LiTL

Lightweight Trace Library

User Manual

Roman Iakymchuk and François Trahay

August 23, 2013

Contents

Chapter 1

License of LiTL

LiTL is developed and distributed under the GNU General Public License.

This library is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

LiTL is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with LiTL; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Chapter 2

Overview of LiTL

LiTL [?] is a lightweight binary trace library that aims at providing performance analysis tools with a scalable event recording mechanism that utilizes minimum resources of the CPU and memory. In order to efficiently analyze modern HPC applications that combine OpenMP (or Pthreads) threads and MPI processes, we design and implement various mechanisms to ensure the scalability of LiTL for a large number of both threads and processes.

LiTL is designed in order to resolve the following performance tracing issues:

- Scalability and the number of threads;
- Scalability and the number of recorded traces;
- Optimization in the storage capacity usage.

As a result, LiTL provides similar functionality to standard event recording libraries and records events only from user-space. LiTL minimizes the usage of the CPU time and memory space in order to avoid disturbing the application that is being analyzed. Also, LiTL is fully thread-safe that allows to record events from multi-threaded applications. Finally, LiTL is a generic library that can be used in conjunction with many performance analysis tools and frameworks.

Chapter 3

Installation

3.1 Requirements

In order to use LiTL, the following software is required:

1. autoconf of version 2.63;

3.2 Getting LiTL

Current development version of LiTL is available via Git

```
git clone git+ssh://fusionforge.int-evry.fr//var/lib/
gforge/chroot/scmrepos/git/litl/litl.git
```

After getting the latest development version from Git, the following command should be run

```
./bootstrap
```

And, only afterwards the tool can be built.

3.3 Building EZTrace

At first, to configure LiTL the following configure script should be invoked

```
./configure --prefix=<LITL_INSTALL_DIR>
```

The configuration script contains many different options that can be set. However, during the first try we recommend to use the default settings.

Once LiTL is configured, the next two commands should be executed to complete the building

```
make
make install
```

In order to check whether LiTL was installed correctly, a set of tests can be run as

```
make check
```

Chapter 4

How to Use LiTL?

4.1 Reading Events

After the application was traced and events were recorded into binary trace files, those traces can be analyzed using `litl_read` as

```
litl_read -f trace.file
```

This utility shows the recorded events in the following format:

- Time since last probe record on the same CPU;
- ID of the current thread on this CPU;
- Event type;
- Code of the probe;
- Number of parameters of the probe;
- List of parameters of the probe, if any.

4.2 Merging Traces

Once the traces were recorded, they can be merged into an archive of traces for further processing by the following command

```
litl_read -o archive.trace trace.0 trace.1 ...  trace.n
```

4.3 Splitting Traces

In case of a need for a detailed analysis of a particular trace files, an archive of traces can be split back into separate traces by

```
litl_read -f archive.trace -d output.dir
```

4.4 Environment Variables

For a more flexible and comfortable usage of LiTL, we provide the following environment variables:

- `LITL_BUFFER_FLUSH` specifies the behavior of LiTL when the event buffer is full. If it is set to one, which is a default option, the buffer is flushed. This permits to record traces that are larger than the buffer size. Otherwise, if it is set to zero, any additional event will not be recorded. The trace is, thus, truncated and there is no impact on the application performance;
- `LITL_THREAD_SAFETY` specifies the behavior of LiTL while tracing multi-threaded applications. If it is set to one, which is a default value, the thread safety is enabled. Otherwise, when it is zero, the event recording is not thread safe;
- `LITL_TIMING_METHOD` specifies the timing method that will be used during the recording phase. The LiTL timing methods can be divided into two groups: those that measure time in clock ticks and those that rely on the `clock_gettime()` function. The first group has only one method:
  - `ticks` that uses the CPU specific register, e.g. rdtsc on X86 and X86_64 architectures.

  The second group comprises of the other five different methods:
  - `monotonic` that corresponds to `CLOCK_MONOTONIC`;
  - `monotonic_raw` – `CLOCK_MONOTONIC_RAW`;
  - `realtime` – `CLOCK_REALTIME`;
  - `thread_cputime` – `CLOCK_THREAD_CPUTIME_ID`;
  - `process_cputime` – `CLOCK_PROCESS_CPUTIME_ID`.

  User can also define its own timing method and set the environment variable accordingly.

Chapter 5

LiTL in Details

5.1 Event Types and The Storage Usage

Each event in the LiTL library consists of two parts: the event core (the event code, the time when the event occurred, the thread identifier, and the number of parameters) and event parameters. The number of event parameters recorded by LiTL varies from zero to ten.

