kernel is not compiled against the system's standard startup files and C library [1]. Therefore, it comes with its own standard library code in form of C headers. The kernel header files differ from the standard C library headers and can not be used directly in userspace applications. Hence, the required kernel headers necessary for the marker and tracepoint code must be selectively ported to userspace. The kernel compatibility headers have been implemented from scratch and are released under LGPL (see also 3).

The tracer daemon acts as a trace data sink for the traced application.

2.1 Userspace Markers and Tracepoints

Markers and tracepoints are lightweight code instrumentation mechanisms. Linux Kernel Markers, created as part of the research on the LTTng, are now fully integrated in the mainline Linux kernel. Userspace Markers as described in this section are a port of the Linux Kernel Markers for use in userspace applications.

The initial motivation to create a static instrumentation infrastructure for the Linux kernel based on markers is because the KProbe mechanism (as used by DProbes and SystemTAP) adds a large performance overhead due to being based on breakpoints. Comparing static instrumentation with the KProbe-based instrumentation found in SystemTAP also provides a second motive to use static instrumentation: its ability to follow more easily source code changes in an open source project like the Linux kernel. Markers can be placed at important locations in the code. These are lightweight hooks that can pass an arbitrary number of parameters, described in a printk-like format string, to the attached probe function.

A marker placed in code provides a hook to call a function (probe) that can be provided at runtime. A marker can be "on" (a probe is connected to it) or "off" (no probe is attached). When a marker is "off" it has no effect, except for adding a tiny time penalty (checking a condition for a branch) and space penalty (adding a few bytes for the function call at the end of the instrumented function and adds a data structure in a separate section). When a marker is "on", the function provided is called each time the marker is executed, in the execution context of the caller. When the function provided ends its execution, it returns to the caller (continuing from the marker site).

This instrumentation mechanism enables the instrumentation of an application at the source-code level. Markers consists essentially of a C preprocessing macro which adds, in the instrumented function, a branch over a function call. By doing so, neither the stack setup nor the function call are executed when the instrumentation is not enabled. At runtime, each marker can be individually enabled, which makes the branch execute both the stack setup and the function call.

[1]See gcc's `-nostdinc` option.

Markers provide an `API` based on format strings, which limits type verification to basic types, but allows to easily add new markers to source code by modifying a single line.

After experimenting with the Linux Kernel Markers, two downsides with this infrastructure have been noticed. First, it only allows limited type-checking, which can be problematic if pointers must be dereferenced by the tracer code. Second, it hides the instrumentation in the source code, keeping no global registry of the instrumentation. It is therefore hard to impose name-space conventions and to keep track of instrumentation modification without monitoring the whole kernel tree.

Therefore, Tracepoints have been created to deal with this problems. They are extensively based on the Linux Kernel Markers code, with a couple of modifications to fill the missing parts. The Tracepoints are now integrated in the mainline Linux kernel and already used extensively.

The main difference between Tracepoints and Markers is that tracepoints require an instrumentation declaration in a global header. It allows type-aware verification of the tracer probes (callbacks) connected on the instrumentation site by declaring both the instrumentation call and the probe registration and unregistration function within the same declaration macro, which is aware of the types expected. Therefore, it ensures that both the caller and the callee types will match.

It also provides the needed global instrumentation registry: all the global tracepoint declarations are kept in the `include/trace/` directory of the Linux kernel tree.

2.1.1 Marker Usage

A marker, added into C code, looks like:

```
trace_mark(subsystem_event, "myint %d
     mystring %s", someint, somestring);
```

Where :

- *subsystem_event* is an identifier unique to the event
  - *subsystem* is the name of the subsystem.
  - *event* is the name of the event to mark.
- *"myint %d mystring %s"* is the formatted string for the serializer. *"myint"* and *"mystring"* are repectively the field names associated with the first and second parameter.
- *someint* is an integer.
- *somestring* is a char pointer.

This generates the assembly shown in figure 1 and 2 in the instrumented function. This infrastructure is voluntarily biased to minimize the performance overhead when tracing is disabled. Therefore, the branch executing stack setup and function call (figure 2) is marked as being unlikely using the gcc `__builtin_expect()`. This gives a hint to the compiler to move the instructions executed only when tracing is enabled outside of cache-hot function cache lines.

Connecting a function (probe) to a marker is done by providing a probe (function to call) for the specific marker through `marker_probe_register()`. Removing a probe is done through `marker_probe_unregister()`; it will disarm the probe. Calling `marker_synchronize_unregister()` is required after unregistering a probe before its dynamic shared object can be unloaded.

2.1.2 Tracepoint Usage

Two elements are required for tracepoints :

- A tracepoint definition, placed in a header file.
- The tracepoint statement, in C code.

In order to use tracepoints, you need to include `ust/tracepoint.h`.

