Automatic performance monitoring tools

Nicolas Desprès

Technical Report n^o0601, January 2006
revision 915

The LRDE has been mainly developing, for more than five years, two libraries dedicated to scientific computing in the field of images and automata processing. These two libraries aim at being both generic and efficient. The generic aspect has been ensured, since the beginning, by a good test suite to avoid functional regression. However, the performance of these libraries has decreased. As a solution, we have built some tools to automatically track the performances. This report introduces the tools set up to gather any kind of measures. It also describes the tools used to draw chart based on the data.

Le LRDE développe principalement, depuis plus de cinq années, deux bibliothèques pour le calcul scientifique pour le traitement d'images ou d'automates. Ces bibliothèques aspirent à être à la fois génériques et performantes. Au cours du développement, une bonne batterie de test a permit d'assurer le maintient de l'aspect générique. En revanche, les performances de ces bibliothèques ont diminué au fil des différentes modifications. Pour pallier ce problème, des outils de suivi automatique des performances ont été mis en place. Ce rapport présente tout d'abord les moyens mis en oeuvre pour rapporter n'importe quel type de mesure. Enfin, les moyens d'exploitation de ces données seront abordés.

Keywords
automatic, regression benchmark, performance analysis, visualization, database, data collection

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
nicolas.despres@lrde.epita.fr – http://www.lrde.epita.fr/

Copying this document

Copyright © 2006 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

1 Introduction **4**

2 Requirements **5**
2.1 Overview 5
2.2 Automatic data acquisition 5
2.3 Measurements gatherer 5
2.4 Unified results format 6
2.5 Results repository 6
2.6 Results analyzes/visualization interface 6
2.7 Platform-dependency 6

3 Ranch: measurement library **7**
3.1 Example 7
3.2 Implementation 9
3.2.1 Bencher 10
3.2.2 Input 10
3.2.3 Output 10
3.2.4 Dumper 11
3.3 Measurements incompatibility 12
3.4 Unified result format 13
3.5 Automatic data gatherer 13

4 Ranch: web interface **14**
4.1 Global architecture 14
4.2 The database structure 15
4.2.1 The projects table 15
4.2.2 The benches table 15
4.2.3 The inputs table 15
4.2.4 The outputs table 16
4.2.5 The systems table 16
4.3 Systems management 16
4.4 Visualization interface 16
4.5 Results 17

5 Conclusion **19**

6 Bibliography **20**

A Web application screen shot **21**

Chapter 1

Introduction

This technical report presents a set of tools used together to form a regression benchmark system. Such a system aims at preventing performance regression which may be introduced during a project development.

Between two major versions of a program, many small losses of performance may be introduced continuously by maintainers while they are developing the program. They often do not detect these small performance regressions because they do not run their benchmark suite for every patch they apply [1] or because their performance measurements are not accurate enough. The sum of all these small losses of performance may result in a significant performance regression. When maintainers detect an important regression of the efficiency of their program, hundreds of patches are already committed. Thus, they are unable to find when this regression happened, especially if it is the sum of several small regressions.

This problem can be avoided if the benchmark suite is run continuously while the program is modified. It is not rare that a developer team do tens of patches per day (sometimes more) on their project. Thus, it is very cumbersome to run the benchmark suite manually after every patch, specially if it takes a long time to run[2]. Moreover, the amount of benchmark results may increase quickly since the number of revisions of an average project is often around 500. Thus, a regression benchmark system must be fully automatic in order not to overload the whole development cycle.

This report describes the regression benchmark system developed in our laboratory: *Ranch*[3]. To address the issues introduced above, our approach relies on a continuous build system [Gournet (2005)] and provides a library to gather measurements inside the program and a visualization application for data consulting. After the description of the requirements of the system, we present the library written to gather measurements. Finally, the analysis/visualization tool is detailed.

[1] Numerous projects do not even have any benchmark suite.

[2] This often happens since the test suite must be run before the benchmark suite and that both suite may be larger and larger as the program grows.

