

FAST User's Manual

Document version 0.06.02

1 Introduction

The Flexible Analysis and Storage Toolkit (FAST) is a set of tools designed to help improve the performance—primarily the speed—of singly-threaded computer programs written in C or C++. It has components for the collection, analysis, and display of performance data.

The tools in the suite are designed to allow the user access to as much of the measured data as possible, and to customize his view of the data. They are designed for *exploratory data analysis*, because understanding the performance of large and complex programs is a task that requires as much creativity as many physics analyses.

2 Why does this tool exist?

Applications in High Energy Physics (HEP) have several features that make the standard profiling tools difficult to use for performance analysis. The applications run a large body of C++ code comprised of many independently developed algorithms, assembled into a serial pipeline. These algorithms manipulate many complex data structures. The complexity of these applications makes it difficult to localize the causes of poor performance. Often, a body of code performs poorly in one calling context, but not in others. To diagnose such situations, call path information is essential.

A detailed discussion of the requirements of FAST can be found in the companion document **The FAST project**, available from [the FAST project web site](#).

3 Overview

FAST consists of several subsystems:

- SIMPLEPROFILER, the data collection component,
- PROFGRAPH, a *call graph*¹ analysis tool,

¹A call graph is a directed graph that represents calling relationships between subroutines in a computer program. [1]

- PROFSAVE, a tool for managing large amounts of profiling data, and
- PROFSTATS, a tool for statistical analysis of profiling data.

In this prerelease, the PROFSAVE and PROFSTATS components are not yet included.

3.1 SIMPLEPROFILER

SIMPLEPROFILER is the data collection part of FAST. SIMPLEPROFILER contains a *sampling-based* (or *statistical*) call graph profiler for use with single-threaded applications. SIMPLEPROFILER is non-intrusive; no instrumentation of the target program is necessary.²

As a sampling profiler, SIMPLEPROFILER periodically samples the target program, assembling, over time, a statistical estimate of the time each function or subroutine of the target program consumes. As a call graph profiler, each sample records not only the active function, but the full *call stack*³ of the target program.

The SIMPLEPROFILER package includes the measurement tool (the dynamic library `libSimpleProfiler`—the suffix of the filename depends on your operating system), as well as some programs to simplify the use of the measuring tool:

- `profdemangle`, a program to provide “demangled” C++ function names for each C++ function listed in the profiler output, and
- `profrun`, a program that automates the process of collecting profiling data and system performance data, which is useful when studying the effect of the system state on the behavior of the program being profiled.

End-users do not generally need to call `profdemangle`; it is automatically called by `profrun`.

3.2 PROFGRAPH

PROFGRAPH is the call graph analysis tool in FAST. It uses GRAPHVIZ to produce a visualization of the call graph represented in the data collected by SIMPLEPROFILER. Because the call graph for large programs can be unwieldy, PROFGRAPH allows the user to filter the input. The user specifies a function on which to concentrate, and can optionally determine what amount of the call stack should be included in the output. One can also filter out little-used paths, to further reduce the clutter common to the call graphs of large programs.

Call graph analysis can be a valuable first step when one is trying to understand the dynamic structure of a large body of code with which one is not familiar. It can also be invaluable in understanding conditions in which the speed of a given function is dependent upon the context in which it is called.

4 Getting started

4.1 Prerequisites

Instructions for downloading and building the FAST software, including the components of SIMPLEPROFILER, are available at the project web site: <https://cdcvs.fnal.gov/redmine/projects/show/fast>.

FAST is built upon other software, including:

²The target executable must be built with debug symbols for SIMPLEPROFILER to be able to identify function names.

³A *call stack* is a stack data structure that stores information about the active subroutines of a computer program. [2]

- LIBUNWIND, which is used to capture the call stack, and is needed for building `libSimpleProfiler`,
- LIBBFD and its dependencies, an optional component used to improve the quality of resolution of addresses to function names, and
- LIBRT, an optional component to provide high-resolution timer support for run time counters.

