Benchmarking VAUCANSON and large C++ libraries with CBS

Florent D'Halluin

Technical Report n^o0902, June 2009
revision 2086

VAUCANSON is an extensive C++ library for the manipulation of finite state machines. Compared to its main competitor, OPENFST, VAUCANSON has major performance issues.

In order to improve the performance of VAUCANSON, a set of tools is required to analyse the library's behavior in terms of CPU time requirements and memory usage.

Up to March 2009, no existing profiling tool was fully adapted to VAUCANSON and practical to use.

CBS is a C++ Benchmarking Suite that measures the performance of C++ projects and provides tools to display, analyze and compare results in a human-readable form. It is used to profile VAUCANSON and it helps rewrite algorithms.

VAUCANSON est une bibliothèque C++ de manipulation d'automates finis. Par rapport à son concurrent principal, OPENFST, VAUCANSON souffre d'importants problèmes de performances.

Afin d'améliorer les performances de VAUCANSON, il est nécessaire d'avoir des outils appropriés pour analyser le comportement de la bibliothèque en termes d'utilisation de temps CPU et de gestion de la mémoire.

Jusqu'en Mars 2009, aucun outil de ce type n'était pratique à utiliser avec VAUCANSON.

CBS (C++ Benchmarking Suite) est une suite d'outils d'analyse de performances pour projets C++. Ces outils permettent de mesurer, d'afficher, et de comparer l'utilisation de ressources (temps, mémoire), dans un format accessible à l'utilisateur. Ils sont utilisés pour analyser VAUCANSON afin de réécrire les algorithmes les moins efficaces.

Keywords
Vaucanson, performance analysis, profiling, C++

Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22
d-halluin@lrde.epita.fr – http://publis.lrde.epita.fr/200905-Seminar-DHalluin

Copying this document

Copyright © 2009 LRDE.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being just "Copying this document", no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license is provided in the file COPYING.DOC.

Contents

Introduction 6

1 Comparison of performance analysis methods 9

1.1 Definitions . . . 9

1.2 Indicators . . . 11

1.3 Profiling mechanisms . . . 12

1.3.1 Sampling . . . 12

1.3.2 Instrumentation . . . 14

1.4 Existing tools . . . 14

1.4.1 GPROF . . . 15

1.4.2 CALLGRIND . . . 16

1.4.3 CBS . . . 17

1.4.4 List of profilers . . . 17

2 CBS structure & implementation 19

2.1 Description . . . 19

2.1.1 Features . . . 19

2.1.2 LIBBENCH . . . 20

2.1.3 Usage . . . 21

2.2 Output . . . 21

2.2.1 Text . . . 22

2.2.2 DOT . . . 23

2.2.3 XML . . . 23

2.3 Implementation 25
2.3.1 Archive structure 25
2.3.2 Measures 25
2.3.3 Building the call graph 26
2.3.4 Performance 27
2.3.5 Licensing 28

3 Integration in VAUCANSON **29**
3.1 Benchmarking VAUCANSON 29
3.1.1 Affected files 30
3.1.2 Interpreting results 31
3.2 TAF-KIT 31
3.2.1 Affected files 31
3.2.2 Measuring command execution 32
3.3 Improving performance 32
3.3.1 Optimizing algorithms 33
3.3.2 Example for quotient() 33

Conclusion & Perspectives **38**

A CBS macro interface **40**

B CBS Benchmarks **42**

C VAUCANSON LISTG & BMIG benchmarks **44**

Index **47**

List of Figures **48**

List of Listings **49**

References **50**

Introduction

There exists few open-source libraries to manipulate finite-state machines. VAUCANSON (Lombardy et al., 2004) and OPENFST (Allauzen et al., 2007) are two representative projects of this field. VAUCANSON is a large C++ project that focuses on genericity, while OPENFST works on a specific kind of automaton, weighted finite-state transducers (FSTs).

Compared to OPENFST, VAUCANSON has major performance issues: VAUCANSON's version of the determinization algorithm runs six times slower that OPENFST's version on a worst-case example. This performance gap is observed consistently on the algorithms that are available in both libraries. This is easily noticeable by users, who point out that VAUCANSON indeed runs much slower that OPENFST on the automata that they actually use.

This performance gap is caused by multiple factors. In order to maintain genericity, the library stucture is fairly complex, which induces additional processing costs compared to a lightweight structure that handles only one specific kind of automaton. Also, a complex structure leads to dense and sometimes obscure code in automaton manipulation algorithms. Dense code is hard to optimize, and many of VAUCANSON's algorithms could by improved through selective optimization.

The library structure is being reworked to make the overall code clearer and more efficient. There is ongoing work on the Algebra module, which is the mathematical core of VAUCANSON (Hamelin, 2009), and on the modeling of automaton data structures (Galtier, 2009). In order to measure the impact of those changes on the performance of VAUCANSON, an efficient benchmarking suite is required.

Structural changes could also open many possibilities for optimization within algorithm code. However, bottlenecks are currently hard to find because the code is complex. This makes optimization a long and difficult process. In order to selectively optimize the part of the algorithm code that is responsible for VAUCANSON's performance issues, an efficient performance analysis tool is also needed.

Performance analysis is not a new field, and many tools were designed to work on such issues. These tools range from the simplest scripts to the most complete and complex commercial tools. Some characteristics of the C++ language, such as the object-oriented paradigm and the possibility for parallel programming and multi-core support have to be given special attention. One of the earliest tools to deal with these considerations is TAU (Shende et al., 1998).

VAUCANSON, however, does not use parallel programming, which makes simple tools an attractive option. In 2005, a tool for regression benchmarks, Ranch, was developed at the LRDE (Desprès, 2005). Ultimately, its development was discontinued before its integration in

VAUCANSON. The first chapter of this report lists other tools that may adapt well to VAUCANSON and compares their characteristics to CBS, a benchmarking suite developed specifically for VAUCANSON in 2009.

Before CBS, there were essentially two uncoordinated performance analysis systems in VAUCANSON:

- A small C++ profiler to measure command execution time in TAF-KIT (command-line tools).
- A benchmark suite to measure algorithm efficiency on a set of common examples.

Despite having the same purpose, those systems were completely separated and were limited in precision and depth of analysis, using only one of the many performance indicators for a program (observed execution time).

CBS is a C++ Benchmarking Suite that brings together VAUCANSON's two performance analysis systems into a simple, comprehensive, and easy-to-use set of tools. CBS is designed to work well with VAUCANSON, and can be used in any C++ project. The second chapter of this report focuses on CBS and its implementation.

Using CBS in Vaucanson is promising: Benchmarking results highlighted a surprising loss of performance in a new alternative for automaton data stuctures, BMIG, which was thought to be more efficient than the original version of the data stuctures, LISTG. Also, in less than an hour, analyzing the implementation of one of the most used algorithms, quotient, led to straightforward optimizations that reduced execution time by 40% on the benchmark examples. The third chapter of this report details the integration of CBS in VAUCANSON and illustrates its use with the optimization of quotient.

The reader is expected to be familiar with the C++ syntax and to have a basic understanding of computer hardware and Unix systems.

Contributions

This report is the summary of the research and implementation work that I did at the LRDE from February to May 2009. My contributions to the VAUCANSON project include:

- Rewriting TAF-KIT's global timer system (in 2007).
- Implementing CBS.
- Integrating CBS in VAUCANSON.
- Benchmarking LISTG and BMIG implementations (using the existing benchmarking system).
- Optimizing the quotient algorithm on boolean automata.

Acknowledgments

I received invaluable help from many people, including:

During the research and implementation work:

- Jacques Sakarovitch
- Sylvain Lombardy
- Alexandre Duret-Lutz
- Jérôme Galtier
- Alex Hamelin

During the redaction of this report:

- Vincent Ordy
- Warren Seine
- Guillaume Sadegh
- Damien Lefortier
- Étienne Folio
- Samuel Da Mota

Chapter 1

Comparison of performance analysis methods

1.1 Definitions

This section defines some vocabulary terms used in the remainder of this chapter to compare performance analysis tools.

Benchmarking

Benchmarking is comparing the efficiency of different programs over the same set of examples that represent a large range of situations encountered by users. Its goal is to give an idea of how well a given program runs in relation to the most representative programs in the same field.

Profiling

Profiling and *performance analysis* refer to the investigation of a program's behavior using information gathered as the program executes. Its goal is to determine which section of the program to optimize in order to increase its speed or to reduce its memory usage.

Profiling is used to locate *bottlenecks*, areas in the program code that poorly spend a resource (CPU time or memory). The effect of bottlenecks can then be reduced via selective optimization in order to make the program more efficient.

Profilers

In this report, *performance analysis tools* and *profilers* refer to the same type of tools.

Profilers are designed to give information about a program's resource usage. They usually

link the functions defined in the source code to their execution time during a program run. The basic output of a profiler is a flat profile showing the average resource percentage used in each function. Some profilers have more complex outputs, including a graph representing the chain of calls between functions, refered to as *call graph*.

Profilers rely of different data gathering methods described in section 1.3. More information on profiling and profilers can be found on Wikipedia (Wikipedia, 2009c).

Optimization

The optimization of a program's efficiency can be done at several levels:

- **Design level**: this includes changing data structures and using algorithms that perform better on the manipulated data.
- **Source code level**: this includes improving the code quality to avoid obvious slowdowns. Note that code optimization can often make the code harder to understand, since it relies on using techniques and tricks to express the same sequence of operations in more efficient, but less natural way.
- **Compile level**: some compilers can automatically make a number of optimizations on the source code.
- **Run time**: *just in time* compilers may be able to perform run-time optimization by dynamically adjusting parameters according to a program's actual input or other factors.

