EZTrace

A Generic Framework for Instrumenting Applications

User Manual

October 24, 2012

Contents

Chapter 1

License of EZTrace

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

This program 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.

EZTrace 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 EZTrace; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Chapter 2

What is EZTrace?

EZTrace [2, 8, 4] is a tool that aims at generating automatically execution traces from High-Performance Computing (HPC) programs. It generates execution trace files that can be interpreted by visualization tools such as ViTE [5].

EZTrace has been designed to provide a simple way to trace parallel applications. This framework relies on plugins in order to offer a generic way to analyze programs; depending on the application to analyze or on the point to focus on, several modules can be loaded. EZTrace provides predefined plugins that give the ability to the user to analyze applications that use MPI libraries, OpenMP, or Pthreads. However, user-defined plugins can also be loaded in order to analyze application functions or custom libraries.

Chapter 3

Installing EZTrace

3.1 Requirements

In order to run EZTrace, the following software needs to be installed:

1. autoconf 2.63;
2. [Optional] Any MPI implementation. Make sure your MPI installation provides dynamic libraries.

We use custom versions of the following software:

1. FxT 0.2.10 (`http://download.savannah.gnu.org/releases/fkt/`);
2. GTG 0.2 (`http://gforge.inria.fr/projects/gtg/`).

Those are already included in EZTrace, but you can also provide your own versions.

3.2 Getting EZTrace

1. You can get the latest stable release from the EZTrace website, `http://eztrace.gforge.inria.fr/`;
2. Current development version is available via SVN
   `svn checkout svn+ssh://scm.gforge.inria.fr/svn/eztrace`.

After getting the latest development version (from SVN), you need to run
```
./bootstrap
```
and only then build the tool.

3.3 Building EZTrace

At first, you need to configure EZTrace by invoking the configure script:

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

Options to configure. You can pass several options to the configure script for specifying where it should find the required libraries:

- `--with-fxt=$FXT_ROOT` – specify where FxT is installed;
- `--with-gtg=$GTG_ROOT` – specify where GTG is installed;
- `--with-mpi=$MPI_ROOT` – specify where MPI is installed. The `mpi.h` file must be located in the `$MPI_ROOT/include/directory`;
- `--with-mpi-include=<PATH_TO_MPI.H>` – specify the directory that contains the `mpi.h` file.

Once EZTrace is configured, just run:

```
make
make install
```

Chapter 4

EZTrace in Details

EZTrace uses a two-phases mechanism for analyzing performance. During the first phase that occurs while the application is executed, functions are intercepted and events are recorded. After the execution of the application, the post-mortem analysis phase is in charge of interpreting the recorded events. This two-phase mechanism permits the library to separate the recording of a function call from its interpretation. It thus allows the user to interpret a function call event in different ways depending on the point he/she wants to focus on. It also reduces the overhead of profiling a program; during the execution of the application, the analysis tool should avoid performing time-consuming tasks such as computing statistics or interpreting function calls.

During the execution of the application, EZTrace intercepts calls to the functions specified by plugins and records events for each of them. Depending on the type of functions, EZTrace uses two different mechanisms for interception. The functions defined in shared libraries can be overridden using `LD_PRELOAD`: When the EZTrace library is loaded, it retrieves the addresses of the functions to instrument. When the application calls one of these functions, the version implemented in EZTrace is called. This function records events and calls the actual function. The `LD_PRELOAD` mechanism cannot be used for functions defined in the application since there is no symbol resolution. In that case, EZTrace uses the DynInst [3] tool for instrumenting the program on the fly. Using DynInst, EZTrace modifies the program to record events at the beginning and/or at the end of each function to instrument.

EZTrace is structurally divided into two parts: the EZTrace core and the EZTrace modules.

4.1 EZTrace Core

The EZTrace core is composed of several libraries such as the FxT [6] and the Generic Trace Generator (GTG) [4, 7] libraries.

4.1.1 FxT

For recording events, EZTrace relies on the FxT library. Each process being instrumented by EZTrace generates a trace file using FxT. In order to keep the trace size as compact as possible, FxT records events in a binary format that contains only the minimum amount of information: a timestamp, an event code and optional parameters.

4.1.2 GTG

During the post-mortem analysis phase, EZTrace browses the recorded events and interprets them. It can then generate statistics – such as the length of messages, the duration of critical sections, etc.– or create a trace file for visualizing the application behavior. For generating trace files, EZTrace relies on the GTG library. GTG provides an abstract interface for recording traces. This permits the application to use a single interface for creating traces in multiple formats. Thus, an application can generate PAJÉ traces or OTF files without any modification.