The parameters passed to each event have different data type. In order to handle the variety of possible cases, event's parameters in LiTL can be represented by the largest data type, which is `uint64_t` on x86_64 architectures. Hence, any parameter – no matter whether it is a `char`, an `int` or a `long int` – can be recorded without being truncated. However, the reserved slot for each parameter is often bigger than its actual size. Thus, this leads to the non-optimal usage of resources. Our goal is to keep trace files as small as possible without losing any of the recorded data. Therefore, we propose to use the compacted event storage that aims at utilizing every byte from the allocated space.

In our approach, we introduce three different types of events: regular, raw, and packed. The regular event is without any major optimization being involved. The raw event stores parameters in the string format. Its purpose is to gather either the regular parameters in a string format or the information about the abnormal behavior of applications like thrown exceptions. The packed event represents the optimized versions of storing events, where each parameter can be saved as a group of bytes. Accordingly, by using the event type packed for recording and storing events, we theoretically are capable to save up to 65 % of the disk space compare to the regular LiTL.

?? shows, on an example of three regular events with different number of parameters, the occupied space of events within the trace file recorded by EZTrace with LiTL. We symbolically partitioned the trace file into bytes and also chunks of bytes, which store event's components. The space occupied by each event is highlighted with parentheses.

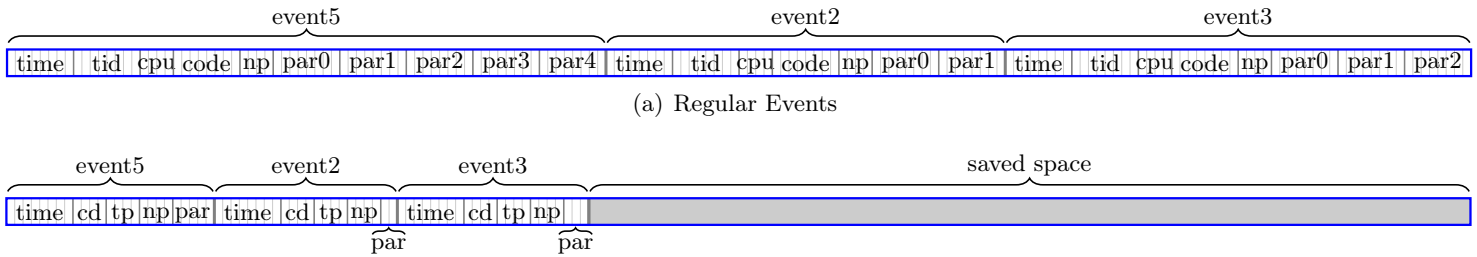

Figure 5.1: Storage of different kinds of events in the trace file. In the figure, *time* is the time when the event occurred; *cd* means the event code; *tp* is the event type; *np* stands for the number of event's parameters; *par* – an array of parameters.

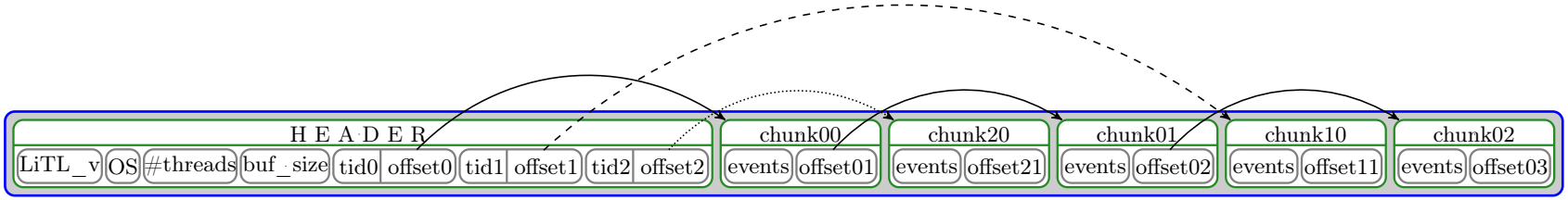

Figure 5.2: Storage of events recorded by LiTL on multi-threaded applications. In the figure, *LiTL_v* contains information about LiTL; *OS* – about OS and architecture; *#threads* stands for the number of threads; *buf_size* – the buffer size.

?? shows the storage of the recorded packed events in the trace file while using EZTrace with LiTL. We consider one particular scenario when each event's parameter can be represented by `uint8_t`; this requires only one byte for the storage. To store larger event's parameters we use arrays of `uint8_t`. This scenario corresponds to the optimal performance in terms of the memory and disk space usage. Under this approach, not only the size of the core event's components is shrunk, but also the size of event's parameters is reduced significantly. The gained performance, e.i. the reduced space, can be characterized by the gray area that corresponds to the difference in storage between the regular and packed events. The size of three packed events is smaller than the size of one regular event with five parameters. This figure confirms our assumption regarding the possibility of reducing the size of both the recorded events and trace files.