Optimization should be selective and occur only when bottlenecks are located. The *Pareto principle* states that 20% of a cause produces 80% of its effects. In software engineering, an often better approximation is that 90% of the execution time of a program is spent executing 10% of its code. Optimizing outside of these 10% not only achieves little, but also tends to make the code obscure and to introduce new bugs.

Donald Knuth made the following statement on optimization:

> "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%."

More information on program optimization and performance tuning can be found on Wikipedia (Wikipedia, 2009a) and (Wikipedia, 2009b).

Overhead

During profiling, many ressource usage *measures* are made. *Overhead* refers to the additional cost of those measures.

When choosing a profiler, there is a trade-off between overhead and measure precision. Some profilers (such as CALLGRIND) can make very precise measures, but running the profiler on a

program is many times slower than running the program by itself. Others (such as GPROF) have little to no overhead, but measures can be less reliable.

Swapping

Swapping or *paging* is the transfer of pages between the main memory and an auxiliary store. Because of the I/O operations on the auxiliary device, swapping is a costly process, and is very noticeable by the user.

Excess swapping can make a program run many times slower than it would if it had enough available memory. A number of algorithms were designed to optimize swapping, but their performance usually depends on the type of programs being used.

1.2 Indicators

This section describes several indicators of a program's efficiency.

CPU time

The *CPU time* is the amount of time that would be needed to execute a program if that program were alone in memory and using a single CPU core. In other words, CPU time does not depend on the amount of running processes, and thus gives a reliable indicator of a program's efficiency.

CPU time can be divided into *user time* and *system time*:

- **User time** is the time spent in user-defined functions, i.e. in the program code and in the code of the libraries linked with it.
- **System time** is the time spent in system calls, i.e. in kernel code.

CPU time is hardware-dependent and will vary according to the CPU type and frequency, regardless of the number of cores in multi-core architectures. Since the CPU is not processing data during swapping, swapping does not have an influence on CPU time.

Wall time

Wall time refers to actual or real time, i.e. the time that is measured by looking at a clock on the wall.

CPU time measures do not account for the cost of operations when the CPU is not actively processing data, e.g. when swapping or establishing a connection to a network or hardware device. Wall time measures can give a good estimation of the cost of a program execution as perceived by the user. However, it depends more on hardware, configuration, and other running processes than CPU time.

On single-core systems, wall time is always higher than or equal to CPU time. A consequence of multi-core systems is that wall time can be lower than CPU time when computations are distributed over several cores.

Virtual memory

Virtual memory usage is the amount of memory allocated to a program. It it divided into *pages* that can be stored in RAM or in an auxiliary store, such as a hard disk drive.

Many programs have more allocated memory than they actually use. For example, programs using *garbage collection* will not give unused memory back to the system. Instead, they will keep it until it can be used again.

As a result of this, virtual memory usage tends never to decrease during a program execution. Its peak value is a good indicators of a program's efficiency, but it is difficult to locate the part of the program that actually uses this memory.

Resident Set Size

Resident Set size (RSS) is the portion of a program's memory that is stored in RAM. It gives a rough idea of how much memory is actively used by the program. The main drawback of using Resident Set Size as a performance indicator is that it depends on the system's swapping policy and on other running programs.

1.3 Profiling mechanisms

This section describes two main mechanisms used in profiling tools.

1.3.1 Sampling

Sampling is measuring performance by interrupting the target program at regular intervals to find out which function is being executed. Time consumption per function can then be computed using statistical approximation on the measures.

Sampling profilers are typically less accurate and specific than instrumenting profilers, but have little overhead. In practice, they can give a good picture of a program's execution and are not as intrusive as instrumenting profilers. This reduces the impact of the measurement process on the measures and has no risk of introducing bugs in the program.

A straightforward implementation of sampling would be composed of 3 steps:

Listing the monitored functions:

In the source code or in the compiled code, one lists relevant symbols and their position in memory, so that probing the *program counter* can be used to find out which function is being executed.

Computing the number of calls per function:

Given:

- m, the number of measures.
- F_i, the function being executed at measure i.
- c_F, the number of calls for a given function F.

The number of calls per function c_F can be expressed:

$$c_F = \sum_{i<m} \begin{cases} 1 & if\ F_i = F\ and\ F_i \neq F_{i-1}, \\ 0 & otherwise. \end{cases}$$

In other words, count one call per non-consecutive appearance of a function in the measures.

Note that given:

- Δ_m, the interval between measures.
- Δ_F, the actual time spent in a call of the function F.
- Δ_{Δ_F}, the interval between two calls of the function F.

1. Any function call for which $\Delta_F < 2\Delta_m$ will not always be listed.
2. Any two function calls for which $\Delta_{\Delta_F} < 2\Delta_m$ will sometimes count as one.

This explains why sampling profilers are typically less numerically accurate.

Computing the time spent in each function:

Given:

- t, the observed time for the whole execution (on most platforms, Wall time, CPU time, user time and system time are observable).
- t_F, the total time spent in a given function F.

- a_F, the average time spent per call of the function F.

The time spent per function t_f can be expressed:

$$t_F = \frac{\sum_{i<m} \begin{cases} 1 & if\ F_i = F, \\ 0 & otherwise. \end{cases}}{m}$$

The average time spent per function a_F is simply:

$$a_F = \frac{t_F}{c_F}$$

1.3.2 Instrumentation

Instrumentation is measuring performance by adding instructions in the target program to collect measurement data.

Instrumenting a program always has an impact on its execution. It can potentially cause inaccurate results and *heisenbugs* (bugs that occur only in either the instrumented or the non-instrumented version of the same program).

However, instrumentation can be very specific and carefully controlled to have a minimal impact on measures. The impact on a particular program depends on the placement of instrumentation points and on the measurement mechanism.

Instrumentation can be:

- **Manual**: Done by the programmer, e.g. by adding calls to measurement functions (as used in CBS).
- **Automatic**: added to the source code by an automatic tool or by a specific compiler option, e.g. `gcc -pg ...` for GPROF.
- **At runtime**: directly before execution, the code is instrumented so that the program run is fully supervised by an external tool, e.g. CALLGRIND.

Hardware support for trace capture means that on some targets, instrumentation can be one machine instruction. The impact of instrumentation can often be predicted and deducted from the results.

1.4 Existing tools

This section compare the profilers GPROF, CALLGRIND, and CBS, then gives a list of other tools.

Overhead estimates are made on a small demo program available in the CBS archive in `demo/comparison`. It contains generic operations on file I/O, memory allocation, and list

manipulation. The demo program is compiled using options specific to each profiler. Using the demo with no specific option gives the result in Listing 1.1.

Listing 1.1: Generic demo timing.

```
..cbs/_build/demo/comparison $ time ./demo-generic 7
[...]
./demo-generic 7  2.35s user 1.14s system 99% cpu 3.496 total
```

The flat profile was predicted when designing the program (Listing 1.2). n is a complexity parameter. In the examples, $n = 7$.

Listing 1.2: Predicted flat profile.

```
Task                 Calls  Recursive/cyclic calls  Total calls
parent()               1              0                  1
file_io()              n            n * n            n * (n + 1)
new_integer()          n            n * n            n * (n + 1)
list_push_back()       n            n * n            n * (n + 1)
```

Recursive calls happen when a task calls itself. Cyclic calls happen when tasks call each other in a circular fashion. Note that all profilers may or may not include recursive calls and cyclic calls in the flat profile.

Overhead estimates are gathered in Figure 1.3 for easy comparison.

Profiler	Total time	Overhead
no profiler	3.49	0%
GPROF	3.45	0%
CALLGRIND	227.88	6500%
CBS	3.21	0%

Figure 1.3: Overhead estimates for different profilers

1.4.1 GPROF

GPROF is C/C++ profiler that uses both instrumentation and sampling. It was written in 1993 as part of the GNU Binutils (Osier, 1993). Instrumentation is used to create the call graph and the actual timing values are obtained by statistical sampling.

GPROF is used in 3 steps:

1. Compile the source code using `gcc -pg` or `g++ -pg`.
2. Execute the program to gather data into a file (`gmon.out`).
3. Extract results in a readable form using `gprof <program>`.

GPROF has irrelevant overhead on the demo program (Listing 1.4).

Listing 1.4: GPROF demo timing.

```
..cbs/_build/demo/comparison $ time ./demo-gprof 7
[...]
./demo-gprof 7  2.30s user 1.15s system 99% cpu 3.452 total
```

The flat profile does not account for the total running time (Listing 1.5). Recursive calls are included in the flat profile, but cyclic calls are not. The call graph (not displayed here) gives the rest of the information. To get the call graph, run the demo manually.

Listing 1.5: GPROF flat profile.

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 37.56      0.03     0.03        7     4.29     4.29  list_push_back(int, std::list<int
    , std::allocator<int> >&)
 25.04      0.05     0.02       56     0.36     0.36  new_integer(int)
 25.04      0.07     0.02        1    20.03    80.13  parent(int)
 12.52      0.08     0.01        7     1.43     3.93  file_io(int)
  0.00      0.08     0.00        1     0.00     0.00  global constructors keyed to main
```

1.4.2 CALLGRIND

CALLGRIND is a runtime-instrumentation profiler that uses the VALGRIND framework for its cache simulation and call graph generation. KCACHEGRIND is a GUI for browsing profiling results. The latest revisions of CALLGRIND and VALGRIND date back to 2005 (Weidendorfer, 2005b).

CALLGRIND is invoked using `valgrind --tool=callgrind` and produces measurement files named `callgrind.out.<pid>`. KCACHEGRIND looks for these files automatically and displays the results.

Because VALGRIND controls many aspects of a program's execution, CALLGRIND has an important overhead (Listing 1.6).

Listing 1.6: CALLGRIND demo timing.

