Université Grenoble - Alpes

Nano2017 Dema
SP 1—Interactive Debugging

Livrable D2 : Version 1.0 du debugger de performance

Kevin Pouget, Jean-François Méhaut
UGA-Lig/Inria Corse
January 30, 2017

Contents

Chapter 1

Introduction

In this document, we present the work done in the first semester of 2016 as part of the Nano2017 Dema project. While the first year of the project (deliverables D1 and D3) was focused on improving *functional* interactive debugging, this year aimed at introducing *performance* debugging capabilities into interactive debuggers.

As an introduction, we draw an overview of today's profiling tools and interactive debuggers' capabilities. Then we present in Chapter 2 our solution for performing interactive code profiling. This interactive profiling environment is the foundation of the interactive performance debugging methodology developed as part of the Deliverable D3.

In Chapter 3 we tackle some understanding questions related to the profiling measurements. The first part treats the problem of nesting and interleaving profiles, and the second part discusses the display and understanding of the different measurements.

1.1. Profiling Today

Code profiling is an essential part of performance debugging. It consists in measuring how the application, or some part of it, behaves during the execution. This behavior can be measured against different metrics that highlight some aspects or others of the execution characteristics. We can sort out the metrics into four broad categories: hardware, kernel, runtime and language level. We describe below these different categories and the kind of information they can provide.

These measurements provide different information to the developers, who will have to combine and confront them with their understanding of the code and hardware characteristics, to understand their meaning.

Once the measurements have been understood, the developers have to decide if the code can be optimized and sketch out and test new solutions.

Different types of profilings

Hardware counters These counters provide information related to the CPU execution. They are implemented at hardware level, and hence their availability and accuracy depend of the processor design. In x86 processors with `perf stat` or `papi`, we can find the following counters:

- instruction and cycles counters
- cache references and misses
- branch instructions and misses

Kernel counters These counters are related to the kernel. Their handling lays inside the its source code. In Linux kernel with `perf stat` or `papi`, we can find the following counters:

- CPU migrations and context switches
- CPU and task clocks
- major and minor page faults

Runtime counters These counters are tight to the runtime libraries, and hence they can vary a lot from one library to another. If we look at Aftermath tracing library [4] for OpenMP [1], we can find the following information:

- beginning and end of OpenMP regions
- thread executing the region
- indices of the OpenMP loop-chunk being executed

Language counter Finally, some counters can be implemented at language level, usually with the help of compiler instrumentation. For instance, GNU `gprof` is tight with the GNU compiler `gcc`. It provides different information related to the function calls:

- number of calls,
- time spent inside
- nesting (call tree)

Profiling tools lack interactivity

In the previous subsection we mentioned a few profiling tools and the type of information they can provide. However, we did not detail how they can be controlled, as it is somehow orthogonal. Let us detail the controls they provide:

Aftermath tracing library full execution tracing + post-mortem analysis [4]

gprof full execution tracing + post-mortem analysis [5]

papi counters requested by application during the execution [3]

perf stat full execution profiling or command-line attach/detach [2]

As we can see, these tools offer virtually no interactivity with the user. `Papi` can be controlled from the application, but that is mostly for auto-tuning; and `perf stat` can be attached and detached on user request, but this is very coarse-grained as the control originates from the command-line.

To contrast with the lack of interactivity of profiling tools, let us introduce the capabilities of interactive debugging tools such as GDB, the free debugger of the GNU project.

1.2. Interactive Debugging

The main usage of source-level interactive debuggers such as GDB is functional debugging. In this kind of debugging, the developers run the application step by step, and confront their mental representation of what they expect the code to do against what it actually does. Their mental representation comes from the specification of the application, and they query the actual execution state to the debugger, by printing the values of variables at different strategic points. A divergence between these two representations implies a bug in the code, or better stated, a defective statement.

Debugging is an application of the scientific method: observe bugs; think about the reason they can occur; formulate hypotheses about their causes; draw testable predictions (eg, if this hypothesis is true, then this variable should have this property); gather data to test the predictions; locate and fix the bug, or refine the hypothesis.

In this process, interactive debugging helps developers to test their predictions. Where `printf` statements can display a few values, defined at compile time, the debugger gives developers the ability to access most of the program internal state. Hence, multiple hypothesis and predictions can be tested at the same time, during in a single run.

In this report, we discuss how we extended interactive debuggers so that developers can also measure and test *performance* problems.