5.2 Scalability vs. the Number of Threads

The advent of multi-core processor have led to the increase in the number of processing units per machine. It becomes usual to equip a typical high performance computing platform with 8, 16, or even more cores per node. In order to exploit efficiently such facilities, developers can use hybrid programming models that mix OpenMP (or Pthreads) threads and MPI processes within one application. Hence, the number of threads per node, which executes the same application, can be quite large – 8, 16, or even more threads. The number of threads per node is the scalability issue for the conventional binary tracing libraries such as FxT [**?**], because in its implementation all threads within one process record events into a single buffer, see **??**. This recording mechanism causes a *contention* problem – when multiple threads record events simultaneously, the pointer to the next available slot in the buffer is changed concurrently. The modifications of the pointer can be done atomically in order to preserve the data consistency. However, the atomic operation does not scale quite well when it is performed by a large number of threads at the same time. Thus, analyzing OpenMP applications that run lots of threads using such tracing libraries may result in the high overhead.

5.2.1 Recording Events

While designing LiTL, we aim at resolving the above-mentioned limitation of FxT. Thus, we propose to record events into separate buffers, meaning to have one buffer per thread instead of one buffer per process. This approach is illustrated on **??**.

To keep multiple buffers in order within the trace file, we add a header into the trace file with the information regarding the number of threads and pairs *<tid, offset>*; *tid* stands for the thread identifier; *offset* corresponds to the position of the first chunk of events for a given thread within the trace

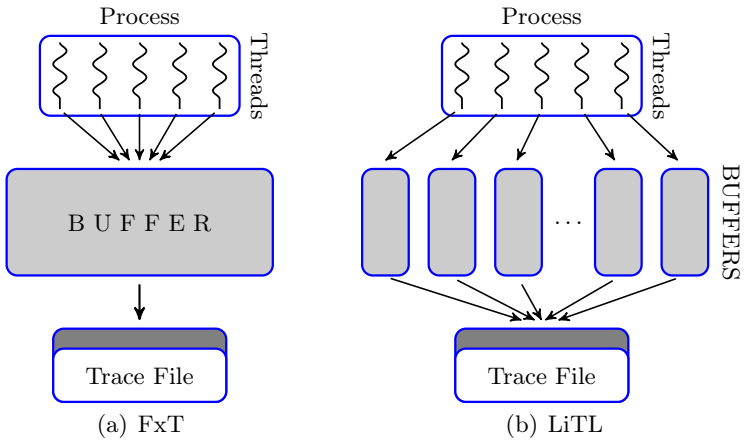

Figure 5.3: Event recording mechanism on multi-threaded applications.

starting from its beginning. The last event of each chunk contains either an *offset* to the next chunk of events or a symbol specifying the end of recording for a given thread. While flushing the current buffer to the trace file, the following two actions are performed:

1. Setting the offset of the current chunk to specify the end of the recording;

2. Update the offset from the previous chunk to point to the current one.

?? demonstrates the storage mechanism on an example of three threads, including the positioning of chunks of events as well as the way of linking those chunks into one chain of the corresponding thread using offsets.

During the application execution, it may occur that some threads start recording events later than others. This scenario requires appropriate modifications and adjustments to the above approach. According to the previous approach, the header is the first block of data that is added to the trace file; it is written before flushing the first chunk of events. Thus, the header contains the information only regarding the started threads. In order to add pairs *<tid, offset>* of the late threads, we reserve a space for 64 pairs (chunk of pairs) between chunks of events within the trace file. So, when one among those late threads wants to flush its buffer to the trace file, we add its pair *<tid, offset>* directly to the next free spot in the chunk of pairs. The chunks of pairs are binded with offset in the same way as chunks of events. Therefore, EZTrace does not have limitations on the number of threads per process and also processes.

5.2.2 Post-Mortem Analysis

We develop the functionality for analyzing the generated traces by capturing the procedure of the event recording mechanism. At first, LiTL reads the trace header with the information regarding the buffer size, threads (the number of threads, tids, and offsets), and also pairs *<tid, offset>* that correspond to the late threads. Using this preliminary information, LiTL allocates memory buffers for reading; the number of buffers equals the number of threads used during the recording phase, meaning one buffer per thread. Then, LiTL loads chunks of events from the trace file into these buffers using pairs *<tid, offset>*. After processing the first chunks of events, LiTL loads the buffers with the next ones using the information concerning their positions in the trace, which is given by the offsets. This procedure is recursive and stops when the symbol specifying the end of recording is reached.