```
..cbs/_build/demo/comparison $ time valgrind --tool=callgrind ./demo-
    generic 7
[...]
valgrind --tool=callgrind ./demo-generic 7  225.03s user 2.13s system
     99% cpu 3:47.88 total
```

KCACHEGRIND is a powerful and complete GUI to visualize flat profile and call graph from CALLGRIND results.

Since all context changes are listed with CALLGRIND, the flat profile and call graph are cluttered with hard to read information. Information can be sorted and selected to some extend, but remain dense.

Note that KCACHEGRIND can be used to visualize output from other profilers, provided they follow the CALLTREEE file format (Weidendorfer, 2005a).

1.4.3 CBS

CBS is a C++ Benchmarking Suite developed for VAUCANSON that provides an instrumentation profiler. Profiling is made by instrumenting C++ code with calls to a separate library, LIBBENCH, that does the measuring. CBS was created in 2009.

Because profiled symbols are defined by the programmer, CBS has little overhead as long as the instrumentation remains light (Listing 1.7).

Listing 1.7: CBS demo timing.

```
..cbs/_build/demo/comparison $ time ./demo-cbs 7
[...]
./demo-cbs 7  2.44s user 0.77s system 99% cpu 3.214 total
```

Notice that the CBS demo takes even less system time (in bold) than the generic demo. This appears consistently when executions are repeated. Compiler optimizations might explain the odd behavior. If nothing else, this shows that instrumentation does have an effect on the instrumented program.

CBS's flat profile is similar to GPROF's (Listing 1.8).

Listing 1.8: CBS flat profile.

```
Charge  id:          <name>      total     self      calls   self avg. total avg.
100.0%   0:         _program     3.22s    3.22s         1      3.22s       3.22s
 19.7%   1:         parent()     3.22s    0.64s         1      0.64s       3.22s
 30.1%   3:        file_io()   (C:   0)   0.97s        56    17.36ms     (C:   0)
 26.6%   2:    new_integer()   (C:   0)   0.86s        56    15.29ms     (C:   0)
 23.6%   4: list_push_back()     0.76s    0.76s        56    13.57ms     13.57ms
```

CBS provides flat profile and call graph in several formats and several levels of verbosity, according to the developer's need.

Note that CBS's primary use is to generate benchmarks for large C++ libraries such as VAUCANSON. Its use as an instrumenting profiler applies well to such libraries. The next section lists alternatives that may be more suitable for different project needs.

1.4.4 List of profilers

There exists many profiling tools. Below is an arbitrary list put together from various sources:

Sampling profilers

- GPROF *(Unix-like systems, no GUI, GPL)*
- AMD CodeAnalyst *(AMD processors, Windows, Linux, GUI included, free)*
- Intel VTune *(Intel processors, Windows, Linux, GUI included, commercial)*
- Shark *(MAC OS X, part of Apple Developer Tools)*

- DTrace *(Unix-like systems, GUI on MAC OS X only, CDDL)*
- OProfile *(Linux, no GUI, GPL)*
- memtime *(Linux, Solaris, no GUI, free)*

Instrumenting profilers

- CALLGRIND *(Unix-like systems, GUI available, GPL)*
- GPROF *(Unix-like systems, no GUI, GPL)*
- CBS *(Linux, no GUI (some visual output), C++, free)*
- Visual Quantify *(Intel processors, Windows, commercial)*
- Tuning and Analysis Utilities *(GUI included, multi-language, free)*

Chapter 2

CBS structure & implementation

This chapter relates to the implementation of CBS as of May 2009. File names used in this chapter refer to their relative location in the CBS archive (D'Halluin, 2009a).

2.1 Description

This section describes CBS from a user perspective.

CBS refers to a set of C++ benchmarking tools. Its main purpose is to benchmark all the components of a large C++ library such as VAUCANSON. This provides a *performance trace* that can be compared between versions of the library.

In VAUCANSON, CBS is also used as a profiling tool to estimate the cost of each segment in a chain of operations. It showed that in VAUCANSON's set of command-line tools, TAF-KIT, most of the resources are spent on I/O operations. The development of a result caching system improved performance at the design level.

CBS can also be used within algorithms to locate and optimize bottlenecks at the source code level. Using it on one of VAUCANSON's most important algorithms led to a 40% performance gain after less than an hour of work (see subsection 3.3.1). Profiling is made by manual instrumentation, which may not suit all projects but applies well to VAUCANSON.

CBS is meant to be free, comprehensive, and simple. It currently works on Unix-like systems only. Mac compatibility has not been tested, but is an objective. A Windows version may also be developed.

2.1.1 Features

CBS is made of several components that work together:

- **XML archiving**: Benchmarks and statistics are exported in a simple XML format.

- **Libbench**, a C++ profiling front-end relying on code instrumentation. The library is accessed from C++ code directly or using macros. It performs measures at run-time and exports results in text, DOT, or XML formats. LIBBENCH is the main CBS component.
- **Plot helpers**, scripts that extract data from a set of XML files in order to generate plots and figures. Typically, extracted data is gathered in a file that can be read by plotting programs such as GNUPLOT.

CBS is designed to be simple to understand and to manipulate. Additional features can be written effortlessly, such as a script front-end to generate benchmarks without the need for instrumentation.

2.1.2 LIBBENCH

LIBBENCH is the main CBS component. It relies on code instrumentation.

Developers choose which parts of the code to instrument. While instrumenting is not automatic and takes time, this design reduces clutter in the output. It also helps reduce the cost of measures, as proper instrumentation remains meaningful while having very little overhead.

When only main statistics are needed (global time and memory consumption), the instrumentation is limited to a few lines of code giving a name and description to the benchmark, and listing the desired output formats.

LIBBENCH is composed of two modules, TIMER and MEMPLOT, used together during profiling. Each module can be enabled or disabled at will without changing the instrumentation.

TIMER

TIMER is the LIBBENCH module responsible for building a call graph of the profiled program. It relies on the definition of *tasks* within the code, for which time statistics are gathered.

TIMER was previously used by itself in VAUCANSON and refered to as the *global timer* system (Bigaignon, 2006). It was rewritten in 2007 to improve its output and to generate a call graph.

During a program execution, data is first gathered in a raw format, then processed to detect cycles and build the call graph. TIMER can have different degrees of output verbosity showing more or less statistics about number of calls and time taken, including total time for a task and its children, time taken by the task itself, and average time per call. Time is broken down into Wall time, CPU time, user time, and system time.

MEMPLOT

MEMPLOT is the LIBBENCH module responsible for measuring memory usage in the profiled program. It relies on the definitions of *plot points* where memory usage is measured. Plot points are linked to the task being executed in TIMER so that memory usage can be displayed in relation to the tasks.

As of May 2009, MEMPLOT exports only in XML format. Global memory usage is however available in all formats of the CBS benchmark summary.

2.1.3 Usage

Using CBS is a 4-step process:

1. **Get CBS**: CBS is available as an autoconf build that can be installed on the target system or directly integrated in another autoconf project. To get CBS, visit its temporary homepage (D'Halluin, 2009a).
2. **Instrument code**: use the macros defined in the `<cbs/bench/bench-macros.hh>` header to instrument code with calls to measurement functions. The macro interface is included in Appendix A, along with a short example. Link with LIBBENCH (`-lbench` with gcc). Documentation and examples can be found in `README` and `demo/`. For a list of best practices, see subsection 2.3.4.
3. **Run benchmarks**: running the program as usual will do the measures and output the files specified in the instrumented code.
4. **Extract results**: use the `plot.pl` script in `bin/` to extract results from XML files in GNUPLOT format. Note that as of May 2009, this script is not installed by default. Some GNUPLOT usage examples are located in `demo/timer-bench`.

2.2 Output

This section lists the three output modes provided by CBS (XML, DOT, and text. It also describes how to interpret them.

For each mode, there are three output elements:

- **Benchmark summary**: a short summary of the benchmark with a the default statistics and user-defined parameters and results. Available in text and XML modes.
- **Callgraph**: the callgraph output from TIMER. Available in text, XML and DOT modes. Includes the flat profile in text mode.
- **Memplot**: the memplot output from MEMPLOT. Available only in XML mode.

Several verbosity levels are available. The verbosity level determines how much information is shown in the output. For a list of verbosity levels and how to use them, refer to the example in Appendix A and to the file `<cbs/bench/bench.hh>`.

2.2.1 Text

CBS's text output is simple and straightforward. In the summary output, all times are in ms, and memory is in bytes (divide by 1024 twice to have the value in MB). Listing 2.1 shows an example of text summary.

In the TIMER output, the flat profile is easy to understand (Listing 2.2). Cycles are shown with `(C: x)`, where `x` is the cycle id. Note that `_program` is the default task representing the whole program. More information on the TIMER text ouput is available in `<cbs/bench/timer.hh>`.

Listing 2.1: Summary text output.

```
Profiler comparison

[Description:]
A simple program that consumes time and memory
The parameter n defines the program complexity,
i.e. the time and memory taken.

[Infos:]
  * Date: Mon May  4 14:43:22 2009

[Parameters:]
  * n: 20

[Results:]
  * memory peak: 686280704
  * relative memory usage: 671924224
  * time: 67435
  * time (system): 15796
  * time (user): 51639
  * time (wall): 67482
```

Listing 2.2: Flat profile text output.