[3] Ranch is an acronym for Regression BENCHmark

Chapter 2

Requirements

This chapter covers the specification of the requirements of an automatic regression benchmark system. First of all, we give a short description of all of them as an overview. Then, we detail each of them individually.

2.1 Overview

The regression benchmark framework must fulfill the following requirements [Courson et al. (2000); Kalibera (2004); Kalibera et al. (2004)]:

- must perform and collect the measurements results automatically,
- must provides features to easily probe programs,
- must provide a unified results format,
- must manage a result repository,
- must feature a user-friendly interface for results analysis and/or visualization,
- It must be platform-independent.

All these requirements are detailed in the following sections.

2.2 Automatic data acquisition

As mentioned in the introduction, we aim at measuring performances for every revision of a project, in order to detect performance regression as soon as possible. Because performance measurements may take a long time, and because it is common to commit changes more than ten times per day, the performance data acquisition for a given project is very time consuming and thus can't be performed manually. It is crucial that the entire benchmark process is performed automatically: from the program and benchmark environment installation and running, to the addition of the results into the repository.

2.3 Measurements gatherer

The regression benchmark system must provide tools to ease the task of the benchmark writer. The benchmark writer should only write code to stress the program but not code to get the measures. Thus, the regression benchmark system must come with a library to perform this task.

This library must be as non-intrusive as possible in the benchmark code. Plus, it must provide basic built-in measurements, such as elapsed user time and memory usage. Nevertheless, it has to be extensible by the end user since we cannot anticipate all its needs. It must also take care of measurements tools which may invalidate other measures which are performed at the same time. Finally, it must guaranty the validity of the result format.

2.4 Unified results format

A benchmark consists in making a comparison of two measurements. The comparison may be done against another contestant program or an older version of the same program. In order to do benchmark comparison, the result must be stored using the same format to avoid the use of converters. Moreover, we want to be able to benchmark sub parts of a benchmark. Thus, we need a result format that supports nested structures. However, it is out of the scope of our project to write a complex parser and a complex pretty printer. Thus, we need a format which is easy to read and write from the perspective of a script.

2.5 Results repository

The amount of collected data may increase quickly because we want to perform measurement for every revision. Thus, we need a strong storage system (e.g. not a regular file). Moreover, we need to be able to search and group benchmarks together when we analyze the data. These analyzes must be fast enough and support scalability.

2.6 Results analyzes/visualization interface

The results analyzes/visualization interface must provide an easy and user-friendly way to generate graphs and charts based on the collected data. The most important graph we need in a regression benchmark system is the one representing the performance evolution in respect to the revision number of the program. We also need to compare different programs: typically our program and its contestants.

2.7 Platform-dependency

The end-user may wish to see the difference of performance of its program from one architecture to another. Thus, the benchmark environment must be able to run on different architectures. This constraint is applied especially on the benchmark suite written by the project authors. Most of, the project under test is compatible with the architecture the benchmark suite is compatible too. The task of the regression benchmark system is only to run the benchmark suite and to collect the result.

Chapter 3

Ranch: measurement library

This chapter describes the measurement library developed as part of our regression benchmark system: *Ranch*. First of all, we give an example of its usage, in order to give an overview of our library from the user perspective. Then, we present the chosen implementation, in order to show the developer side of our library. Finally, we detail the problems that may happen and how we try to address them.

3.1 Example

In the basic example 3.1 (page 8), we show how our library is used to instrument a benchmark code to collect measurements. This example is drawn from the source code of a finite state machine manipulation platform developed in our laboratory: the *Vaucanson* library [Claveirole (2004); Claveirole et al. (2005)].

This benchmark consists in measuring the time consumed by the execution of the `determinize` algorithm (line 31). We also measure the number of edges of the output automaton. This algorithm takes an automaton as input and produces an automaton as output. We measure it for different inputs size: 9, 12 and 15. We do so in order to test the scalability of our implementation of the algorithm. The input size is not exactly the number of states of the automaton. Actually, an input size of n represents an automaton composed by 2^n states, in our case. The input automaton is built by the `aut_2n` [1] function (lines 9 to 13).