Chapter 2

Interactive Performance Profiling

The goal of interactive performance profiling is to give developers the ability to easily profile some specific regions of the code. In this chapter, we study how this capability can be implemented within a source-level debugger. We focus our study in GDB (the free debugger of the GNU project) and its Python interface, but the results apply to any debugger that can be extended with the full-flavored programming language.

In Section 2.1, we introduce the different levels of granularity at which the profiling can be done. Then in Section 2.2, we detail the different metrics that can be profiled with our tool.

2.1. Code Sections to Profile

Code performance profiling can be done in two different ways: either during the process normal execution; or outside of it, with parameters controlled by the debugger/profiler.

In this section, we first review the normal execution profiling abilities, then continue with the outside-of-execution profiling.

Normal Execution Profiling

Interactive debuggers have the ability to control the process execution and stop it upon various events. Among these events, we find the instruction breakpoints, the memory read/write watchpoints and the system-event catchpoints. Our tool relies on these events to start and stop the profiling regions:

- `function` — starts the profiling each time the given function is called, stops it when it returns (implemented with breakpoints and finish-breakpoints);
  - `(gdb) profile function compute_kernel`
- `region` — profiles a code region delimited by user breakpoints (or watchpoints/ catchpoints)
  - `(gdb) profile region main.c:110 kernel.c:25`

- `manual` — the profiling starts and stops on user request, from the command-line.
  - `(gdb) profile manual start`
  - `(gdb) profile manual stop`

Outside-Of-Execution Profiling

Outside-of-execution profiling relies on GDB and GCC just-in-time compilation library (`libgccjit`) cooperation. This cooperation gives GDB the ability to compile code on-the-fly and insert it temporarily in the application memory space.

We leveraged this feature to help developers to test and profile some code (mainly function calls) in different situations, without having to recompile and restart the application execution.

Historically, GDB already knows how to call functions within the debuggee process. Now it can profile it as well:

```
(gdb) call compute_kernel(arg1, arg2, arg3)

(gdb) profile interactive
  -repeat:10
  -code compute_kernel(arg1, arg2, arg3)
  -flags -O3
```

Our interactive profiling command accepts different parameters:

`-code` — the code to profile. It can use the local and global variables reachable from the current location;

`-file` — alternatively, the code can be passed through a file;

`flags` — flags to pass when compiling the code. This parameter can be set multiple times to profile different versions (eg, `-O0`, `-O3`, ...);

`-repeat` — how many time the code should be executed during the profiling;

`-app` — profiles the whole application instead of a code chunk.

It is important to note that the code dynamically inserted by this function is removed at the end of the profiling (this is the behavior of GDB `compile` command), but any side effect remains.

In single-threaded applications, this limitation could be relieved with GDB's checkpoint/restart capability (based on POSIX `fork` system call). However, as of today, this does not work in multithreaded environments.

Now that we have described how to specify the regions to profile, let us continue with the nature of the information that can be obtained.

2.2. Metrics to Measure

An interactive debugger cannot, by itself, profile the execution. It is not designed for that purpose. However, it can cooperate with an existing profiling tool, or any source of execution information. Let us go though the different information provider we implemented so far:

Linux `/proc` filesystem

The Linux kernel provides some process execution information though its `\proc\$PID` mount-point. Usually, the information indicates the current value of the counter, from the beginning of the execution. Hence, to compute a region profile, we just have to substrate the initial values from the final ones.

In these files, we can find:

/proc/$PID/status voluntary and non voluntary context switches

/proc/$PID/stat user and system execution times, major and minor page faults, stack size, heap size, . . .

/proc/$PID/io number of bytes read and written

Linux `perf stat`

Linux `perf stat` [2] is a tool exposing hardware and kernel counters. In its standard version, as described in the introduction, it does not allow a precise control of the profiling regions. However, controlled from the debugger, it can provide fine-grained measurements.

Our initial approach was to attach and detach `perf` each time a profiling region started or stopped. However, this solution appears to be too costly, because of `perf` process creation and initialization time.

We turned towards a better approach, that consists in attaching `perf` to the application from the beginning of the interactive profiling, and querying the values of the counter before and after the profiling region.

However, `perf` does not natively support this usage. To bypass this limitation, we built a shared-library that is preloaded into `perf` address space, and prints out the counters upon receiving a predefined signal (`SIGUSR2`). As we know that the debuggee process is stopped when the profiling regions start and stop, we can guarantee that `perf` will measure the right information.