```
[Task list:]

Charge  id:         <name>       total     self      calls    self avg.  total avg.
100.0%   0:          _program   67.44s    67.44s         1    67.44s      67.44s
 30.7%   3:         file_io()   (C:  0)   20.68s       420    49.23ms     (C:  0)
 25.5%   2:     new_integer()   (C:  0)   17.21s       420    40.96ms     (C:  0)
 23.5%   4:  list_push_back()   15.84s    15.84s       420    37.73ms     37.73ms
 20.3%   1:          parent()   67.44s    13.71s         1    13.71s      67.44s
```

2.2.2 DOT

CBS's DOT output is specific to TIMER. It gives a visual representation of the call graph. Typically, the call graph output will be processed with dot, e.g.:

```
dot -Tpng bench_callgraph.dot -o bench_callgraph.png
```

Figure 2.3 shows an example of processed output.

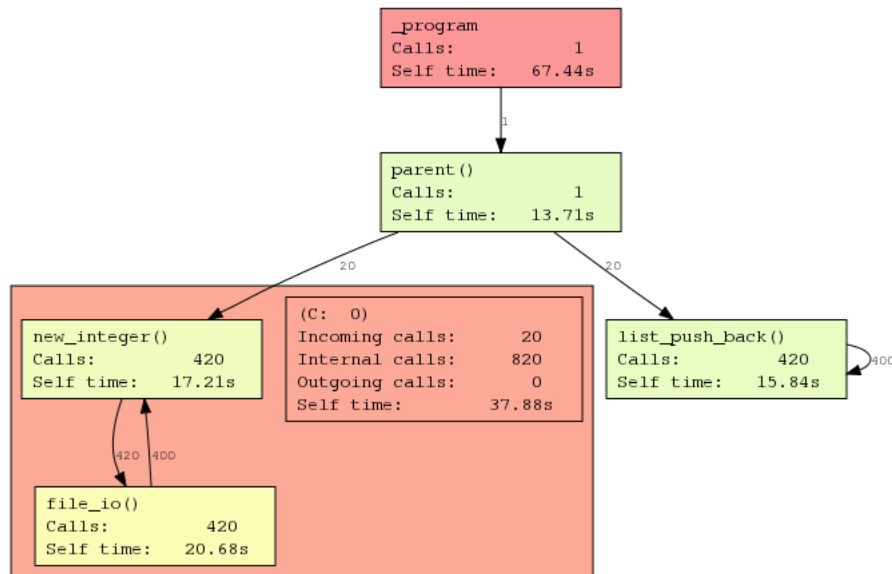


Figure 2.3: DOT output example.

2.2.3 XML

For each program run, CBS can store all the profiling information in a single XML file. This file is meant to be processed later, either by itself or together with other files resulting for the same benchmark run with different parameters.

The format is not yet set in stone and will be subject to changes. Each input element has its own tag, either `<bench>`, `<timer>`, or `<memplot>`. A simple example is provided in Listing 2.4. It does not show the timer and memplot outputs since they are more complex. A draft of XML format definition is available in `cbs.xsd`.

Listing 2.4: Main section of XML output.

```
<?xml version="1.0" encoding="UTF-8"?>
<bench>
  <name>Profiler comparison</name>
  <date>Mon May  4 14:43:22 2009</date>
  <time>1241441002</time>
  <description>A simple program that consumes time and memory
The parameter n defines the program complexity,
i.e. the time and memory taken.</description>
  <parameters>
    <parameter name="n" value="20" />
  </parameters>
  <results>
    <result name="memory␣peak" value="686280704" />
    <result name="relative␣memory␣usage" value="671924224" />
    <result name="time" value="67435" />
    <result name="time␣(system)" value="15796" />
    <result name="time␣(user)" value="51639" />
    <result name="time␣(wall)" value="67482" />
  </results>
</bench>
```

2.3 Implementation

This section details the inner workings of CBS and its performance.

2.3.1 Archive structure

Listing 2.5 describes the structure of the CBS archive, along with its most important files.

Listing 2.5: CBS archive structure.

```
cbs                                 Main directory
cbs/stats.sh                          Stats generator
cbs/ChangeLog                         Changelog
cbs/bootstrap                         Bootstrap script
cbs/README*                           Help files
cbs/cbs.xsd                           XML Format definition
cbs/demo                            Demos and examples
cbs/bin                             Scripts and binaries
cbs/bin/plot.pl                       Result extractor
cbs/include                         Headers
cbs/include/cbs/bench               Libbench headers
cbs/include/cbs/bench/timer*          Timer headers
cbs/include/cbs/bench/bench*          Libbench interface
cbs/include/cbs/bench/internal      Internal headers
cbs/include/cbs/bench/memplot*        Memplot headers
cbs/lib                             Source code files
cbs/lib/memplot*                      Memplot source code
cbs/lib/bench*                        Libbench source code
cbs/lib/timer*                        Timer source code
```

2.3.2 Measures

CBS relies on standard Unix functions, system calls, and files to measure performance. Time-related measures occur in `lib/timer_gathering.cc` while memory-related measures occur in `lib/memplot.cc`.

CPU time

CPU time is measured using the `getrusage()` system call. It provides user time and system time in microseconds. CPU time is the sum of user time and system time. In practice, it has a 4ms precision on the tested systems (Linux distributions Kubuntu & Debian).

Since results for intervals of less than 4ms are either 0ms or 4ms and are properly distributed, the average time taken by a task can be more precise than 4ms if that task is repeated many times.

Wall time

Wall time is measured using the `gettimeofday()` system call. This is precise up to the millisecond on the tested systems (Linux distributions Kubuntu & Debian).

Memory usage

Memory usage is obtained by reading the content of `/proc/self/stat`. This file contains a number of statistics about the running process from which it is accessed, including the allocated virtual memory and RSS.

Since the file must be read and parsed, memory usage measures cost more time than time-related measures (see CBS benchmarks in Appendix B). MEMPLOT was designed as a separate module from TIMER in order not to impose an unwanted cost on the time-related measures.

Note that the virtual memory usage can also be deduced from the value of the break, obtained using `sbrk(0)`. This is much faster than parsing a file, but does not give the RSS.

2.3.3 Building the call graph

The call graph is built by the TIMER module (see subsection 2.1.2). Building the call graph is a 2-step process:

- **Data collection**: while the program is executing its main operation, time measures are kept as little obtrusive as possible.
- **Call graph computation**: once the program has finished its main operation, the call graph is built and additional values (such as averages) are computed.

Data collection

Running task instances are kept in a *call stack*. Total time consumption for each task is stored in a lightweight custom graph structure.

Once a task instance is completed, its time consumption is added to the total time for the task given its parent, and deducted from its parent's own time consumption (this happens in the custom graph structure).

For each task instance, the measurement cost amounts to:

- one value query in a `std::map` to find the corresponding task structure.
- two system calls to `getrusage()`.
- two system calls to `gettimeofday()`.
- several direct structure access.

- some additions and substractions.

In the resulting call graph, the measurement cost for a given task is split: half will appear in the actual task, and half in its caller.

Note that the map value query can be avoided by calling tasks with their id number instead of their name. This is, however, impractical.

Call graph computation

The call graph is stored in a `boost::adjacency_list` structure. The final call graph contains more data than the ligthweight graph used for measuring. Additional data include average time per call and charge (percentage of the total time), both including and excluding the task's children.

Cycles in task execution correspond to strongly-connected components in the call graph. Some data is meaningless for cycles, e.g. the total time spend in a task and its children when they both call each other. In those cases, a separate entry is available for the whole cycle.

2.3.4 Performance

Complete CBS benchmarks are available in Appendix B.

CBS is designed to have as little an impact on performance as possible during the execution of a program's main task. Computing the call graph is done once the main task is complete and requires an additional cost that does not appear in the measures.

In order to reduce the measurement cost and improve the clarity of the results, instrumentation should follow a few simple guidelines:

- Use variable (context-dependent) task names in callers, not callees.
- Do not instrument very small tasks (such as small memory allocation).
- Instrument outside of loops when applicable.

Over-instrumentation can have a significant impact on performance. Figure 2.6 shows the influence of instrumentation when CBS is used to locate a bottleneck in an actual VAUCANSON algorithm. First, light instrumentation is used outside loops and the bottleneck is located within a 30-line portion of code (in a 900-line file). Then, two additional instrumentation points are inserted within a loop repeated 3 million times. This results in the profiling taking more that twice as long as the algorithm itself.

The impact of instrumentation can be estimated using the benchmarks in Appendix B and the number of calls displayed on the call graph. In the future, this could be done automatically by CBS.

Instrumentation	None	Light	Heavy
Area to optimize (lines)	900	30	10
Instrumentation points	0	11	13
Number of measures	0	30000	3330000
Execution time (ms)	4216	4422	15970
Overhead	0%	5%	278%

Figure 2.6: Influence of over-instrumentation.

2.3.5 Licensing

CBS is a fresh project and the license type has yet to be chosen. To be used in VAUCANSON, CBS's license must be compatible with the GPL. Several alternatives can be considered:

- GPL: widely used free software license, derived works must be released under the GPL.
- BSD: widely used permissive free software license, derived works can be released under a different license.
- CeCILL: French free software license similar to the GPL.
- EUPL: free software license for the European Union.

The most straightforward option would be to use the GPL license, but this forbids the use of CBS in commercial projects. BSD is the second most likely candidate, as CeCILL and EUPL are still rarely used. This matter is to be settled before September 2009.

Chapter 3

Integration in VAUCANSON

This chapter relates to the use of CBS in VAUCANSON. The integration of CBS was made in the `exp/libbench` branch of VAUCANSON's git repository and is to be included in VAUCANSON releases past 1.3. File names used in this chapter refer to their relative location in the VAUCANSON archive (VAUCANSON Group, 2008).

3.1 Benchmarking VAUCANSON

This section describes how CBS is used in the benchmarking system to generate a trace that can be compared between versions.

In order to compare VAUCANSON to other manipulation libraries in terms of performance (Lazzara, 2006), a benchmarking system was written by Michaël Cadilhac in 2005. It now serves a multiple purpose:

- Give an overview of VAUCANSON's performance across versions.
- Compare VAUCANSON to its competitors (including OPENFST).
- Compare data structure implementations (LISTG and BMIG).
- Measure the impact of structural changes in the library.

Benchmarking is done by executing `make bench` in a build directory. This compiles a set of short programs located in `src/bench/<program>`. Each program measures the performance of a specific part of the library.

Benchmarking programs work in 4 steps:

1. **Generate a test automaton**, e.g. the nth product of a simple automaton that gives a worst-case input for the algorithm considered.

2. **Save the test automaton**, e.g. in OPENFST format, so that it can be used to run the same test on other automaton manipulation libraries.
3. **Run the test** and measure its performance.
4. **Ouput performance results**.

Typically, a program works on one version of a function or algorithm and takes an integer parameter (n) that defines the complexity of the input automaton.

Note than generating the test automaton usually takes longer than actually running the test. This is nothing more than a practical consideration, since it is irrelevant to the performance results. However, considering how memory allocation works (see 1.2 — *memory usage*), the memory taken to generate a test automaton does have an influence on the test results. A way to work around this problem would be to generate the test automaton in a separate process. This has been given thoughts, but is not yet implemented.

In the original version of the benchmarking system, each program printed the time (wall time) taken to run the test. Integrating CBS to VAUCANSON's benchmarking system meant changing the measurement instrumentation to use CBS. Using CBS improved the quality of the output and allowed results to be archived into files.

For each program, 3 result files are generated, each corresponding to a CBS output type:

- `bench_<program>_<n>.xml`: fully verbose XML output.
- `bench_<program>_<n>.dot`: normally verbose call graph DOT output.
- `bench_<program>_<n>.out`: fully verbose text output.

3.1.1 Affected files

Only minor changes were necessary to integrate CBS to the benchmarking system:

- Replace macros in program source files `src/bench/<dir>/*_bench.hh` with CBS equivalent.
- List the desired output in `src/bench/common/bench_macros.hh`.
- Remove obsolete measurement tools (`include/vaucanson/tools/bencher*`).
- Tweak makefiles to compile with CBS.

These changes are detailed in the git patch `0f1e404a2f` and in the corresponding entry in `ChangeLog`, *Equip bench with CBS*.

3.1.2 Interpreting results

The full VAUCANSON performance trace comparing LISTG and BMIG is available in Appendix C and in a separate archive, in raw format (D'Halluin, 2009b).

Figures can be easily generated from benchmark runs using the `plot.pl` script in `cbs/bin` and GNUPLOT. Listing 3.1 shows a typical example. A complete set of scripts is provided in `cbs/demo/vcsn`.

Listing 3.1: VAUCANSON benchmark plotting example.