Optionally used are:

- Ruby, either version 1.8.x or version 1.9.x, which is used for creating call graphs,
- GRAPHVIZ, which is used to create graphical displays of call graphs,
- PS2PDF, for the creation of PDF files from GRAPHVIZ,
- SAR, for the collection of system activity data during profiling,
- NUMACTL, to control the NUMA policy for the profiled program, and
- R, for statistical and graphical analysis of profiling results.

4.2 Quick start

To obtain your first set of measurements with `SIMPLEPROFILER`, after building the tools, run the `profrun` command, giving it the name of the program you want to profile as an argument:

```
$> source <path-to-fast>/etc/setup
$> profrun myprog
```

This will run the program `myprog` while collecting both profiling data and system performance data. Output of this process will be written to several files, with the name `profdata_n_<x>`, where `n` is the process id of the profiled program, and `<x>` takes many different values. The content of each of these files is explained in §7, below.

5 How FAST collects data

5.1 Profiling data

As stated above, `SIMPLEPROFILER` is a sampling, or statistical, profiler. The actual work of collecting samples is done by code in a dynamic library, `libSimpleProfiler`. Once loaded by the operating system's dynamic loader, `libSimpleProfiler` registers a signal handler to respond to the `SIGPROF` signal, and then sets up an interval timer to send the `SIGPROF` signal every ten milliseconds. Each call to the registered signal handler captures a single *sample*.

A sample consists of a series of memory addresses which make up the call stack, the location in memory to which each called function will return when that function is completed. These samples are buffered in memory, and when a sufficient number have been recorded, they are written to the raw data file described in section 7.5.

When the program's *main* function exits, the raw data files written during data collection are processed, and the several output files described in section 7 are written.

5.2 System activity data

On supported systems⁴, the `profrun` script offers the ability to collect system activity information, using `sar`. This information can be used to identify “bad” profiling data

⁴Currently only Linux.

```

linux$ profrun
Usage: profrun [ profrun options ... ] program [ program options ... ]
where program is the program to be profiled and options are indicated
below.

Examples: profrun examples/ex01/Linux.x86_64/ex01
          profrun -s examples/ex01/Linux.x86_64/ex01

Profrun options
-h or --help          Print this help message.
-v or --version       Print SimpleProfiler version.
-s or -{}-sar         Run program while collecting SAR data (Linux only).
-n or --numactl       Turn off the use of numactl (Linux only).

Program options
Any options used by the program you wish to profile.

```

Figure 1: Printout from profrun, run with no arguments.

sets—where “bad” means that system activity from other processes interfered with the operation of the program being profiled, or collection of profiling data, rendering the profiling data suspect.

If recording of system activity data is selected, several additional data files are written, as described in section 7.6.

6 Measuring your application

The easiest way to collect profiling data for your application is to use the script `profrun`. If, for some reason, `profrun` fails in your environment, it is possible to collect data by using `libSimpleProfiler` directly. This is not recommended for normal use.

6.1 Using profrun

To enable use of `profrun`, one must first set up the appropriate environment. The script `etc/setup` is provided to do this, for both C-family and sh-family shells.

Running `etc/setup` puts the directory containing the programs provided by FAST, including `profrun`, onto the `PATH`. Running `profrun` with no arguments yields a printout of `profrun`’s help message. Figure 1 shows the printout. On a platform for which SAR data collection is not yet supported by FAST, the `-s|--sar` switch is not available.

6.2 Using libSimpleProfiler directly

It is possible to use `libSimpleProfiler` directly, without relying on the `profrun` script. To do so, one must arrange to have the dynamic library `libSimpleProfiler` loaded before program start-up. On Linux, this can be done using `LD_PRELOAD`. Please consult your system’s documentation for how to use these environment variables.

Users who use `libSimpleProfiler` directly should also run `profdemangle` on the *names* file (see Section 7), to put it into the canonical format. `profdemangle` takes a single argument, the filename of the *names* file that is to be modified.

7 Understanding the profiling data

Each time you collect data using `SIMPLEPROFILER`, a set of files are written containing the profiling data. The two files of primary importance written by `SIMPLEPROFILER` are the *names* file and the *paths* file. Each of these is explained, in detail, in sections 7.2 and 7.3. The contents of the other files are both simpler and less important; they are summarized together in section 7.5. Additional files written when the `-s|--sar` switch is used are summarized in 7.6.