Linux `perf` can measure a large number of counter. Its configuration is let open to the debugger user:

```
(gdb) set profile-perf-counters <counters>
```

Here are some examples of common event counters:

Hardware events branch-instructions/misses, cache-references/misses, cycles, instructions

Software events alignment-faults, context-switches, cpu/task-clock, major/minor-page-faults

Debugger breakpoints

Another information provider is the debugger itself. With the help of internal (invisible) breakpoints and watchpoints, we can count :

- how many times a function is executed
- how many times a memory location accessed within a profiling region.

Open to other tools

In the paragraphs above, we have listed different sources of profiling information. However, this list is not fixed and can be easily extended in the interactive profiler source code. The tool wrappers just need to be able to start and stop the profiling on demand, and provide the values of their different counters. See Appendix A.2 for the Python interface that should be implemented.

Among the possibilities of extension, we considered GCC `gprof` language-level execution counters. Their support would require an inspection of the `glibc` internal structures to extract the interesting information.

Another possibility is the information gathered by Aftermath tracing library. Instead of its default post-mortem treatment, the information could be measured and interpreted during the execution, with a cooperation between the interactive profiler and Aftermath visualization engine.

In this chapter, we introduced the main goals and capabilities of interactive performance profiling. We described the different granularities at which the profiling can be done, as well as the nature of the information that can be profiled.

In the following chapter, we discuss some concerns regarding the understanding of the profilings. We tackle the question of nested and interleaving regions, as well as the understanding of the measurement values.

Chapter 3

Understanding the Profilings

So far, we have demonstrated that profiling could be done interactively, under the control of a source-level interactive debugger such as GDB, the free debugger of the GNU project. We detailed the different scope at which this profiling could be done, as well as the information that could be profiled.

In this chapter, we tackle some questions related to the nesting of profiling zones, as well as their interleaving, in multithreaded applications. In the second part, we discuss how these measurements can be presented to the developers. In the Deliverable 4 report, we go further in the question of understanding the measurements by studying an actual performance bug.

3.1. Nested and Interleaving Profiling Regions

While profiling code at the function level, the problem of nesting quickly shows up: when the function being profiled calls itself, a new profiling region is supposed to start. But should the measurement of the inner function be included in the outer one? Likewise, when a function is called by several threads, there might be a period of interleaving in both of the threads.

The problem posed by these situations is in the understanding of the measurements. If the developer asks for the profiling of a function with an implicit recursion, and nesting is not taken into account, then the profilings will be skewed. Indeed, the measurements from the inner function will also be part of the outer one, with no explicit link between the two.

In the following, we discuss possible strategies to deal with such situations.

Nested Regions

We proposed and implemented three strategies to handle the nested profiling regions, as show in Figure 3.1.1:

First start last stop — only the outer region is recorded.

Per function — every region is profiled independently.

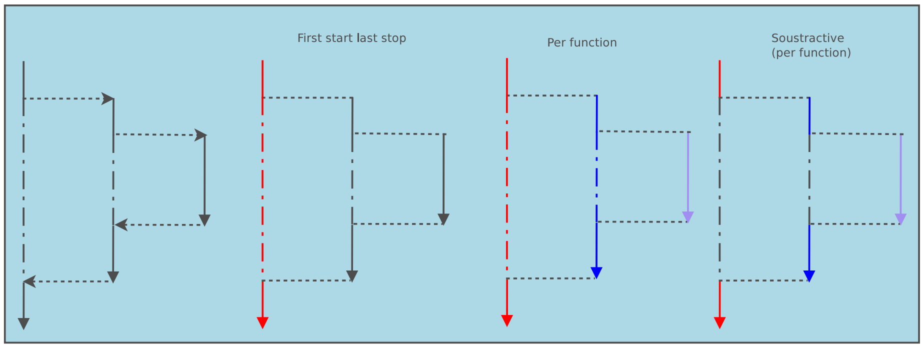

Figure 3.1.1: Different strategies to deal with nested profiling regions.

Per function, soustractive — every region is profiled, but we subtract the measures of the inner region from the outer one.

From our point of view, these three strategies have their own interest, depending of the context of the profiling. Only the developers can select the most accurate view for their debugging objectives. Hence, the three strategies are implemented, and all the relevant information is recorded. The choice of the visualization is deferred to the user interface. At the moment, it consists of command-line flags. We consider though that the **first-start-last-stop** strategy is the most intuitive, so it is the one displayed by default.

