The FAST project

Document version 0.06.02

1 Introduction

1.1 Purpose

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.

1.2 Scope

FAST includes tools for many of the phases of performance tuning:

- a sampling-based profiler for collecting measurements,
- a documented file format in which profiling data are written,
- tools for extracting *call path*, *function call* and *library call* information from the raw data,
- tools for the collection of information about the environment in which the profiled program is run,
- tools for organization and storage of these data,
- tools for graphical display of call paths,
- tools for statistical analysis of profiling data, including statistical comparison of large numbers of program executions, allowing the user to understand the statistical significance of apparent performance differences, and
- a web application that allows for the storage and comparison of profiling data.

The FAST data collection tools are supported on Linux systems.

The FAST analysis tools are supported on a wider variety of platforms; they will work wherever the tools on which they are based will work, which includes Linux, Mac OS X and Microsoft Windows systems.

1.3 Project rationale

Development of parts of FAST began, by Jim Kowalkowski, in 2001. At that time there was a need to profile the reconstruction code of the CDF and DØ experiments, and the GNU profiling tool `gprof` was unable to deal with executables as large as those in question. Furthermore, `gprof`, when used in *sampling* mode, provided only summary data, and did not make available full call path information. `gprof` tries to reconstruct the call graph from its accounting of the caller of each observed function. But because the entire call stack is not sampled, much information is lost. The technique used by `gprof` to propagate information up the call tree is accurate only if the time taken by each function call is constant. When the function call time is variable, the results inferred by `gprof` can be misleading[1].

When used in *call graph* mode, `gprof` required a special build of the software to be profiled, and that build injected additional code at every function call site. This has the effect of substantially distorting the observed effect of small functions, which are common in object-oriented code.

In analyzing complex bodies of code it can be important to have call path information available. It is common to find that a given function is time-consuming when called through one path, but not when called through a different path. In the absence of call path information, such features of the code are not observable. Other tools available at the time this project was started did not capture full call stack information, or did not make that information available to the end-user.

In addition, `gprof` provides no facility for storing profiling results, organizing them, and comparing them. Since the goal of performance measurement is most often to understand where to modify the code to obtain speed improvements, it is important to be able to conveniently compare the behavior of the program in question before and after modification.

The tools in FAST address each of these issues and combine to provide a toolkit that can be used in part or in its totality, in whatever manner is most convenient for the task at hand.

2 Requirements

In this section, we enumerate

- *functional requirements*, which describe what the FAST software must do,
- *performance requirements*, which put constraints on how the software is to meet the functional requirements and
- *constraints*, which are imposed on the project for a variety of external reasons.

Some of these "requirements" may seem to fit in more than one category. We believe the categorization of the requirements is not nearly as important as their enumeration.

2.1 Functional requirements

1. Special compilation of the source code shall not be required. Because it is common for the code bases of the programs we wish to profile to be very large, it is inconvenient to recompile them to allow profiling.
2. Instrumentation of existing executables shall be possible. Because it sometimes takes special expertise to build the programs we wish to profile, and because it is not always easy to obtain expertise in building them, it is important for us to be able to instrument a program without rebuilding the program.

3. Complete call path information shall be collected. The behavior of a function often depends on the values of the data on which it is operating; this in turn often depends on the chain of function calls leading to that function.
4. Privileged (*e.g.*, root) access shall not be required for data collection. It is often necessary for us to perform profiling on machines to which we do not have privileged access.
5. The tools shall provide a mechanism to organize collections of data and to allow the statistical comparison of program performance before and after source code modification.
6. The tools shall provide means to allow the user to understand the high-level organization of large code bases, and to determine what organizational units of the software (*e.g.*, classes and libraries) are most time-consuming, as well as those individual functions are most time-consuming.
7. The tools shall provide a means to focus the user's attention on individual classes or functions. Because the code bases which we analyze often contain tens of thousands of functions, the ability to limit the quantity of information being considered is critical.
8. The tools shall be able to distinguish between functions that are always time-consuming, and those that are time-consuming under only specific conditions (*e.g.*, when called from a specific function).
9. The tools shall allow the collection of sufficient information regarding the execution environment to allow for the identification of anomalous profiling results due to effects other than the direct behavior of the code being profiled. For example, it must be possible to identify results that are unusual because the system on which the measurement was made was heavily loaded.
10. The tools shall allow the recording of sufficient information regarding the build environment to allow the user to understand and characterize the differences in program performance due to differences in the build environment. The exact character of the data to be recorded is determined by the user.
11. For programs that are able to record such information, the tools shall allow collection of information regarding the timing of the processing of the *units of work* appropriate for the task, at a level of granularity determined by the program. In HEP event processing applications, especially in the large and modular frameworks used by large experiments, measuring the time taken by each module on each event is often a key feature of understanding what parts of the program are in need of improvement.
12. The tool will not lose information about the paths and functions recorded in the raw samples during the summarization process.