7.1 The raw data file

The raw data collected during profiling is written to the *raw data* file. This file has a name in the format `profdata_<pid1>_<pid2>_<disambiguation string>`. For programs that do not fork additional processes, `<pid1>` and `<pid2>` will be the same number, the id of the process that ran the program being profiled. For programs that fork additional processes, `<pid1>` is the process id of the parent, `<pid2>` is the process id of the process whose data is in that file, and `<disambiguation string>` is a string of random numbers to prevent runs that use many `<pid1>` from inadvertently overwriting data.

The raw data file contains an eight-byte header, which contains the identifying four-character “FAST”, three bytes of file version information, and one byte telling the size *n* (in bytes) of the pointers recorded in the sample data.

Following the header is the sample data, in the format of a series of call stack dumps. Each call stack dump consists of an *n*-byte unsigned integer, which specifies the depth of the call stack, followed by that many additional *n*-byte unsigned integers, each of which corresponds to an address read from the call stack.

7.2 Contents of the names file

Data for every function observed in any call stack sample is summarized in the *names* file. This file has a name in the format `profdata_<pid1>_<pid2>_<disambiguation string>_names`. The *names* file contains tab-delimited columnar data. Each line of the output contains information about a single function or subroutine of the observed program.

There are at least nine columns in the raw output file; an optional tenth column is added in post-processing. This post-processing is done automatically by `profrun`. These columns contain the following information.

1. the *function id*, an integer identifier assigned for its associated function. These identifiers are unique only within the scope of a single run of `SIMPLEPROFILER`; a second run of the same program might not yield the same function identifiers for some (or more rarely all) of the functions.
2. The *address* of the function in memory.
3. The *leaf count* for the function. This is the number of times the function was observed as the last entry in the call stack. This number is proportional to the amount of time that was spent executing the code of the corresponding function, not including functions it calls. Some profilers call this the *exclusive* time taken by the function.

4. The *total count* for the function. This is the total number of times the function appeared in the call stack. This number can be informative when analyzing recursive functions. Because a single sample can observe the same function multiple times, this number is not related in any simple way to the time spent in this function.
5. The *path count* for the function. This is the total number of times the function was observed anywhere in the call stack. It is therefore always at least equal to, and often greater than, the *leaf count*. This number is proportional to the amount of time that was spent executing the code of the corresponding function plus the time spent executing all the functions it calls. Some profilers call this the *inclusive* time taken by the function. For functions that are not recursive, this number will be equal to the total count; for deeply recursive functions, this value can be much smaller than the total count.
6. The *leaf fraction* for the associated function. This is the ratio of the leaf count for this function divided by the total number of samples taken.
7. The *path fraction* for the associated function. This is the ratio of the path count for this function divided by the total number of samples taken. Because sampling actually begins slightly before the profiled program begins, this number is usually slightly less than one even for the top-level function of the program being profiled.
8. The (dynamic) *library* in which the associated function is located. A short version of the library name is printed, in order to help keep the display compact. See section 7.4 for the location of the full path information. Note also that functions compiled directly into the profiled program are shown as belonging to a library with same name as the executable.
9. The *name* of the associated function. For C++ functions, this is the so-called “mangled” function name.

In post-processing, which can be performed automatically by `profrun`, an optional tenth column can be added. This column contains the so-called “unmangled” function name. For non-C++ functions, this does not differ from the contents of the previous column. For C++ functions, this column shows the name of the function in a much more human-friendly format. Because some C++ implementations (e.g., GCC) can create, for implementation-specific purposes, more than one implementation for a given function, it is possible that the some unmangled function names may appear more than once. Only the function identifiers and mangled function names are guaranteed to be unique.

Because it is possible to obtain some samples with zero-length paths, the sum over all functions of the leaf counts might be less than the total number of samples taken. A zero-length sample is obtained when `LIBUNWIND`, engaged from the signal handler function, is unable to find the stack of the main program.