Interleaving Regions

A similar problem occurs with multithreaded applications, where a function can be called simultaneously from different threads. Figure 3.1.2 presents two solutions that can be adopted in this situation:

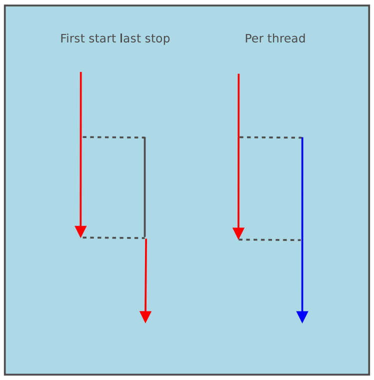

Figure 3.1.2: Different strategies to deal with interleaving profiling regions.

First start last stop — similarly to the nested regions, this strategy considers the earliest and latest boundaries of the profiling region

Per thread — this strategy splits the profiling region in two, one per thread. However, it is not guaranteed that all the profiling back-end tools support this single-thread profiling.

We did not study in detail the question of interleaving, as this deliverable was focused on sequential applications. In the Deliverable D4, we did not pursue this aspect either, as we focused on OpenMP profiling.

Another possibility is to force the sequentiality of the profiling regions: the first one must finish before the second one starts. This is not difficult to implement in a source-level debugger (see GDB's `scheduler-locking` parameter), however this may introduce a thread deadlock in the execution if some locks are taken outside of the profiling regions.

In the following section, we look at how to present the profiling results to the developers.

3.2. Presenting the Measurements

So far, we have presented where and what to measure, along with some nesting and interleaving problem that can occur during the profiling, but we did not discuss how to present the measurements to the user. Let us finish this document with that topic. (Deliverable D4 illustrates this aspect with more details, with the case-study of an actual performance debugging.)

Raw counters

For a limited number of profiles, the developers may be interested in the raw values of the counters. These values provide precise information about the execution that advanced developers may be able to understand and exploit:

```
(gdb) profile info 1
| function profile[compute_forces_crust_mantle_dev]
| ================================================
| min_flt:                              13
| cycles:                      113,705,103
| branch-misses                    110,791
| task-clock:                          39.381
| branches:                     10,622,409
| instructions:                188,404,850
| stalled-cycles-frontend:      50,032,346
| stalled-cycles-backend:       15,004,422
| ...
```

Aggregated values

When the number of profiles increases, it quickly gets hard to process the raw values manually. So our tool can display a summary of a list of profiles, aggregated in different ways. As part of the prototype development, we implemented three common aggregators: minimum and maximum values, and arithmetic average:

```
(gdb) profile summary 2-12
| cycles                     avg:  11553677
|                            min:     13549
|                            max: 218766806
| ----------------------------
| stalled-cycles-backend     avg:  1517803
|                            min:  1178455
|                            max: 26184303
| ----------------------------
| instructions               avg:  19149774
|                            min:     18745
|                            max: 367804777
| ...
```

Plotted charts

Charting is a convenient way to represent a large number of measurements. They allow developers to visualize the variation and find correlations in the different counters.

We designed a debugger command that allows developers to interactively plot the charts they are interested in, as well as performing some simple operations on the dataset:

1. Prepare the profile results for the plot:

   ```
   (gdb) profile graph plot-all all
   cycles | 77230671 73753874 79916837 64098704 82873074  ...
   instructions | 120456822 120458577 120451359 120448095 ...
   task-clock | 32.186531 30.738873 33.305455 26.724866   ...
   ...
   ```

2. Launch the interactive graph plotter, either on the same computer or on another one:

   ```
   (gdb) profile graph offline
   ```

3. Copy and past the lines that should be plotted. For instance, the number of instructions against the number of cycles:

   ```
   instructions | 120456822 120458577 ... <
   cycles | 77230671 73753874 ... y2
   ```

```
| Rendering chart plot of instructions (sorted), cycles
| into /tmp/chart-20170117-155044.png
```