2.2 Performance requirements

1. The time overhead entailed by the measurement process shall be small; to be concrete, we have kept the overhead below 5%.
2. The precision of measurements shall allow a 1% change in the time taken by a single function to be identified.

2.3 Constraints

1. Collection of profiling data on Linux systems shall be supported.
2. Collection of profiling data shall be supported on 32- and 64-bit systems.
3. Only Intel and Intel-compatible architectures shall be supported.
4. No kernel modification shall be required.

3 Architectural overview

FAST consists of a set of loosely-coupled tools, described in §3.1–3.3.

3.1 SimpleProfiler, the data collection tool

SimpleProfiler is a dynamic library which, when loaded, actives a signal handler that responds to the *sigprof* signal by capturing the complete call stack of the running program at that time. SimpleProfiler is written in C++, and uses the libunwind[1] library to capture the call stack. In order to keep the execution of the signal handler fast, the addresses that make up the call stack are not resolved to function names during data collection.

Call stack samples are batched in memory, and periodically written in binary format to a file. At the end of the program run, the recorded addresses for the call stack data are translated into function names. At this stage, several files are written:

1. A file containing function names (mangled function names, in the case of a C++ program). For each function, the library in which that function is found is also recorded, as is the number of samples in which the function appeared in the call stack, and also the number of times the function appeared as the leaf function in the call stack.
2. A file containing the address range to which each library was mapped.
3. A file containing a summary of the number of samples taken.
4. A file containing a summary of the number of samples for which the stack unwinding mechanism was unable to collect any data.
5. On Linux systems, a file containing information on the address ranges to which dynamically loaded libraries were mapped.
6. A file containing some information useful for debugging the stack unwinding mechanism.

The details of these files, including the file formats, will be provided in a forthcoming paper.

3.2 `profgraph`, a graphical analysis tool

`profgraph` is a tool that is used to generate graphical displays of call stack data. These are especially useful when analyzing large programs, or programs about which one has little knowledge. These graphical displays can be used to identify the functions which are primarily responsible for consuming program execution time. Facilities are provided to allow the user to select the function of primary interest for the display, as well as to limit the scope of the generated graphs.

`profgraph` is a Ruby program that uses AT&T's Graphviz [2] to produce its graphical output.

3.3 `perfdb`, the web application

`perfdb` is a web application that allows uploading of sets of measurements of a given program. It can manage metadata describing the circumstances under which the program was built, and also under which the program was run. The *build conditions* include information such as the compiler name and version, and the versions of important libraries; the *run conditions* include information such as the type of CPU of the machine on

[1]The libunwind library is available from `http://www.nongnu.org/libunwind/`.

which the program execution was profiled. `perfdb` provides means to compare different runs of the same program, so that the user can discover the effect of modifications in code.

`perfdb` is implemented in the Rails[3] web programming framework.

4 Release management and deployment

Currently, FAST is deployed in source code form only. Please see the FAST web site, `https://cdcvs.fnal.gov/redmine/projects/show/fast`.

5 Possible future enhancements

Two new tools are under development:

- `profsave`, a tool for saving profiling data in a local SQLITE database, and allowing data to be uploaded to a `perfdb` installation, and
- `profstats`, a tool for the statistical analysis of profiling data.

It would be useful to record when each *unit of work* is started, so that one may observe how each function's behavior varies across different units of work.

It may be interesting to allow measurement of the time at which each sample was collected. This would allow one to understand when the process executing the instrumented program has been swapped out.

Future deployment plans include bundling the Ruby analysis tools as a gem[4], and bundling the statistical analysis tools as an R [5] package.

A Glossary of terms

This appendix contains definitions of the some of the terms used in document, especially those terms defined to have a special meaning in the context of this project.

Call stack A stack data structure that stores information about the active subroutines (in C++, functions) of a computer program.

Call path A distinct call stack observed during the profiling of a program. A call path may be observed once, or more than once, in a given program run.

Leaf function The last function in a call path.

Unit of work Some programs are organized to perform the same or similar processing on a sequence of data elements presented to the program. We call each such data element a *unit of work*. In many HEP data processing programs the unit of work is the *event*; in many astronomical data processing programs the unit of work is the *image*.

Bibliography

[1] Varley, Dominic, *Practical Experience of the Limitations of GRPOF*, **Software—Practice and Experience**, vol. 23(4), 461–462 (April 1993).

[2] Graphviz is available at `http://www.graphviz.org`.

[3] Rails is available at `http://rubyonrails.org`.

[4] Ruby *gems* are redistributable libraries of Ruby code; see `http://www.ruby-lang.org/en/libraries/` for details.

[5] R is a free software environment for statistical computing and graphics. See `http://www.r-project.org` for details.