```
../vaucanson $ cbs/bin/plot.pl -a _build/src/bench/determinization/
    bench_determinize_*.xml > determinize.data
../vaucanson $ echo "set term postscript eps enhanced" > determinize.
    plot
../vaucanson $ echo "set output 'determinize.eps'" >> determinize.
    plot
../vaucanson $ echo "plot 'determinize.data' using 1:3 smooth unique"
     >> determinize.plot
../vaucanson $ gnuplot determinize.plot
../vaucanson $ display determinize.eps
```

3.2 TAF-KIT

TAF-KIT is a set of command-line tools used to access algorithms from the VAUCANSON library without writing any code. This section describes how CBS is used in TAF-KIT and how to profile command executions.

3.2.1 Affected files

Before relying on CBS, TAF-KIT used the *global timer* system that became CBS's TIMER module. Because the two systems are very similar, integrating CBS to TAF-KIT was straightforward:

- Replace global timer macros with CBS equivalent in algorithm definitions (`include/vaucanson/algorithms/*.hxx`).
- Remove global timer in `taf-kit/src/main.cc`.
- Tweak makefiles to compile with CBS.

These changes are detailed in the git patch `461842172b` and in the corresponding entry in `ChangeLog`, *Replace timer and global_timer with cbs*.

Note that using CBS macros in algorithm definitions allows both TAF-KIT and the benchmarking suite to output the same call graph for a given algorithm. The call graph was previously only available with TAF-KIT.

3.2.2 Measuring command execution

TAF-KIT is invoked with a set of options, a command, and input arguments. It prints results on the standard output. In order not to interfere with the command results, all profiling reports go the error output. TAF-KIT options can trigger the output of CBS reports in all three formats and all verbosity levels. A list of these options is available by invoking TAF-KIT with the `--help` parameter. Listing 3.2 shows a few examples.

Listing 3.2: TAF-KIT command profiling.

```
../_build/taf-kit/tests $ ./vcsn-char-b --report-time=1 determinize
    ../../data/automata/char-b/ladybird-6.xml > result.xml 2> flat-
    profile.out
../_build/taf-kit/tests $ ./vcsn-char-b --export-time-xml determinize
     ../../data/automata/char-b/ladybird-6.xml > result.xml 2> profile
    .xml
../_build/taf-kit/tests $ ./vcsn-char-b --export-time-dot=0
    determinize ../../data/automata/char-b/ladybird-6.xml > result.xml
     2> callgraph.dot
../_build/taf-kit/tests $ dot -Tpng callgraph.dot -o callgraph.png
```

Figure 3.3 shows the resulting call graph. *Vertex color represents the task charge. Light/blue vertices have a small charge, while dark/red vertices have a heavy charge. The "_program" task always has a 100% charge.* Notice that most of the execution time of TAF-KIT is spent executing I/O operations.

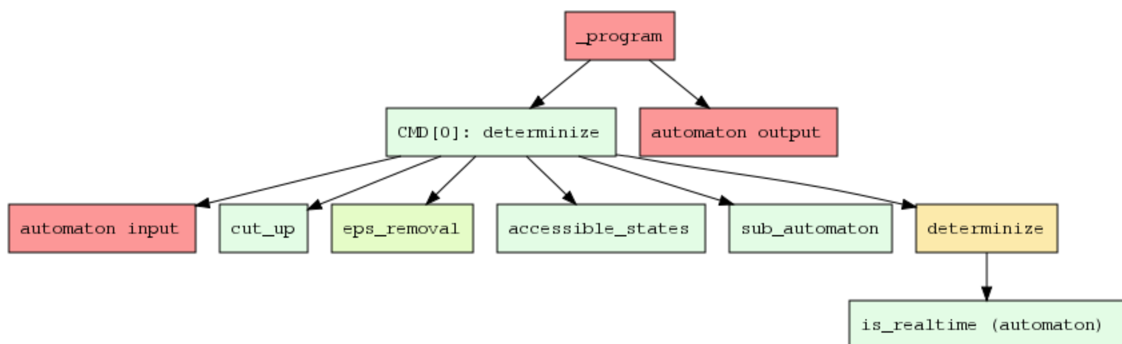


Figure 3.3: TAF-KIT minimal call graph.

3.3 Improving performance

This section describes how to use CBS to optimize algorithms and lists some of the results obtained.

3.3.1 Optimizing algorithms

CBS can be used as an instrumenting profiler to locate bottlenecks in the VAUCANSON code. This process involves 4 steps:

Select an algorithm to optimize

Optimization should be done on the algorithms that are the most used in practice. The benchmarking system gives a good starting list. Other examples can be found in specialized literature (Sakarovitch, 2003). Once the algorithm has been selected, list a set of representative input automata.

Write a test program

Given an algorithm and a set of input automata, write a simple test program. Refer to the benchmarking system examples in `src/bench/`, and to the VAUCANSON documentation in `doc/`.

Instrument the algorithm code

Insert measurement macros in several places in the algorithm code. Algorithms code is usually located in `include/vaucanson/algorithms/*.hxx`. Running the test program should show which the part of the code could use deeper instrumentation.

Optimize selectively

When measures start having a noticeable effect on performance (compared to the non-instrumented run, or when more that a million measures are made), focus on optimizing the most demanding tasks listed in the call graph.

3.3.2 Example for quotient()

The method described in subsection 3.3.1 was used on the algorithm for boolean automaton quotient to illustrate the use of CBS.

Given:

- The test program located in `src/bench/quotient/quotient_bool_bench.hh`.
- The input automaton that recognizes words containing b on the ab alphabet (Figure 3.4).
- An input parameter n.

Performance tests were made on the quotient of the nth product of the input automaton for successive values of n. The test automata tend to have a very large number of transitions (up to a million), and a large number of states (up to 10 000). Selective code optimization led to a 40% performance boost on the test cases.

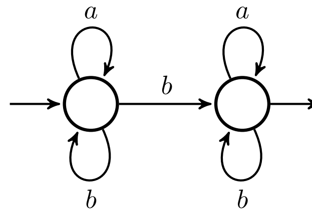


Figure 3.4: Base automaton for quotient() benchmark.

Instrumenting `include/vaucanson/algorithms/minimization_hopcroft.hxx` gave the call graph shown in Figure 2.6. This call graph was used to isolate a 30-line section of code responsible for 90% of the execution time. The original code section is listed in Listing 3.5.

Notice the two similar statements on lines 11 and 12, `input_.series_of(*t)`. Provided that they have no side effect, a straightforward optimization would be to make only one call, store its result in a new variable and use it in place of the two statements. The optimization was indeed possible and led to a 20% performance boost.

VAUCANSON developers will quickly realize that two other statements (`monoid_elt_t w` and `input_.series().semiring().wzero_`) can be computed before the double loop, which is executed 3 million times for a parameter $n = 13$. The resulting code is included in Listing 3.6. *It runs up to 40% faster than the original code on the test cases*. Figure 3.7 show the benchmark results for time and memory, before optimization (dark/red) and after optimization (light/green).

Note that this optimization did not require the knowledge of any complex C++ tricks, nor did it make the code any less readable. It is a simple case of straightforward code optimization that proved very efficient because it was made on a bottleneck.

Listing 3.5: quotient() code excerpt, before optimization.

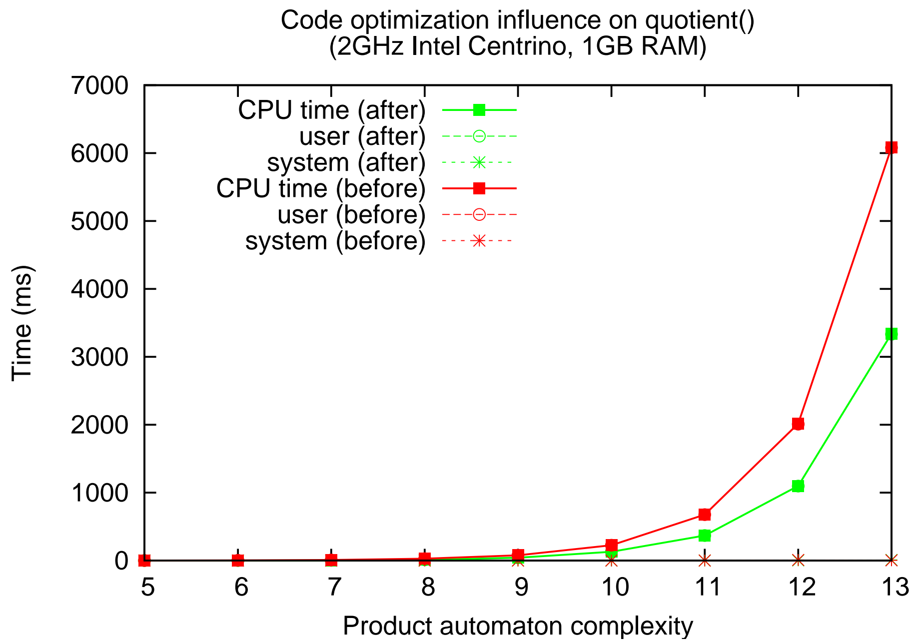
```
1  bool compute_going_in_states (partition_t& p, letter_t a)
2  {
3    for_all_(going_in_t, s, going_in_)
4      *s = false;
5
6    for_all_(partition_t, s, p)
7    {
8      for (rdelta_iterator t(input_.value(), *s); ! t.done(); t.next())
9      {
10       // Some code optimization is possible here
11       monoid_elt_t w(input_.series_of(*t).structure().monoid(), a);
12       if (input_.series_of(*t).get(w) != input_.series().semiring().
             wzero_)
13       {
14         hstate_t s = input_.src_of(*t);
15         if (!going_in_[s])
16         {
17           going_in_[s] = true;
18           const unsigned i = class_[s];
19           if (count_for_[i] == 0)
20             met_class_.push_back(i);
21           count_for_[i]++;
22         }
23       }
24     }
25   }
26   return !met_class_.empty();
27 }
```

Listing 3.6: quotient() code excerpt, after optimization.