4. See Figure 3.2.1 for the chart generated.

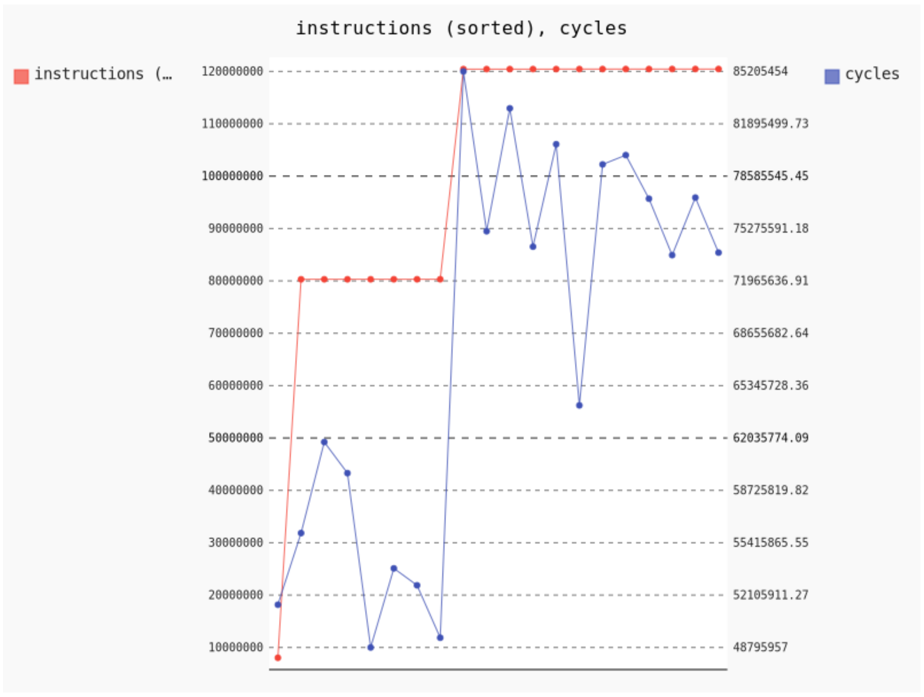

Figure 3.2.1: Example of a multi-profile plot, showing the instruction count and number of CPU cycles, sorted over the number of instructions.

In step 3, we can notice two modifiers at the end of the line. These modifiers are used to customize the way the measurements are plotted. By default, they are all on the same axis, plotted one against each other. The modifiers change this behavior:

`y2` puts the serie on the secondary axis

`y#` hides the serie (use it with sorting)

`<` sorts the serie in increasing order (when the measurement order is not important), and use the same ordering in the following series (to ensure the column consistency)

`@n` keep only the values equals to `n` in this serie, and apply the same selection in the following series

`/` divides the current serie by the next one

`+` sums the current serie with the next one

These different modifiers were implemented to facilitate the debugging of the performance problem presented with the Deliverable D4. The list can be easily extended in Python source code.

Chapter 4

Conclusion

In this report, we presented the work done during the first semester of 2016 for the deliverable D2 of the Nano2017/DEMA project. This deliverable was defined as an early work towards interactive performance debugging. We successfully developed a prototype of an interactive profiling tool based on GDB, the source-level *functional* debugger of the GNU project.

Our interactive profiling tool allows developers to finely control the execution of their application, and start/stop on demand the profiling of the execution. We interfaced our tool with different well-know profiling tools so that developers can measure performance metrics they are already familiar with.

At the end of semester, our tool was able to profile different code regions (a function execution, a particular section of the code (defined with the breakpoint syntax) as well as regions manually controlled by the debugger user, in the command-line interface. On these regions, our tool can measure different kinds of information, coming from existing profilers. As part of the prototype development, we implemented the support of `/proc` kernel counters, `perf stat` hardware/kernel counters and debugger breakpoint hits counters.

During the second semester of 2016, we continued this work with the D4 deliverable. We turned towards parallel computing and focused our study on the performance debugging of OpenMP applications running on NUMA computers.

References

[1] OpenMP 4.0 standard. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[2] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/. Accessed: 18.04.2014.

[3] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, August 2000.

[4] Andi Drebes, Jean-Baptiste Bréjon, Antoniu Pop, Karine Heydemann, and Albert Cohen. Language-Centric Performance Analysis of OpenMP Programs with Aftermath. In *International Workshop on OpenMP (IWOMP)*, pages 237–250, October 2016.

[5] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 39(4):49–57, April 2004.