Althought PAJÉ and OTF are both traces format, they have some differences. Thus, adding a meaning to a raw FxT event is the critical part and the event must be interpreted in a way that is conformed to the output format choosen by the user. Otherwise, the traces will not represent the data they should.

4.2 EZTrace Modules

Since EZTrace uses the two-phases mechanism, plugins are organized in two parts, accordingly: the description of the functions to instrument, and the interpretation of each function call. During the execution of the application, the first part of the plugin is in charge of recording calls to a set of functions. The second part of the plugin is in charge of adding semantic to the trace. EZTrace provides plugins for major parallel programming libraries (MPI, OpenMP, PThread, etc) but also allows user-defined plugins designed for custom libraries or applications. For exam- ple, the PLASMA linear algebra library [1] is shipped with an EZTRACE plugin.

Chapter 5

How to Use EZTrace

5.1 Selecting Functions to Instrument

First, you should select functions in your application that you want to instrument. For this, you can set the `EZTRACE_TRACE` environment variable to the list of plugins that should be used. For instance, set

```
export EZTRACE_TRACE="pthread mpi"
```

if you want to instrument the PThread and MPI functions.

You can get the list of available plugins using the `eztrace_avail` command:

```
eztrace_avail
  3  stdio    Module for stdio functions (read, write,
              select, poll, etc.)
  2  pthread  Module for PThread synchronization functions
              (mutex, semaphore, spinlock, etc.)
  6  papi     Module for PAPI Performance counters
  1  omp      Module for OpenMP parallel regions
  4  mpi      Module for MPI functions
  5  memory   Module for memory functions (malloc, free,
              etc.)
```

You can get the list of selected modules with the `eztrace_loaded` command:

```
export EZTRACE_TRACE="pthread mpi"
eztrace_loaded
  2  pthread  Module for PThread synchronization functions
              (mutex, semaphore, spinlock, etc.)
  4  mpi      Module for MPI functions
```

5.2 Generating Execution Traces

Once the list of plugins is selected, you can run your application with EZ-Trace. For example:

```
eztrace ./my_program my_arg1 my_arg2
```

This command line executes your program and generates a trace file in the `/tmp` directory (usually the file is named as `/tmp/<username>_eztrace_log_rank_<rank>`).

5.3 Using EZTrace for MPI Applications

EZTrace needs to instrument each MPI process. Thus, you can run the following command:

```
mpirun -np nproc eztrace ./my_program,
```

where `npoc` is the number of MPI processes. When your application ends. Each process writes a file named

`/tmp/<username>_eztrace_log_rank_<rank>`.

5.4 Changing the Output Directory

By default, each process saves its trace in the local `/tmp` directory. You can change this by setting the `EZTRACE_TRACE_DIR` environment variable.

5.5 Merging Execution Traces

Once the execution traces are recorded, you can merge and convert them into a file format that can be read by your visualization software:

```
eztrace_convert -o my_paje.trace /tmp/<username>_eztrace_
  log_rank_0 /tmp/<username>_eztrace_log_rank_1
```

This converts the trace files into the Paje format. If GTG is installed with OTF support (this is enabled by default), you can choose to convert into the OTF file format with the `-t OTF` option:

```
eztrace_convert -t OTF /tmp/<username>_eztrace_log_rank_0
  /tmp/<username>_eztrace_log_rank_1
```

5.6 Filtering Events

You can select the plugins to use for the conversion phase by using the `EZTRACE_TRACE` environment variable. For instance, if your traces contains MPI and other events, then by setting `EZTRACE_TRACE` to `mpi` and calling `eztrace_convert` you will receive an output trace that contains only MPI events.

5.7 Computing Statistics

Instead of creating a merged trace file, you can tell EZTrace to compute statistics on the recorded traces:

```
eztrace_stats /tmp/<username>_eztrace_log_rank_0
  /tmp/<username>_eztrace_log_rank_1
    [...]
    PThread:
    -------
    6 locks acquired

    MPI:
    ---
    27 messages sent
    MPI_RECV: 10 calls
    MPI_BARRIER: 11 calls
    163 events handled
```

5.8 Defining Custom Plugins

Since EZTrace works with plugins, you can create one and instrument the functions that you want. An example of a custom plugin is available in the `example` directory.