```
bool compute_going_in_states (partition_t& p, letter_t a)
{
  for_all_(going_in_t, s, going_in_)
    *s = false;

  // Compute as much as possible outside loops.
  semiring_elt_t weight_zero = input_.series().semiring().wzero_;
  monoid_elt_t w(input_.series().monoid(), a);

  for_all_(partition_t, s, p)
  {
    for (rdelta_iterator t(input_.value(), *s); ! t.done(); t.next())
    {
      // Optimization lead to a 40% performance boost on test cases.
      if (input_.series_of(*t).get(w) != weight_zero)
      {
        hstate_t s = input_.src_of(*t);
        if (!going_in_[s])
        {
          going_in_[s] = true;
          const unsigned i = class_[s];
          if (count_for_[i] == 0)
            met_class_.push_back(i);
          count_for_[i]++;
        }
      }
    }
  }
  return !met_class_.empty();
}
```



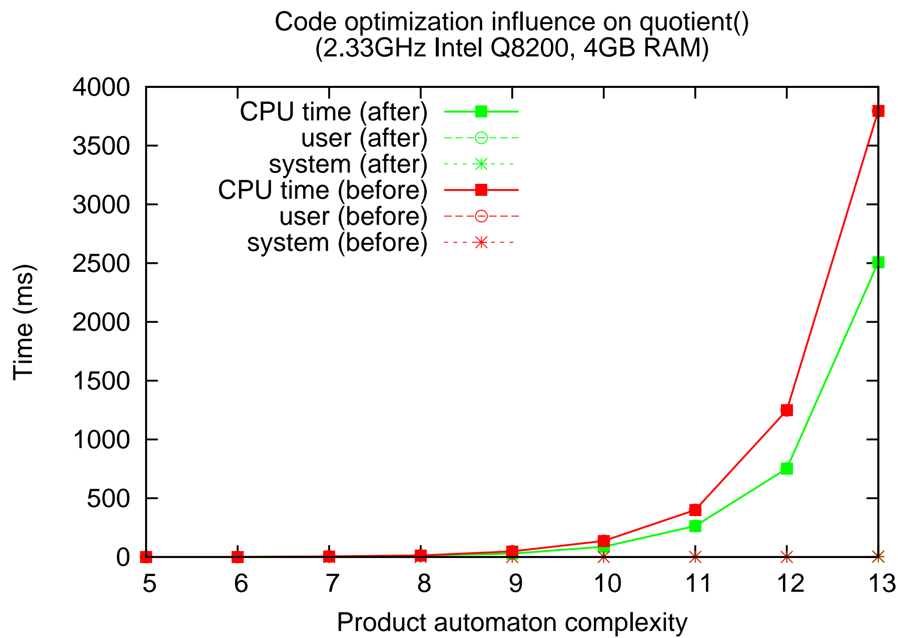


Figure 3.7: Quotient time optimization.

Conclusion & Perspectives

VAUCANSON is one of the two representative automaton manipulation libraries, but it suffers from major performance issues compared to its competitor, OPENFST.

These performance issues are rooted in the structure of the library: in order to remain generic, the structure of the library is fairly complex and leads to obscure code, which is difficult to optimize selectively.

There is ongoing work on the structure of the VAUCANSON library meant to make it clearer and more efficient. This will open new possibilities for the optimization of algorithms.

In order to make the most of the upcoming changes, two performance analysis tools are needed: a benchmarking suite and a C++ code profiler. CBS is a set of tools developed specifically for VAUCANSON that merges together the existing benchmarking and profiling systems in VAUCANSON.

CBS was used to compare two automaton data structures, LISTG and BMIG, showing a surprising loss of performance in BMIG compared to LISTG.

CBS was also used to profile and optimize an important algorithm of the VAUCANSON library, quotient, leading to a 40% performance boost on a test example.

This report listed other tools for performance analysis in large C++ projects. Many of these tools are widely used, but are not adapted to VAUCANSON. While it is still early to tell how useful CBS will be in improving the performance of VAUCANSON, it quickly showed promising results: selective optimization is made easy, and the impact of structural changes to the library can be measured efficiently.

Future works

CBS is still in the early stages of developement. This section lists several tasks that could be made in continuation of the work I did on CBS:

- Optimize VAUCANSON's algorithms.
- Expand CBS with benchmarking systems for other languages, such as shell script.
- Expand CBS with tools to display benchmark results.
- Automatically compare performance between versions of VAUCANSON.
- Add support for Mac & Windows to CBS.
- Turn CBS into a separate open-source project.

Appendix A

CBS macro interface

This appendix lists CBS equipment macros, extracted from `include/cbs/bench/bench_macros.hh` from the CBS git repository, revision `31b2e`. See the file for details. Below the interface is a short example of instrumented code.

Additional documentation and examples can be found in `README_LIBBENCH` and `demo/` in the CBS archive.

Listing A.1: CBS macro interface.

```
1  // General
2  #define BENCH_START(Name, Description)       bench::start(Name, Description)
3  #define BENCH_STOP()                         bench::stop()
4  #define BENCH_PARAMETER(Name, Value)         bench::parameter(Name, Value)
5  #define BENCH_RESULT(Name, Value)            bench::result(Name, Value)
6
7  // Timer
8  #define BENCH_TASK_START(Name)               bench::task_start(Name)
9  #define BENCH_TASK_STOP()                    bench::task_stop()
10 #define BENCH_TASK_SCOPED(Name)              bench::ScopedTask bench_task(Name)
11
12 // Memplot
13 #define BENCH_MEMPLOT(Description)           bench::memplot(Description)
14
15 // Export
16 #define BENCH_PRINT(Options)                 bench::print(Options)
17 #define BENCH_DUMP(Stream, Options)          bench::dump(Stream, Options)
18 #define BENCH_SAVE(Filename, Options)        bench::save(Filename, Options)
19
20 // Types
21 // Name, Description, Filename: std::string.
22 // Value: std::string, long, or double.
23 // Options: bench::Options.
24 // Stream: std::ostream.
```

Listing A.2: CBS instrumentation example.

```
#include <cbs/bench/bench_macros.hh>

void do_some_tests();
void work();

int main()
{
  BENCH_START("My␣benchmark", "My␣description");

  BENCH_PARAMETER("My␣parameter", "My␣value");

  do_some_tests();

  BENCH_RESULT("My␣parameter", "My␣result");

  BENCH_PRINT(bench::Options(bench::Options::VE_NORMAL,
                             bench::Options::FO_TEXT,
                             0));

  BENCH_SAVE("benchmark.xml", bench::Options());

  return 0;
}

void do_some_tests()
{
  // Only one scoped task per scope.
  BENCH_TASK_SCOPED("tests");

  BENCH_TASK_START("tests_main");

    {
      BENCH_TASK_SCOPED("tests_inner_scope");
      work();
      // tests_inner_scope ends here.
    }

  // All tasks started with BENCH_TASK_START
  // have to be stopped.
  BENCH_TASK_STOP(); // tests_main

  // tests ends here.
}

void work()
{
}
```

Appendix B

CBS Benchmarks

This appendix shows CBS satistics and benchmark results. The benchmarks were run on two systems:

- A 3-year-old laptop, 2GHz Intel Centrino, 1GB RAM, running Kubuntu 8.10 (32bits).
- A 6-month-old desktop computer, 2.33GHz Intel Core 2 Quad, 4GB RAM, running Kubuntu 9.04 (64 bits)

Figure B.2 shows the time taken to make a given number of measures on a recent desktop computer. Using this figure, one can estimate the overhead cost of the profiling given the total number of measures made.

Note that the two overlapping curves refer to the two methods of starting and stopping tasks: directly (start stop) or through a proxy object, in which start is called in the constructor and stop in the destructor (scoped). These two methods have similar performance.

To get accurate overhead estimates on a different system, run benchmarks directly. See `README_LIBBENCH` and `demo/` in the CBS archive (D'Halluin, 2009a).

Listing B.1 show miscellaneous statistics for the CBS archive.

Listing B.1: Miscellaneous CBS statistics.