Chapter A

Appendix

A.1. Access to the Source Code

A.1.1 Download

mcgdb

- `http://dema.gforge.inria.fr/delivrable/2017-01_mcgdb/mcgdb.tgz`
- `git+ssh://USER@scm.gforge.inria.fr/gitroot/dema/mcgdb.git`

Requirements

```
pip install colorlog pysigset enum34 pyparsing
```

- Profiling
  - Linux Perf (mandatory)
    `https://perf.wiki.kernel.org/`
- Logging
  - Colorlog (recommended)
    `https://pypi.python.org/pypi/colorlog`
- GDB internal thread safety:
  - Pysigset (recommended)
    `https://pypi.python.org/pypi/pysigset/`
- Task/OpenMP
  - Enum34 (Python2 only)
    `https://pypi.python.org/pypi/enum34`

    - pyparsing
      `https://pypi.python.org/pypi/pyparsing`

- Documentation
    - Rendering
        - Sphinx
          `https://pypi.python.org/pypi/Sphinx`
        - Sphinx RTD theme (optional)
          `https://pypi.python.org/pypi/sphinx_rtd_theme`

A.1.2 Installation

Our developments were done with GDB 7.12 and Python 3.5. GDB supports Python 2 and Python 3, and our support should work with both versions.

Load mcGDB from GDB

Put in `.gdbinit`:

```
python
sys.path.append("/path/to/Python")
try:
  import mcgdb
  #mcgdb.initialize()
  mcgdb.initialize_by_name()
except Exception as e:
  import traceback
  print ("Couldn't load Model-Centric Debugging: {}".format(e))
  traceback.print_exc()
end
```

Put in your `$PATH`:

```
ln -s $(which gdb) mcgdb
ln -s mcgdb mcgdb-omp
```

Then load your binary with `mcgdb-omp`

Convenience with GDB/mcGDB

Add these lines to your `.gdbinit`:

```
## almost mandatory:

set height 0
set width 0
```

```
## for convenience:

set breakpoint pending on
set print pretty
set confirm off

# for debugging

set python print-stack full
```

A.1.3 Compile `libmcgdb-omp`

```
cd $MCGDB_PATH
cd model/task/environment/openmp/capture/preload
make # generates __binaries__/libmcgdb_omp.preload.so
```

A.1.4 Compile `libmcgdb_perf_stat.preload.so`

```
cd $MCGDB_PATH
cd model/profiling/
make # generates __binaries__/libmcgdb_perf_stat.preload.so
```

A.1.5 OpenMP environment

Our OpenMP profiling support works with Intel OpenMP.

Intel OpenMP

Intel OpenMP should be compiled with debugging symbols (and optionally OMPT support). Here is the procedure:

```
mkdir -p intel_omp/{build,install}
cd intel_omp
INTEL_OMP_HOME=$(pwd)
# url checked 17/12/2015
wget https://www.openmprtl.org/sites/default/files/libomp_20150701_oss.tgz
tar xvf libomp_20150701_oss.tgz

cd build
cmake -DCMAKE_C_FLAGS="-g -O0"                               \
      -DCMAKE_INSTALL_PREFIX:PATH=$INTEL_OMP_HOME/install \
      -DLIBOMP_OMPT_SUPPORT=true                          \
      $INTEL_OMP_HOME/libomp_oss/

# -- LIBOMP: OpenMP Version       -- 41
# -- LIBOMP: OMPT-support         -- true
# -- LIBOMP: Build                -- 20150701
# -- LIBOMP: Use predefined linker flags      -- true

make && make install

export LD_LIBRARY_PATH=$INTEL_OMP_HOME/install/lib

# compile OMP application
path/to/clang -fopenmp -g $FILENAME

# check that $INTEL_OMP_HOME/install/lib/libiomp5.so is actually used
ldd a.out | grep libiomp5.so

# tested with clang 3.5.0
clang --version
# clang version 3.5.0
# (https://github.com/clang-omp/clang.git a5dbd16db2515a5b2fa82c7dd416d370968646b1)
# (https://github.com/clang-omp/llvm 1c313aa94183e765c450be6bda3913e22abc3073)
# Target: x86_64-unknown-linux-gnu
```

A.2. Execution Profiler Interface

This is the Python interface that should be implemented to integrate an existing exection profiler into `mcgdb` interactive profiler (discussed in Deliverable 2, Section 2.2).

```
from collections import OrderedDict

import gdb

class generic_info():
    "generic template"
    name = "generic template"

    def __init__(self):
        self.__results = OrderedDict(("name", value))

    def start(self):
        pass

    def stop(self, paused=False):
        pass

    def to_log(self, ongoing=False):
        return self.__results

    @property
    def results(self):
        return self.__results

__COUNTERS__ = [generic_info]
```