Once your plugin is created, you should tell EZTrace where to find it. For this, just set the `EZTRACE_LIBRARY_PATH` variable to the appropriate directory(-ies):

```
export EZTRACE_LIBRARY_PATH=plugin1:plugin2
eztrace_avail
  3   stdio    Module for stdio functions (read, write,
               select, poll, etc.)
  2   pthread  Module for PThread synchronization func-
               tions (mutex, semaphore, spinlock, etc.)
  6   papi     Module for PAPI Performance counters
  1   omp      Module for OpenMP parallel regions
  4   mpi      Module for MPI functions
  5   memory   Module for memory functions (malloc, free,
               etc.)
  99  plugin1  Example module for libplugin1
  98  plugin2  Example module for the plugin2 library
```

5.9 Generating Custom Plugins

You can generate one plugin and instrument the functions you want. In order to generate your plugin, you need to create a file containing:

1. The name of the library you want to trace (`libNAME.so`);
2. [Optional] A brief description of the library;
3. An ID to identify the module. 0 is reserved for EZTrace internal use. Thus, you can use any between 10 and ff;
4. The prototype of the functions you want to instrument.

As a result, your file should look as follow

Listing 5.1: `example.tpl`

```
BEGIN_MODULE
NAME example_lib
DESC "module for the example library"
ID 99
int example_do_event(int n)
double example_function1(double* array, int array_size)
END_MODULE
```

Now use `eztrace_create_plugin` to generate the plugin source code:

```
eztrace_create_plugin example.tpl
  New Module
  Module name:  'example_lib'
  Module description:  '"module for the example library"'
  Module id:  '99'
       emulate record_state for 'example_do_event'
  Function 'example_do_event' done
       emulate record_state for 'example_function1'
  Function 'example_function1' done
  End of Module example_lib
```

The source code is generated in the output directory. Just type:

```
make
```

Then, set the `EZTRACE_LIBRARY_PATH` to the appropriate directory. Now, your custom plugin is ready to be used.

You can also determine (in the `example.tpl` file) the way a function is depicted in the output trace. For instance,

```
int submit_job(int* array, int array_size)
BEGIN
 ADD_VAR("job counter", 1)
END
```

specifies that when the `submit_job` function is called, the output trace should increment the "job counter" variable. You can now track the value of a variable.

The `test/module_generator` directory contains several scripts that demonstrate the various commands available.

Chapter 6

Environment Variables

Here is a list of the environment variables that can be used for tuning EZTrace.

- General-purpose variables:
  - `EZTRACE_TRACE_DIR` specifies the directory in which trace files are stored (by default it is `/tmp`);
  - `EZTRACE_LIBRARY_PATH` specifies the directories in which EZTrace can find EZTrace modules (by default, it is none);
  - `EZTRACE_TRACE` specifies the list of EZTrace modules to load (by default, it is the list of all available modules);
  - `EZTRACE_BUFFER_SIZE` specifies the size of the buffer in which EZTrace stores events (by default, the size is 32 MB);
  - `EZTRACE_FLUSH` specifies the behavior of EZTrace when the event buffer is full. If it is set to one, the buffer is flushed. This permits to record traces that are larger than `EZTRACE_BUFFER_SIZE`, but this may impact the application performance. Otherwise, if it is set to zero, which is a default option, any additional event will be recorded. The trace is, thus, truncated and there is no impact on performance.
- Error-handling variables:
  - `EZTRACE_NO_SIGNAL_HANDLER` disables EZTrace signal handling (by default, it is zero).
- Hardware counters-related variables:
  - `EZTRACE_PAPI_COUNTERS` selects hardware events to trace using the PAPI library, e.g. export `EZTRACE_PAPI_COUNTERS="PAPI_L3_TCM PAPI_FP_INS"`. Please note that the list of supported events

as well as the number of events, which can be traced simultaneously, vary depending on the processor type. This information can be retrieved using `papi_avail` and the processor documentation.

- MPI-related variables:
  - `EZTRACE_MPI_DUMP_MESSAGES` tells EZTrace to dump the list of messages into a file. You can then compute your own statistics on MPI messages.

Chapter 7

Known Limitations

- If EZTrace is compiled with a particular MPI implementation such as OpenMPI, it will not work if you run your application with another, e.g. MPICH2. So make sure your application uses the same MPI implementation as EZTrace.