The measurements are reported by the `Bencher` object built at line 26. As you can notice, in order to create a `Bencher` object, we need to give the name of the benchmark, its inputs, its outputs and an optional comment. The inputs describe the different inputs of the algorithm: their size, their name and their unit. The inputs and the input list are created by the code between lines 17 and 19. The inputs are described in further detail later in this report in section 3.2.2, page 10.

The outputs[2] describe the measurements performed by the benchmark. As for the inputs, the outputs are described by a unit and a name. The outputs and the output list are created by the code between lines 21 and 24. You can notice that the second output, called `utime`, is a special[3] output whereas the first one, `nb_edges`, is not. The outputs are described in greater detail later in this report in the section 3.2.3, page 10.

[1] The implementation is commented in order to not clutter the example with domain specific code which are irrelevant in the usage of our library.

[2] They are not the same as the outputs of the algorithm.

[3] Actually, this is a built-in output.

Program 3.1 *Determinize* benchmark program

```
1  #include <vaucanson/boolean_automaton.hh>
2  #include <vaucanson/algorithms/determinize.hh>
3
4  #include <ranch-cxx/ranch.hh>
5
6  using namespace vcsn;
7  using namespace vcsn::boolean_automaton;
8
9  automaton_t aut_2n(unsigned n)
10 {
11   /* Create an automata of 2^n state. */
12   return a;
13 }
14
15 int main()
16 {
17   Ranch::Input::Input nb_states("nb_states");
18   Ranch::Inputs inputs;
19   inputs.add(nb_states);
20
21   Ranch::Output::Output nb_edges("nb_edges");
22   Ranch::Output::UTime utime;
23   Ranch::Outputs outputs;
24   outputs.add(utime).add(nb_edges);
25
26   Ranch::Bencher b("determinize", inputs, outputs);
27   for (int i = 9; i <= 15; i += 3)
28     {
29       automaton_t a = aut_2n(i);
30       b.start(i);
31       automaton_t ret = determinize(a);
32       b.stop(utime(), (double)ret.edges().size());
33     }
34   return 0;
35 }
```

Once the `Bencher` object created, we use its two methods `start` (line 30) and `stop` (line 32) to tell to the `bencher` when the execution either starts or stops. We give the values of the different inputs as arguments of the `start` method. The value of the different outputs are given as argument of the `stop` method. The usage of these two methods are described in further detail later in this report (see section 3.2.1 on page 10).

Another important point to notice is that, whereas the elapsed user time measurement is obtained via the output itself, the number of edges is computed using a service provided by the tested program. This difference comes from the fact that, the *Ranch* library has to be generic and application domain independent. Thus, it can implement only built-in measuring instruments which are also application domain independent. The `UTime` and `Memory` outputs are two examples of such outputs. Thus, in most cases, the *Ranch* library does not exactly compute the measurements but just collect it. Even in the case of a built-in output, the *Ranch* library relies on other tools that already do the job. Further information about the implementation of the built-in

Figure 3.1 Result produced by the *Determinize* benchmark program

```
- 'determinize':
    type:
      inputs:
        - 'nb_states': ''
      outputs:
        - 'utime': 'sec'
        - 'nb_edges': ''
      comment: |

    content:
      -
        inputs:
          - 9    # nb_states()
        benches:
        outputs:
          - 0.03          # utime(sec)
          - 1536          # nb_edges()
      -
        inputs:
          - 12   # nb_states()
        benches:
        outputs:
          - 0.41          # utime(sec)
          - 12288         # nb_edges()
      -
        inputs:
          - 15   # nb_states()
        benches:
        outputs:
          - 4.45          # utime(sec)
          - 98304         # nb_edges()
```

outputs are available in the section 3.2.3 and 3.2.3 (on page 11).