```
CBS Statistics - Tue Jun  2 14:58:18 CEST 2009
------------------------------------------------------------
C++ source code files: 21
C++ source code lines (including comments): 5398
Script files & helpers: 45
Total script lines (including comments): 1208
README files: 5
Total README lines: 149
Git repository creation date: Mon Mar 23 20:04:06 2009 +0100
Git commits on current branch: 51
```

Note: the printed version of this report only shows a selection of representative figures. The full benchmark is available in the electronic version and separately on the CBS *homepage.*

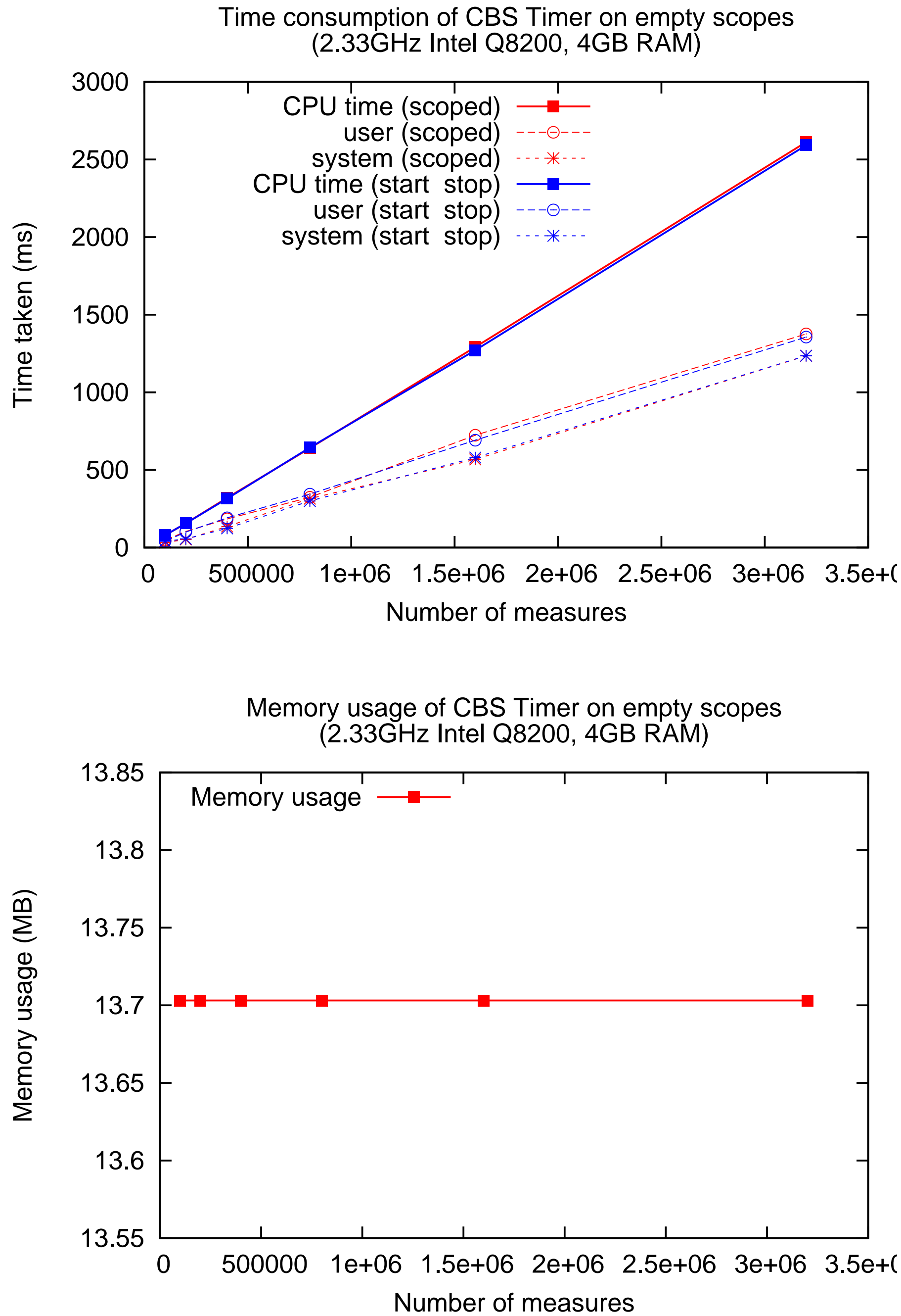


Figure B.2: Timer benchmark (empty loop) – 2.33GHz Intel Q8200, 4GB RAM.

Appendix C

VAUCANSON LISTG & BMIG benchmarks

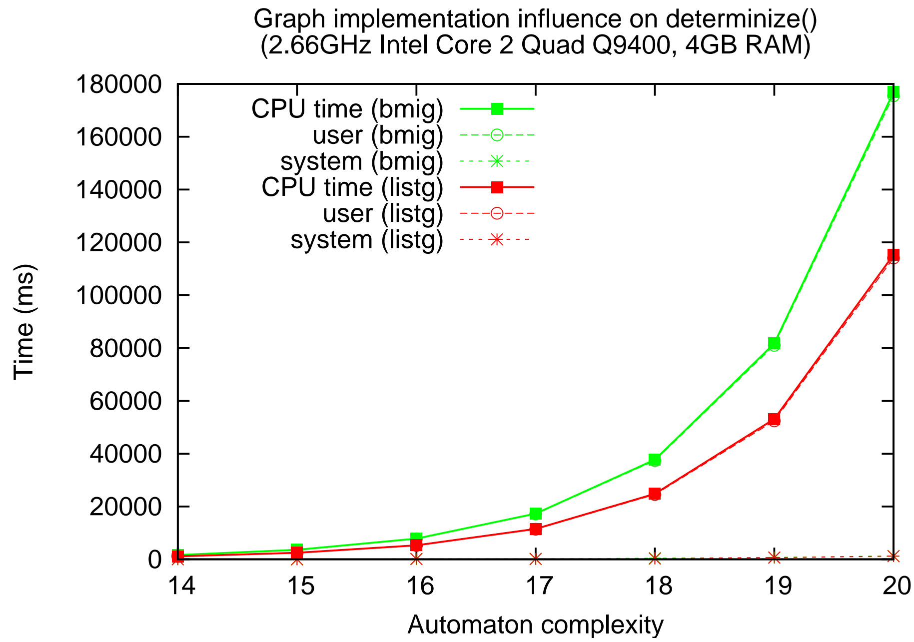
This appendix shows the performance trace of VAUCANSON with the LISTG and BMIG graph implementations.

The benchmarks were run on the LRDE machine `seattle` (2.66 Intel Core 2 Quad Q9400, 4GB RAM, 64bit Ubuntu).

Notice that:

- Figure C.1 shows that LISTG (dark/red) is 30% faster than BMIG (light/green) and requires slightly less memory to run the determinize algorithm on a common example. Similar results are observed for most tests, which supports the idea that LISTG is slightly more efficient than BMIG.
- On `composition-normalized()` (Figure C.2), BMIG has huge performance variations. This is currently unexplained. Repeating the tests gives a similar oddly-looking curve, but with the up-and-down spikes in different places. Time and memory consumption, however, remain linked.
- CBS does not seem to handle memory usage correctly past 4GB. This is shown in the tests where memory usage drops to zero after reaching 4GB. This will be investigated. Note that memory usage is affected by the generation of the input automaton, and thus may not be very reliable.

Note: the printed version of this report only shows a selection of representative figures. The full benchmark is available in the electronic version and separately on the CBS *homepage.*



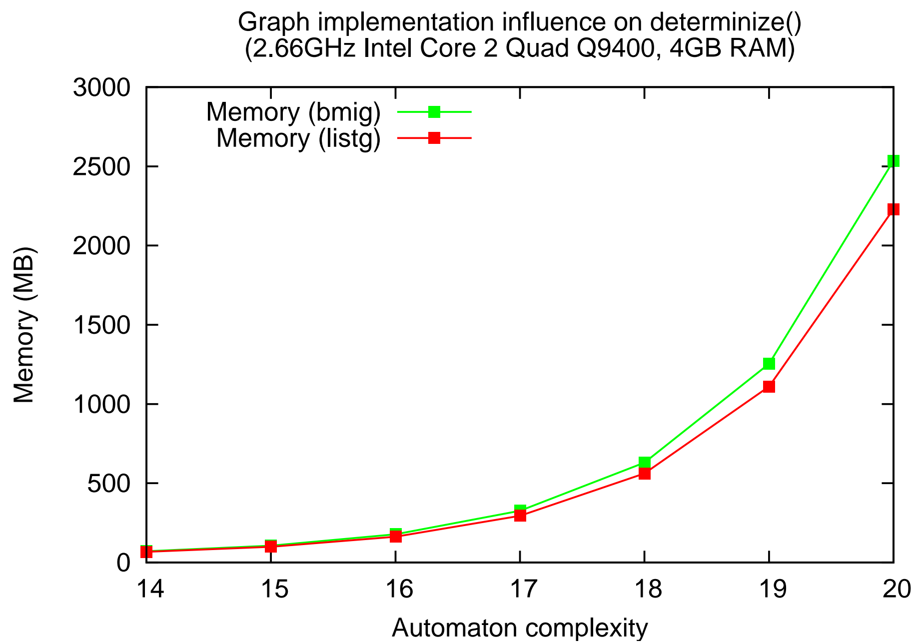
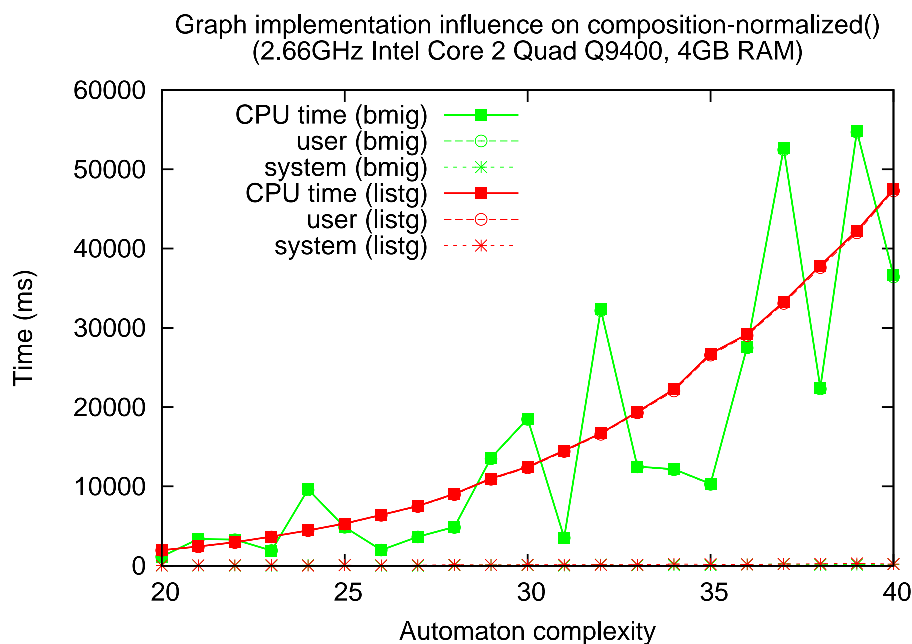


Figure C.1: Determinize benchmark – 2.66 GHz Intel Q9400, 1GB RAM.



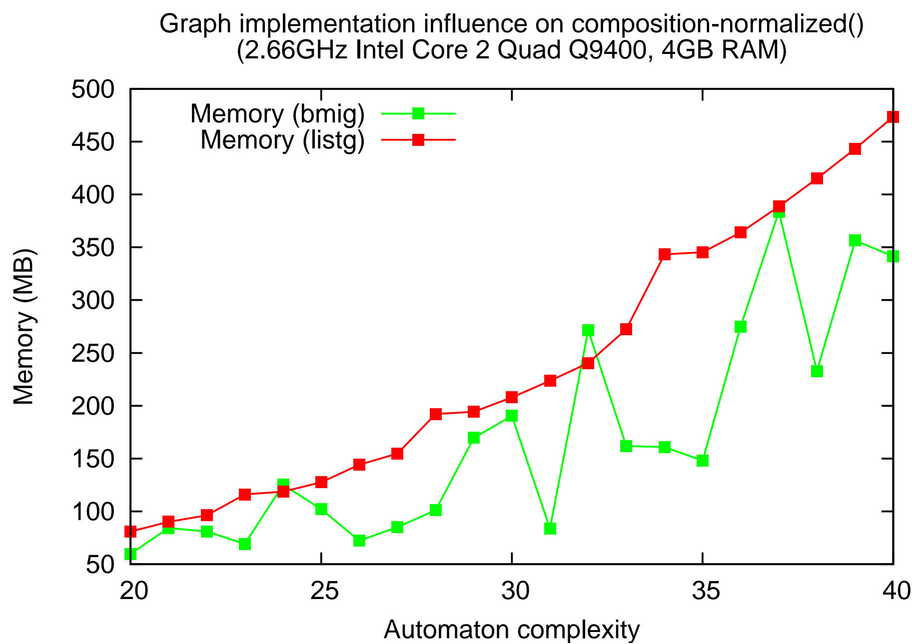


Figure C.2: Composition (normalized) benchmark – 2.66 GHz Intel Q9400, 1GB RAM.

Index

benchmarking, 29
bottleneck, 9

call graph, 9, 20, 26
    computation, 27
call stack, 26
Callgrind, 16
CBS, 17
    benchmarks, 42
    description, 19
    example, 40
    features, 19
    interface, 40
    libbench, 20
    output, 21
    performance, 27
    usage, 21
CPU time, 11, 25

DOT, 23

gnuplot, 21
gprof, 15

heisenbug, 14

instrumentation, 14
    profilers, 18

libbench, 20
    memplot, 20
    timer, 20

measurement, 26
memory usage, 12, 26
memplot, 20

optimization, 10, 33
overhead, 10

performance analysis, *see* profiling
performance analysis tools, *see* profilers
performance trace, 29
profilers, 9, 17
profiling, 9

quotient, 33

Resident Set Size, 12
RSS, 12

sampling, 12
    profilers, 17
swapping, 11
system calls, 25

taf-kit, 31
timer, 20

Vaucanson
    benchmarking, 29
    benchmarks, 44
    performance, 29
virtual memory, 12

wall time, 11, 26

XML, 23

List of Figures

1.3 Overhead estimates for different profilers 15

2.3 DOT output example. 23
2.6 Influence of over-instrumentation. 28

3.3 TAF-KIT minimal call graph. 32
3.4 Base automaton for quotient() benchmark. 34
3.7 Quotient time optimization. 37