7.3 Contents of the paths file

The function return addresses, written into the raw data, are converted into the relevant function names. Each observed sequence of function names is called a *path*; data for each distinct path observed by the profiler is recorded in the *paths* file. This file has a name in the format `profdata_<pid1>_<pid2>_<disambiguation string>_paths`. The paths file contains line-oriented data; each line corresponds to a distinct observed call path. Each line consists of a series of tab-separated integers. On each line, the first value is the id for that path, and the second value is the number of times that path was observed during sampling. The remainder of the line is a sequence of one or more function ids, denoting the functions observed in the call path. These are the same function ids as are used in the *names* file.

7.4 Contents of the libraries file

Data for each (dynamic) library observed during profiling is summarized in the *libraries* file. This file has a name in the format `profdata_<pid1>_<pid2>_<disambiguation string>_libraries`. The *libraries* file contains line-oriented data; each line consists of tab-separated fields. Each line corresponds to an individual library. The first field in the line is the full path to the library; the second field is the short name (as used in the *names* file) for this library, and the third field is the sum of the leaf counts for all the functions belonging to this library.

The full path to the library is printed, to help distinguish between different versions of libraries with the same name. Note also that functions compiled directly into the profiled program are shown as belonging to a library with same name as the executable.

7.5 Other files

SIMPLEPROFILER writes three additional files, which contain information that is generally not of direct use to the end-user. Each of these files has a filename of the format `profdata_<pid1>_<pid2>_<disambiguation string>_suffix`, where the *suffix* is one of `maps`, `sample_info`, `timing`, or `totals`.

The maps file

Under operating systems that support the `proc` filesystem, the *maps* file contains data describing the memory locations into which the different dynamic libraries were loaded. On such systems, this file is a copy of the `proc/<pid2>/maps` file. Under operating systems that do not support the `proc` filesystem, the *maps* file is empty.

The timing file

Contains timing information for when profiling began and ended. It makes use of the `gettimeofday()` function, the `rdtsc` instruction, and, on platforms that support it, the `clock_gettime()` function as timing sources. Due to power management and other factors which can alter the rate at which a timing source accumulates, one or more of the timing sources may be skewed from the others at times, so knowledge of the system on which profiling occurs is useful for selecting the correct timing source. Each source provides a start count, end count, and the difference between them.

The totals file

The *totals* file contains the total count of several distinct types of quantities: the number of samples observed, the number of distinct functions observed, and the number of distinct paths observed.

The debugging file

This file contains information that is sometimes useful to the developers of FAST, but is not relevant for users.

7.6 The optional system activity data files

On systems that support multiple simultaneous users, use of system resources by processes other than that being profiled can render the profiling information collected by a

sampling profiler unusable for the purpose for which it is intended. To help avoid such problems, it is useful to be able to evaluate system activity data describing the operating conditions of the system on which the profiled program was run. Such data can be used to identify profiling runs rendered suspect because of heavy load, memory swapping, or other deleterious conditions caused by other processes.

`profrun` supports collection of system activity data on Linux, using the `sar` facility, which is enabled using the `-s|--sar` switch.

If the use of `sar` is activated, `profrun` begins data collection using `sar -A` before the program to be profiled is started. Data are sampled at 30-second intervals. `sar` data collection is stopped after the program being profiled exits. For programs that run in less than the time of the system data sampling, the generated data files may be empty.

A total of ten additional files are written when `sar` collection is enabled:

1. The file `profdata_<pid1>_sar` contains the binary data written directly by `sar`.
2. The file `profdata_<pid1>_io` contains I/O and transfer rate statistics, from `sar -b`.
3. The file `profdata_<pid1>_paging` contains paging statistics, from `sar -B`.
4. The file `profdata_<pid1>_interrupts` contains interrupt summary statistics, from `sar -I SUM`.
5. The file `profdata_<pid1>_cpu` contains per-processor and total CPU utilization statistics, from `sar -P ALL`.
6. The file `profdata_<pid1>_load` contains run queue length and load averages, from `sar -q`.
7. The file `profdata_<pid1>_memory` contains memory and swap space use statistics, from `sar -r`.
8. The file `profdata_<pid1>_memrates` contains memory statistics, from `sar -R`.
9. The file `profdata_<pid1>_ctxswitch` contains system switching statistics, from `sar -w`.
10. The file `profdata_<pid1>_swapping` contains system swapping statistics, from `sar -W`.