The result produced on the standard input by this example is shown on figure 3.1. The output format is YAML [Ben-Kiki et al. (2004)] compliant. This format is easily readable by a script. Actually, such a script is used to populate theses results into a database. The semantic of the output is detailed in greater detail in the section 3.4 on page 13.

3.2 Implementation

This section describes the implementation of the *Ranch* library. This library may be implemented using different languages. Actually, it depends on the language used to write the code we want to instrument. In our laboratory, our programs are written in `C++`. Thus, the implementation presented here is written in `C++`.

The library consists of four main name spaces: `Bencher`, `Input`, `Output` and `Dumper`. All of them are part of the `Ranch` name space.

3.2.1 Bencher

The `Bencher` class is the corner stone of the *Ranch* library. Indeed, objects of this class drive the measurements performed during the benchmark execution. It starts and stops the registered measurement tools, gets the input and the output values and calls the `Dumper` object to produce the result (see section 3.2.4 for a description of the `Dumper` class).

In addition to the `start` and `stop` methods, the `Bencher` class also provides the following methods:

`Bencher& add_input(Input::Input& input);`
This method is used to register an input into the input list of the `bencher`. This method is usefull when the input list is not given as argument to the constructor.

`Bencher& add_output(Output::Output& output);`
This method is used to register an output into the output list of the `bencher`. This method is usefull when the output list is not given as argument to the constructor.

`Bencher& add_score(Output::Output& output, double score);`
This method adds a score according to the specified output. This method is usefull when the value/score of a given output is not given as argument to the `stop` method[4].

`Bencher& add_score(double score);`
This is an other version of the `add_score` method which allow to give the value of the outputs without specifing the output reference [5].

The reason why the two `add_score` methods exist is tackled in detail in the section 3.3 on page 12.

3.2.2 Input

The `Input` class is very simple. This class only provides *setters* and *getters* for its three attributes which are the name, the unit and the value. Since, these features are also needed by the `Output` classes, they have been factored in the `IO` super class.

3.2.3 Output

The `Output` class inherits from the `IO` super class. Contrary to the `Input` class, it implements two more methods: `start` and `stop`. These two methods are used to respectively start and stop the measuring instruments. For instance, the `UTime` class stores the user time in memory when `start` is called and subtract it from the current user time when `stop` is called. `start` is automatically called when the `bencher` is started. `stop` is either called by the `stop` method of the `bencher` or by the `add_score` method. The `Output` class also provides a short cut to the `stop` method: the parenthesis operator is implemented which call `stop` and then return the value. This allows to write code like that `utime()` at line 32 of the example 3.1 (page 8).

In order to ensure that the `start` and `stop` methods are called in the correct order the `Output` class check it with a flag. Consequently, subclasses such as the `UTime` class must overload two protected methods: `start_` and `stop_`.

If one wants to extend the *Ranch* library by providing additional built-in output, she only has to write a subclass of the `Output` class and to implement the `start_` and `stop_` methods. Nevertheless, if the new output is distributed with the library, it has to fulfill the requirements described in chapter 2 (on page 5) in order not to break the library. Mainly, the measuring instrument tool must be as platform independent as possible non-intrusive with the benchmark

[4] This method can not be mixed with the other version of the `add_score` method for implementation reasons.
[5] As a consequence, registration order must be followed.

code and application domain independent. The *Ranch* library currently comes with two built-in outputs which are described below.

Moreover, you can notice that built-in outputs in the *Ranch* library rely as much as possible on other tools. This is because generally, there already are many good tools that do the same task. The *Ranch* library is here to glue them into a common interface and unify the output format to ease the data gatherer script.

User time

The user time output is implemented by the `UTime` class in the `Output` name space. It measures the user time elapsed between the execution of its `start` and `stop` methods. It relies on the `times` system call and on the `sysconf` standard library call. Both are conformed to the POSIX standard.