Chapter 8

Frequently Asked Questions

Q. When I run my MPI application with EZTrace, all the processes generate the `/tmp/<username>_eztrace_log_rank_1` file. What is going wrong?

A. This happens when EZTrace fails to intercept calls to `MPI_Init` or `MPI_Init_thread`. There can be several reasons for that:

- The EZTrace MPI module was not compiled. For intercepting calls to MPI functions, you need the MPI module in your installation (look for the `EZTRACE_ROOT/lib/libeztrace-autostart-mpi.so` file). If you do not see that file, it means that something went wrong during the configuration of EZTrace, so check for errors or warnings in the `config.log` file.
- You specified the list of modules to use and the MPI module was not there. Check your `EZTRACE_TRACE` variable or use `eztrace_loaded`.

If you still experience problems, please contact the EZTrace development team and we will fix your problem.

Q. What if I do not want to trace the whole application, but only a part of it?

A. Then, you can call `eztrace_start()` and `eztrace_stop()` specify in the code which part to trace. You will need to add `#include <eztrace.h>` and to link with `libeztrace`. Afterwards, you can run your application as usual, i.e. `./my_program my_arg1`.

Q. I need to trace my program while using GDB, how can I do that?

A. Just add the `-d` option to EZTrace to enable GDB:

```
eztrace -d ./my_program my_arg1 my_arg2
```

Please note that this should be applied only when a bug occurs while using EZTrace.

Q. I want my trace to be saved in a specific directory, how can I do that?

A. Please take a look in Section 5.4.

Q. What if I do not care about OpenMP and I only want to see MPI communication?

A. You can set `EZTRACE_TRACE` to the list of modules you want to activate. By default, all the available modules are enabled, but you can tell EZTrace to trace only MPI, OpenMP, or both MPI and OpenMPI functions:

```
export EZTRACE_TRACE=mpi
export EZTRACE_TRACE=omp
export EZTRACE_TRACE="omp mpi"
```

Q. Can EZTrace generate an OTF trace file so that I can visualize it with Vampir?

A. Yes, since EZTrace relies on GTG for writing output traces, it can generate OTF trace files. When converting the trace with `eztrace_convert`, just add the `-t OTF` option:

```
eztrace_convert -t OTF /tmp/<username>_eztrace_log_
  rank_0 /tmp/<username>_eztrace_log_rank_1
```

Chapter 9

Troubleshooting

If you encounter a bug or want some explanation about EZTrace, please contact and ask our development team:

- On the development mailing list, `https://gforge.inria.fr/mail/?group_id=2774`;
- On our IRC channel:
  - Server: `chat.freenode.net`
  - Channel: `#eztrace`

Bibliography

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180, 2009.

[2] Charles Aulagnon, Damien Martin-Guillerez, François Rue, and François Trahay. Runtime function instrumentation with EZTrace. In *Proceedings of the PROPER – 5th Workshop on Productivity and Performance*, Rhodes, Greece, August 2012.

[3] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, November 2000.

[4] Kevin Coulomb, Augustin Degomme, Mathieu Faverge, and François Trahay. An open-source tool-chain for performance analysis. *Tools for High Performance Computing 2011*, pages 37–48, 2012.

[5] Kevin Coulomb, Mathieu Faverge, Johnny Jazeix, Olivier Lagrasse, Jule Marcoueille, Pascal Noisette, Arthur Redondy, and Cleḿent Vuchener. ViTE – Visual Trace Explorer. Available via the WWW. Cited 14 October 2012. `http://vite.gforge.inria.fr/index.php`.

[6] Vincent Danjean, Raymond Namyst, and Pierre-André Wacrenier. An efficient multi-level trace toolkit for multi-threaded applications. In *Proceedings of the 11th international Euro-Par conference on Parallel Processing*, Euro-Par'05, pages 166–175, Berlin, Heidelberg, 2005. Springer-Verlag.

[7] François Rue, François Trahay, Johnny Jazeiz, Kevin Coulomb, Mathieu Faverge, and Olivier Lagrasse. GTG – Generic Trace Generator. Available via the WWW. Cited 14 October 2012. `http://gtg.gforge.inria.fr/`.

[8] François Trahay, François Rue, Mathieu Faverge, Yutaka Ishikawa, Raymond Namyst, and Jack Dongarra. EZTrace: a generic framework for performance analysis. In *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Newport Beach, CA, USA, May 2011.