B.2 Timer benchmark (empty loop) – 2.33GHz Intel Q8200, 4GB RAM. 43

C.1 Determinize benchmark – 2.66 GHz Intel Q9400, 1GB RAM. 45
C.2 Composition (normalized) benchmark – 2.66 GHz Intel Q9400, 1GB RAM. 46

List of Listings

1.1 Generic demo timing. 15
1.2 Predicted flat profile. 15
1.4 GPROF demo timing. 16
1.5 GPROF flat profile. 16
1.6 CALLGRIND demo timing. 16
1.7 CBS demo timing. 17
1.8 CBS flat profile. 17
2.1 Summary text output. 22
2.2 Flat profile text output. 22
2.4 Main section of XML output. 24
2.5 CBS archive structure. 25
3.1 VAUCANSON benchmark plotting example. 31
3.2 TAF-KIT command profiling. 32
3.5 quotient() code excerpt, before optimization. 35
3.6 quotient() code excerpt, after optimization. 36
A.1 CBS macro interface. 40
A.2 CBS instrumentation example. 41
B.1 Miscellaneous CBS statistics. 42

References

Allauzen, C., Riley, M., Schalkwyk, J., Skut, W., and Mohri, M. (2007). OpenFst: A general and efficient weighted finite-state transducer library. In *Proceedings of the Ninth International Conference on Implementation and Application of Automata, (CIAA 2007)*, volume 4783 of *Lecture Notes in Computer Science*, pages 11–23. Springer. http://www.openfst.org.

Bigaignon, R. (2006). Présentation du TAF-KIT. https://svn.lrde.epita.fr/svn/lrde-techreps/trunk/0701.

Desprès, N. (2005). Automatic performance monitoring tools. http://lrde.org/cgi-bin/twiki/view/Publications/20051116-Seminar-Despres-Transformers-BenchmarkTools-Report.

D'Halluin, F. (2009a). CBS archive homepage (temporary). http://lrde.epita.fr/~d-halluin.

D'Halluin, F. (2009b). VAUCANSON benchmark archive (temporary). http://lrde.epita.fr/~d-halluin.

Galtier, J. (2009). Remedial treatment for vaucanson: an enhanced automaton concept. http://lrde.org/cgi-bin/twiki/view/Publications/200905-Seminar-Galtier.

Hamelin, A. (2009). Property based class hierarchy of vaucanson's algebra module. http://lrde.org/cgi-bin/twiki/view/Publications/200906-Seminar-Hamelin.

Lazzara, G. (2006). Automata and performances. https://www.lrde.org/cgi-bin/twiki/view/Publications/200607-Seminar-Lazzara.

Lombardy, S., Régis-Gianas, Y., and Sakarovitch, J. (2004). Introducing Vaucanson. *Theoretical Computer Science*, 328:77–96.

Osier, J. (1993). gprof manual. Available on http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html.

Sakarovitch, J. (2003). *Éléments de théorie des automates*.

Shende, S., Malony, A. D., Cuny, J., Malony, S. S. A. D., Lindlan, K., Beckman, P., and Karmesin, S. (1998). Portable profiling and tracing for parallel, scientific applications using c++. In *In Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 134–145. ACM.

VAUCANSON Group (2008). VAUCANSON home page. http://vaucanson.lrde.epita.fr/.

Weidendorfer, J. (2005a). Calltree file format. http://kcachegrind.sourceforge.net/cgi-bin/show.cgi/KcacheGrindCalltreeFormat.

Weidendorfer, J. (2005b). KCachegrind homepage. http://kcachegrind.sourceforge.net/cgi-bin/show.cgi/KcacheGrindIndex.

Wikipedia (2009a). Optimization (computer science) — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Optimization_(computer_science)&oldid=287117813 [Online; accessed 2-May-2009].

Wikipedia (2009b). Performance tuning — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Performance_tuning&oldid=284963266 [Online; accessed 2-May-2009].

Wikipedia (2009c). Software performance analysis — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Software_performance_analysis&oldid=285420543 [Online; accessed 2-May-2009].