The user time output must be used carefully since other measuring instruments may invalidate its result. Plus, it must be started/stopped as close as possible to the code under test. Typically, the memory usage output may alter the user time result since it generally overloads the memory allocator in order to compute the memory space used.

Memory usage

The memory usage output is implemented by the `Memory` class in the `Output` name space. There are several tools that allow to measure the memory space used between two points of execution of a program. Each of them have their advantages and their drawbacks. They are as follows:

Figure 3.2 Tools monitoring memory space usage

tool	free	portable	implementation	overload
mpatrol[Roy (1997)]	Yes	Yes	Simple	May be important
malloc_hook[GNU (2002)]	Yes	Yes	More complex	Negligible

`mpatrol` implements its own memory allocator and performs many sanity checks on memory accesses. Thus, it may be slower than the regular memory allocator.

The GNU memory allocator allows to install hook handlers in the call of `malloc`, `free` and `realloc`. Thus, it is possible to watch the number of memory allocated and it doesn't overload that much the usual memory allocator.

Thus, even if the `malloc_hook` option is more complex to set up since it is not only a function to call, we have chosen this approach to write the memory usage output. This way, it interferes the least with the other output measurement such as the user time.

3.2.4 Dumper

The `Dumper` name space encapsulates a dumper class hierarchy which is the back-end of the library. At the top of it, there is the abstract class `Dumper` which describes the contract a dumper must implement. The role of this hierarchy is to provide an abstraction on the syntax of the output format. Thus, the *Ranch* library back-end is easily extensible by means of sub-classing. Currently, a dumper that pretty-prints the results using a YAML [Ben-Kiki et al. (2004)] syntax is available.

In order to write a new dumper, developers must implement the four abstract methods defined by the `Dumper` abstract class. These methods are detailed below.

`void begin_bench(const Bencher& bencher);`
This method is called at the beginning of the benchmark. Basically, the first time the `start` method of the `Bencher` class is called. For instance, the YAML dumper prints the lines 1-10 of the result example 3.1. The goal of this hook is to print the header which describe the type of the benchmark.

`void end_bench(const Bencher& bencher);`
This method is called at the end of the benchmark. Basically, when the destructor of the `bencher` object is called. The YAML dumper only decrements the indentation counter but we can imagine that an XML dumper would have closed the opened tags.

`void begin_score(const Bencher& bencher);`
This method is called each time the `start` method of the `Bencher` class is called. The goal of this hook is to print the size of the inputs. The YAML dumper prints the line 11-14, 18-21 and 25-28 of the result example 3.1.

`void end_score(const Bencher& bencher);`
This method is called each time the `stop` or the `add_score` method of the `Bencher` class is called. The goal of this hook is to print the measurements of the benchmark. The YAML dumper prints the line 15-17, 22-24 and 29-31 of the result example 3.1.

3.3 Measurements incompatibility

As introduced in the section 2.3, some incompatibilities between measuring instruments may occur. These incompatibilities result in invalidating the measures obtained with the other instruments. As an example, what happens if the `ret.edges().size()` method call in the example 3.1 is a time consuming call? The `C++` compiler does not guaranty the evaluation order of the arguments. Thus, we can't assume that `utime()` is called before `ret.edges().size()`. That's why, in addition to the method `stop`, the `Bencher` class provides two other methods which allow to report the measurements without using the `stop` method.

The two `add_score` methods exist in order to be able to write the following code:

Program 3.2 `add_score` without output reference

```
b.stop();
b.add_score(utime()).add_score((double)ret.edges().size());
```

Program 3.3 `add_score` with output reference

```
b.stop();
b.add_score(utime, utime());
b.add_score(nb_edges, (double)ret.edges().size());
```

Indeed, with these two new versions of the example, we avoid the problem mentioned above.

From a general perspective, to mix measurement instruments on the same benchmark execution is dangerous and may alter the results. It is up to the benchmark writer to pay attention to this issue. Nevertheless, we are performing *regression* benchmarks so we care more about the difference between two executions rather than the absolute value reported each time. Thus, if the error is constant it may not invalidate too much the regression results. In all the other cases, it is advised to run the benchmark once for every measurement instruments even thought, it may slow down the overall benchmark suite.