5.3 Scalability vs. the Number of Traces

Usually binary tracing libraries generate one trace file per process. This means that for parallel applications with hundreds of MPI processes the equal amount of trace files is created. This is one side of the problem. The other side appears while analyzing the applications execution due to the limitation on the number of trace files that can be opened and processed at the same time. Therefore, often those tracing libraries do not perform well and even crashes when the number of traces exceeds the Linux OS limit on the number of simultaneously opened files.

In order to overcome the opened files limitation imposed by the Linux OS, one may increase the limit to the maximum possible value. However, this would temporarily solve the problem. Instead, we propose to create archives of traces during the post-mortem phase. More precisely, we suggest to merge multiple traces into a trace archive using the `litl_merge` utility from LiTL. **??** illustrates the structure of the new combined trace created by `litl_merge`. The archives of traces preserve all information concerning each trace: headers, pairs *<tid, offset>*, and positioning of events chunks. They also contain new global headers that store the information regarding the amount of trace files in the archive and triples *<fid, size, offset>*; *fid* stands for a file identifier; *size* is a size of a particular trace file; *offset* holds the position of a trace file within the archive of traces. Therefore, archives of traces not only solve the performance analysis problem, but also make the further analysis of the applications performance more convenient.

One more useful feature provided by LiTL, which is the opposite of `litl_merge`, is a possibility to extract trace files from archives with the `litl_split` utility. This utility can be applied when there is a need to analyze a particular trace or a set of traces among the merged ones.

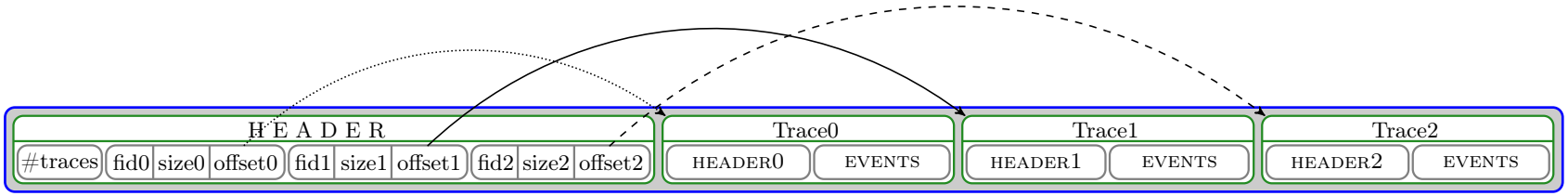

Figure 5.4: The structure of an archive composed of multiple trace files. In the figure, *fid* stands for the trace file name; *size* is the size of a merged trace file.

Chapter 6

LiTL in FxT applications

In this chapter, we present an approach of integrating LiTL (as a possible replacement of or enable its usage in parallel with FxT) into applications that already reply on FxT. To simplify the process of integrating LiTL into such applications, we matched the functionality of FxT to the corresponding functionality from LiTL into `fxt.h` and `fut.h` headers. So, developers of those applications can use LiTL in conjunction with these two header files, which that use anyway, and therefore keeping without changes the code and the major part of the FxT APIs.

Even though LiTL and FxT target the same issue of gathering the information of the application execution, they have differences in the organization of the event recording as well as the event reading processes. In order to deal with those differences, we suggest to modify FxT-related applications by following our suggestions.

6.1 Recording Events

The main difference between two trace libraries is in the organization of the initialization phase of the event recording process. So, in FxT it is implemented as

```
1  // IMPORTANT! setting file name is before
2  fut_set_filename(PROF_FILE_USER);
3
4  if (allow_flush && ...) {
5      enable_fut_flush();
6  }
7
8  fut_enable_tid_logging();
9
10 if (fut_setup(PROF_BUFFER_SIZE, FUT_KEYMASKALL, threadid) < 0) {
11     perror("fut_setup");
12 }
```

While in LiTL the procedure is the following

```
litl_trace = litl_init_trace(PROF_BUFFER_SIZE);
// the recording should be paused, because some further functions,
// e.g. *_set_filename() can be intercepted
litl_pause_recording(&litl_trace);

// yes, the API is the same thanks to fxt.h and fut.h
fut_enable_tid_logging();

if (allow_flush && ...) {
    enable_fut_flush();
}

// IMPORTANT! setting file name is after
fut_set_filename(PROF_FILE_USER);

// Do not forget to resume recording
litl_resume_recording(&litl_trace);
```

For the successful and easy porting of LiTL into your applications the above-mentioned suggestions needs to be incorporated.

6.2 Reading events

Chapter 7

Troubleshooting

If you encounter a bug or want some explanation about LiTL, please contact and ask our development team on the development mailing list

- `litl-devel@fusionforge.int-evry.fr`.