For detailed descriptions of the collected data, please refer to the manual page for `sar` on your system.

8 Generating call graphs

The program `profgraph` can be used to generate a graphical view showing the calling relation of the functions in your program. This view is generated from the sampled function calls, so it will usually not show all the possible call paths that the program could have executed.

Running `profgraph -h` yields a printout of `profgraph`'s help message. [Figure 2 on the next page](#) shows the printout.

`profgraph` produces a graphical display of part of the full call data. The user selects (by function id, which can be obtained from the `names` file) a function on which the display will concentrate. `profgraph` first selects only those paths which contain the function indicated by the user. It then filters out paths with a path count less than that the value for `trim-count` specified on the command line. Finally, it selects those functions in a window of size indicated by `nodes-up` and `nodes-down` around the indicated function, and produces a graph containing those functions.

Each distinct function is indicated by a node on the graph. Each node contains:

- the *function name*, shortened for display purposes, and only if the `-n|--names` switch is supplied,

```

linux$ profgraph -h
profgraph - produce a Graphviz graph from SimpleProfiler profiling data.

SYNOPSIS
  profgraph [opts] run-id [func-id [nodes-up [nodes-down [trim-count]]]]

DESCRIPTION

  Produce a Graphviz graph for the profiling data from the run specified
  by run-id. The graph is 'centered' on the function specified by
  function-id. If nodes-up is supplied, only that many nodes in the up
  direction (toward main) will be shown; if not used, 5 nodes will be
  shown. If nodes-down is supplied, only that many nodes toward the leaf
  will be shown. If not supplied, 5 nodes will be shown. If trim-count is
  supplied, only paths in the path file with a path count greater than or
  equal to trim-count will be retained in the graph. If not supplied, no
  trimming is done.

-h, --help
  Print this help and exit.

-v, --version
  Print the version number and exit.

-n, --names
  Print function names rather than function IDs in the printed graph.

-f, --format=FORMAT
  Use graphics format FORMAT for Graphviz output. Any graphics format
  understood by 'dot' can be used. In addition, if --format=pdf is used,
  PDF output will be generated by using the layout engine with -Tps2, and
  then running ps2pdf.

-l, --layout=LAYOUT
  Use the layout engine LAYOUT. If not specified, the default it to use
  dot. Other choices include fdp, neato, and circo.

```

Figure 2: Printout from `profgraph -h`.

- the *function id*, as found in the *names* file,
- the *path count* and *path fraction* for the function,
- the *leaf count* and *leaf fraction* for the function, and
- the *library* in which the function is found.

The arcs connecting the nodes show which function calls which function. The number on the arc indicates the sum over paths of the path counts of each path containing that function call.

To help the user identify the function of interest in large graphs, the node for that function is colored green. If the call graph is sufficiently large, the call paths with the

largest fraction of counts will appear in red, and with wider arrows connecting the nodes.

9 Known limitations

SIMPLEPROFILER relies upon LIBUNWIND, and thus is constrained by its limitations.

On 64-bit Linux systems, LIBUNWIND has troubles unwinding the stack through system calls. A sign of this problem is the appearance of “disconnected” paths, that is, paths that do not contain the *main* function of the program being profiled. These failures most often appear when a sample is taken while the program is executing system code. Some number of these paths are failures of symbol lookup as opposed to failures to unwind through system code and installing `binutils-devel` and `zlib-devel` to enable `bfd` resolution support may resolve some of these failures. In addition, missing unwind information in system libraries contributes to this problem and newer versions of system libraries, or compilation of system libraries using a more recent library may resolve these issues.

There is no installation target for the build system; currently, the user must delete unnecessary files and directories himself, and is then free to move the remaining directory tree.

Bibliography

- [1] **Wikipedia:** http://en.wikipedia.org/w/index.php?title=Call_graph&oldid=324940773.
- [2] **Wikipedia:** http://en.wikipedia.org/w/index.php?title=Call_stack&oldid=325234638