3.4 Unified result format

In this section, we describe the result format shown in the figure 3.1.

The first line introduce the name of the benchmark. Then, there are two sections: a `type` section used to describe the contents of the benchmark and a `content` section used to report the measured values.

The `type` section lists the inputs name and unit of the benchmark. The same is done for the outputs and then a comment is appended.

The `content` section lists each execution of the benchmark. For each execution, it reports the size of every inputs and the measured value for every outputs.

The `benches` section, which appears at lines 14, 21 and 28, contains sub benchmark results. In our case, this section is empty.

3.5 Automatic data gatherer

In order to automatically gather measurements of our projects, our system relies on a continuous build system developed in our laboratory [Gournet (2005)]. This system is made of a set of Shell scripts, Rsync scripts and some Perl [Wall and Schwartz (1991)] scripts for the web interface. Its task is to watch a set of Subversion [Collins-Sussman (2002)] repositories and to build them as soon as a modification is detected. The build process consists of a configuration part, performed by a Shell script generated by Autoconf [Vaughan et al. (2000)], then the project is compiled using a `Makefile` generated by Automake [Vaughan et al. (2000)]. Finally, the `check` or `test` rules of the `Makefile` are executed in order to run the test suite. The dependencies between projects are also tackled to ensure the success of the build process. Then, we simply add a `bench` rule into the `Makefile` of the monitored project to continuously run the benchmark suite. Finally, the resulting data are parsed using a Ruby [Thomas and Hunt (2000)] script that populates the database storage (see the chapter 4 for further details about the storage format used).

Chapter 4

Ranch: web interface

Ranch provides a web interface in order to allow the users to easily consult the results of the benchmarks. This chapter covers this web interface. First of all, it describes the global architecture of the web application, to give an overview of the system. Then, it precises the structure of the database used to store the measurements. Then, the visualization interface is described. Finally, we show the graph obtained on a real use case.

4.1 Global architecture

The visualization application provided by *Ranch* is implemented as a web application. We have chosen to develop a web application instead of a classic graphical user interface, because it is centralized on the server storing the data and doesn't need to be packaged and deployed. Plus, the available web application frameworks have improved a lot during the past decade, and now, the cost of a web application has significantly reduced. Another point of interest is that web applications deal with databases very easily and our storage support is a database.

We use the *RubyOnRails* [Thomas et al. (2005)] web application framework to develop our web interface. We have chosen *RubyOnRails*, because it is easy to use, productive and testable. All the framework is implemented in Ruby. It implements the MVC[1] design pattern [Gorshkova et al. (2005)]. The application consists of:

- 5 model classes (one per database table) to manipulate the access to the database tables.
- 5 controller classes:
  - 1 for the home part of the site,
  - 1 for the projects management,
  - 1 for the systems management,
  - 1 for the benchmarks management,
  - 1 for the graphs generation.

  These controllers provide requests, scaling from listing the items, providing links to the next part to a form to set up the generated graphs.
- Each controllers requests are associated with one view which consists in Ruby code embedded in HTML code.

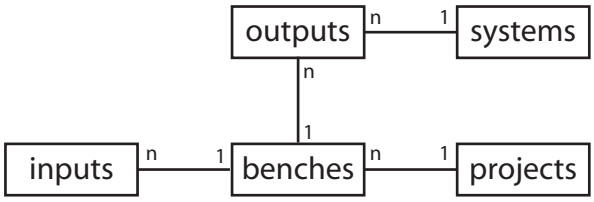
Several screen shots of the web application are available in the appendix A.