An example of tracepoint listing is shown in Figure 3 and 4. In the example following definitions are used:

- *subsys_eventname* is an identifier unique to the event
  - *subsys* is the name of the subsystem.
  - *eventname* is the name of the event to trace.
- `TP_PROTO(int firstarg, struct task_struct *p)` is the prototype of the function called by this tracepoint.
- `TP_ARGS(firstarg, p)` are the names of the parameters, as found in the prototype.

The naming scheme `subsys_event` is suggested here as a convention intended to limit collisions. Tracepoint names are global to the application: they are considered as being the same whether they are in the core application or a dynamic loadable object.

Connecting a function (probe) to a tracepoint is done by providing a probe (function to call) for the specific tracepoint through `register_trace_subsys_eventname()`.

```
  10:   80 3d 00 00 00 00 00    cmpb   $0x0,0x0(%rip)
  17:   75 0e                   jne    27
```

Figure 1: Generated assembly in the instrumented function hot-path.

```
  27:   31 f6                   xor    %esi,%esi
  29:   48 c7 c7 00 00 00 00    mov    $0x0,%rdi
  30:   31 c0                   xor    %eax,%eax
  32:   ff 15 00 00 00 00       callq  *0x0(%rip)
  38:   eb df                   jmp    19
```

Figure 2: Generated assembly in the instrumented function cold-path.

```
#include <ust/tracepoint.h>

DECLARE_TRACE(subsys_eventname,
        TP_PROTO(int firstarg, struct task_struct *p),
        TP_ARGS(firstarg, p));
```

Figure 3: Example for a tracepoint definition.

```
#include "trace/subsys.h"

DEFINE_TRACE(subsys_eventname);

void somefct(void)
{
        ...
        trace_subsys_eventname(arg, task);
        ...
}
```

Figure 4: Example for a tracepoint statement.

Removing a probe is done through a call to `unregister_trace_subsys_eventname()`.

`tracepoint_synchronize_unregister()` must be called before the end of the dynamic shared object's destructor function to make sure there is no caller left using the probe. This, and the `rcu_read_lock()` and `rcu_read_unlock()` around the probe call, make sure that probe removal and dynamic shared object unload are safe.

The tracepoint mechanism supports inserting multiple instances of the same tracepoint, but a single definition must be made of a given tracepoint name over all the application to make sure no type conflict will occur. Name mangling of the tracepoints is done using the prototypes to make sure typing is correct. Verification of probe type correctness is done at the registration site by the compiler. Tracepoints can be put in inline functions, inlined static functions, and unrolled loops as well as regular functions.

2.2. Tracing Daemon

A daemon, `ustd`, collects the trace data as it is produced by traced applications. When a marker is encountered, the tracing library writes the resulting event in a buffer located in a System V shared memory segment which is also mapped in the address space of the daemon. This mapping occurs when the tracing starts: the tracing library connects to the daemon through a Unix socket, sending it the shared memory segment IDs. This approach enables the daemon to send the data to disk or, eventually, to the network without copying it.

If the application ends suddenly, for example in the case of a crash, the daemon detects it and is able to recover the content of the buffers.

A daemon can be started globally for the system. It then creates a named socket in a standard location, so traced applications find it. Additionally or alternatively, private daemons can be created for specific tracing sessions. The named socket is then created in a custom location and the tracing library is told its location via the `UST_DAEMON_SOCKET` environment variable.

2.3. Trace Control

Trace control is done with a tool called `ustctl`. In operates on a list of PIDs, allowing to start and stop the tracing, list the markers and enable or disable them. Markers can be enabled or disabled dynamically, as the trace is being recorded.

The communication with the traced process is done via a named socket: each traced process creates a named socket in a standard directory. The name given to the socket file is the PID of the process. In order to disturb the traced process as little as possible, these communications are handled by a special thread in the traced process. This thread is created only when needed.

Tracing on PID 11036 is started and stopped with the commands:

```
$ ustctl --start-trace 11036
```

```
$ ustctl --stop-trace 11036
```

An example of marker listing is shown in Figure 5. Additionally can be enabled or disabled with the following commands:

```
$ ustctl --enable-marker ust/request_start
      11036
```

```
$ ustctl --disable-marker ust/request_start
      11036
```

Sometimes, tracing is needed from the start of the program's main() function. In these cases, it is not acceptable to have to run `ustctl` to start the tracing, because some events will likely have already been lost. For this reason, it is possible to use environment variables to request that the tracing be already started when the main() function starts.

The `usttrace` script automatically sets up the appropriate environment variables, starts a private daemon for the tracing session and starts the command specified on the command line. The trace is saved in `/.usttrace/`.

3. Userspace Challenges

The targeted licence for the LTTng Userspace Tracer is LGPL. This is in order to allow the distribution of the library linked with non-GPL applications, including proprietary ones. Code ported from the Linux kernel and LTTng has been rewritten when authorizations could not be obtained to relicence the code. At the time of writing this paper, from about 30 files that initially had licensing issues, only one was remaining [2]. A resolution is expected by the end of 2009, otherwise the problematic parts of the file will be rewritten.

Another challenge of userspace tracing is to use an appropriate buffering scheme. Currently, per-process buffers are used. Therefore, there is one buffer per channel, per process. This approach was the most straightforward when

[2]The file is relay.c, that contains the implementation of the buffering mechanism. The licensing issues only apply to the API and not to the LTTng-specific code.

```
$ ustctl --list-markers 11036
{PID: 11036, channel/marker: metadata/core_marker_format, state: 1, fmt: "channel %s name %s
      format %s"}
{PID: 11036, channel/marker: metadata/core_marker_format, state: 1, fmt: "channel %s name %s
      format %s"}
{PID: 11036, channel/marker: metadata/core_marker_id, state: 1, fmt: "channel %s name %s
     event_id %hu int #1u%zu long #1u%zu pointer #1u%zu size_t #1u%zu alignment #1u%u"}
{PID: 11036, channel/marker: metadata/core_marker_id, state: 1, fmt: "channel %s name %s
     event_id %hu int #1u%zu long #1u%zu pointer #1u%zu size_t #1u%zu alignment #1u%u"}
{PID: 11036, channel/marker: metadata/core_marker_format, state: 1, fmt: "channel %s name %s
      format %s"}
{PID: 11036, channel/marker: metadata/core_marker_id, state: 1, fmt: "channel %s name %s
     event_id %hu int #1u%zu long #1u%zu pointer #1u%zu size_t #1u%zu alignment #1u%u"}
{PID: 11036, channel/marker: ust/request_start, state: 0, fmt: "number1 %d number2 %d"}
{PID: 11036, channel/marker: ust/request_end, state: 0, fmt: "str %s"}
{PID: 11036, channel/marker: ust/tptest, state: 0, fmt: "myarg %d"}
```

Figure 5: Example of marker listing. One line represents one marker. The PID is followed by the channel and marker name seperated by a slash. The state can be 0 (marker disabled) or 1 (marker enabled). The format string (`fmt`), describes the arguments of the marker. Each argument name is followed by its printf()-like type specifier. The hash notation in the format specifier is LTTng specific, and is used to specify the size of the integer arguments in the trace file.

porting from the kernel. However, it is not the most efficient. Indeed, the memory of a process may be shared among several threads. If many threads running on different cores are producing events in the same buffers, this induces cacheline bouncing. Modifications to the library to reduce this cacheline bouncing are in progress.

4. Summary and Future Work

With the LTTng Userspace Tracer there is a lightweight userspace tracing solution available that can be used standalone but also integrates seamless into the existing LTTng kernel tracer and its analysis applications available today. The performance overhead introduced by the instrumentation is negligible so that it even can be used in production builds. Nevertheless, as the Userspace Tracer becomes more mature, it is likely that new optimizations will result in an even lower tracing overhead.

The current per-process buffers were a simple first step for a port. However, this approach has an important limitation. It induces cacheline bouncing on multi-threaded applications. Using per-thread buffers would fix this problem.

In the kernel, the most optimized variant of the markers uses immediate values, a technique that modifies an instruction at the instrumentation point site when enabling or disabling markers. This code modification consists in changing the immediate value in a load immediate instruction. This instruction is immediately followed by a test of the register in which the value was loaded. Depending on the result of the test, the event is recorded or not. Work is also ongoing to add support for static jump patching in gcc. Using a similar code modification scheme will therefore permit to modify a jump instruction, therefore saving branch prediction buffers, instruction cache and a few cycles when executed. Although this approach is faster than the current test of a global variable, is much more architecture-dependant.

Work is currently in progress to enhance the daemon so it can send traces over a network. This is particularly useful on special purpose systems with little or no disk space available.

References

[1] LTTng Project, `http://www.lttng.org`

[2] Dynamic Probes Project, `http://dprobes.sourceforge.net`

[3] SystemTAP Manual Page, *STAPPROBES(5)*

[4] Paul McKenney, *What is RCU, Fundamentally?*, `http://lwn.net/Articles/262464/`

[5] Mathieu Desnoyers and Paul E. McKenney, *Userspace Read-Copy-Update Library*, `http://ltt.polymtl.ca/cgi-bin/gitweb.cgi?p=userspace-rcu.git`

[6] Pierre-Marc Fournier and Jan Blunck, *Kernel Compatibility Library*, `http://git.dorsal.polymtl.ca/?p=libkcompat.git`

[7] Mathieu Desnoyers, *Userspace RCU Library : What Linear Multiprocessor Scalability Means for Your Application*, Linux Plumbers Conference 2009, `http://linuxplumbersconf.org/2009/slides/Mathieu-Desnoyers-talk-lpc2009.pdf`