[1] Model View Controller

4.2 The database structure

The database used to store the measurements is MySQL. The structure of the database, in term of table relations is shown on the figure 4.1 on page 15. Instead of trying to find the best structure at the beginning of the project, we have preferred to add fields and to create tables when we needed them. The structure, now, has reached a point of stability. All the fields we need are present and none of them are useless. The main point we have been focus on is to avoid information redundancy in the database. Thus, the structure may appear unfriendly for a human, but we don't care since this is the problem of the web application and the populate script to deal with it. Plus, the *RubyOnRails* framework provides a good abstraction layer, so that we manipulate table's rows as instances of a class which describe the structure of the table. Relation between these objects are deduced from the field ending by `_id`, and thus the framework handles them itself. Nevertheless, it is possible that the database structure evolves again when we will need to add new features, such as users authentication for instance. However, all the data collected by the populate script can be stored in the current database.

Figure 4.1 Database structure



The following subsections describe each table of the database. Every table, have an `id` field used to identified uniquely a record, when managing relations between tables. A table which has a relation n to 1 with another table `foos` has a `foo_id` field. This the case of the `benches` table which has a `project_id` field.

4.2.1 The projects table

The `projects` table stores information about the registered projects. Currently, only the name and the head revision of the project are stored.

4.2.2 The benches table

The `benches` table stores information about the registered benchmarks for all the projects. Currently, only the names of the benchmark are saved.

4.2.3 The inputs table

The `inputs` table stores the value of all the inputs of all the benchmarks of all the projects. Obviously, the name, unit and value are stored. However, the argument number of the input is also stored in order to remember the order of the inputs for benchmarks composed by several inputs. Since a benchmark may consist in several executions of the same algorithm but with

different input values, the number of execution is also stored in order to remember which set of input values was used for a given execution of a benchmark.

4.2.4 The outputs table

The `outputs` table stores the measurements of all the outputs of all the benchmarks of all the projects. Obviously, the name, unit and value are stored. As for the inputs table, the argument number and the execution number are also saved in order to be able to map the outputs of a given execution with its corresponding inputs. The `outputs` has two more fields which are: the revision number of the project and the `system_id` on which the execution has taken place.

4.2.5 The systems table

The `systems` table stores a description of all the systems used to run the benchmarks of all the projects. This table tries to describe a system architecture with as few ambiguities as possible. Actually, we store the system information we expect to be the most significant on the result of the execution of a benchmark.

4.3 Systems management

This section tackles the problem that may occur if a system configuration changes. Technically, a system hardware upgrade or downgrade may significantly changes the measurement results. Specially the elapsed user time measurements. The web interface needs to warn the users that the graph may be invalidated in such a case. In order to address this issue, we have added the `systems` table in the database. The web interface must also be able to distinguish benchmarks executed on different machines, in order to display separate graphs. The information stored in this table is provided by the continuous build system. Obviously, we cannot collect exhaustively all the data needed to perform a rigorous check. Nevertheless, the most significant ones are saved. They are the following:

- hostname
- host type
- CPU frequency
- CPU name
- memory size
- compiler name
- compiler revision
- compiler flags
- operating system name

4.4 Visualization interface

The visualization interface is the most important part of the web interface. This part consists of two forms: one for the regression graph drawing and another for the scalability graph drawing. An example of the regression form can be seen on the figure A.1 in appendix A on page 21.

The regression benchmark form allows the user to choose a range of revisions, an input set and an output. The output value will be printed on the Y axis and the revision on the X axis. If all the output values of a given benchmark are chosen several graph will be drawn. The input set values must be fixed. This parameter is used to draw scalability graph.

We use a binding of the GD chart library in Ruby to draw the graph. The GD chart library is well known and heavily used to draw charts on the web. It is written in C, so that it ensures us a good scalability for big charts.

4.5 Results

Ranch has already given some promising results. We have been able to successfully compile our benchmark program of the *Determinize* algorithm with old revisions of *Vaucanson* (from HEAD revision down to the revision 805).

Generally, it is very hard to be able to compile old revisions of a library or a program. This is due to the fact that, specially configuration script, compilation programs and operating systems evolve and so it is very hard to configure an old program on a recent machine.

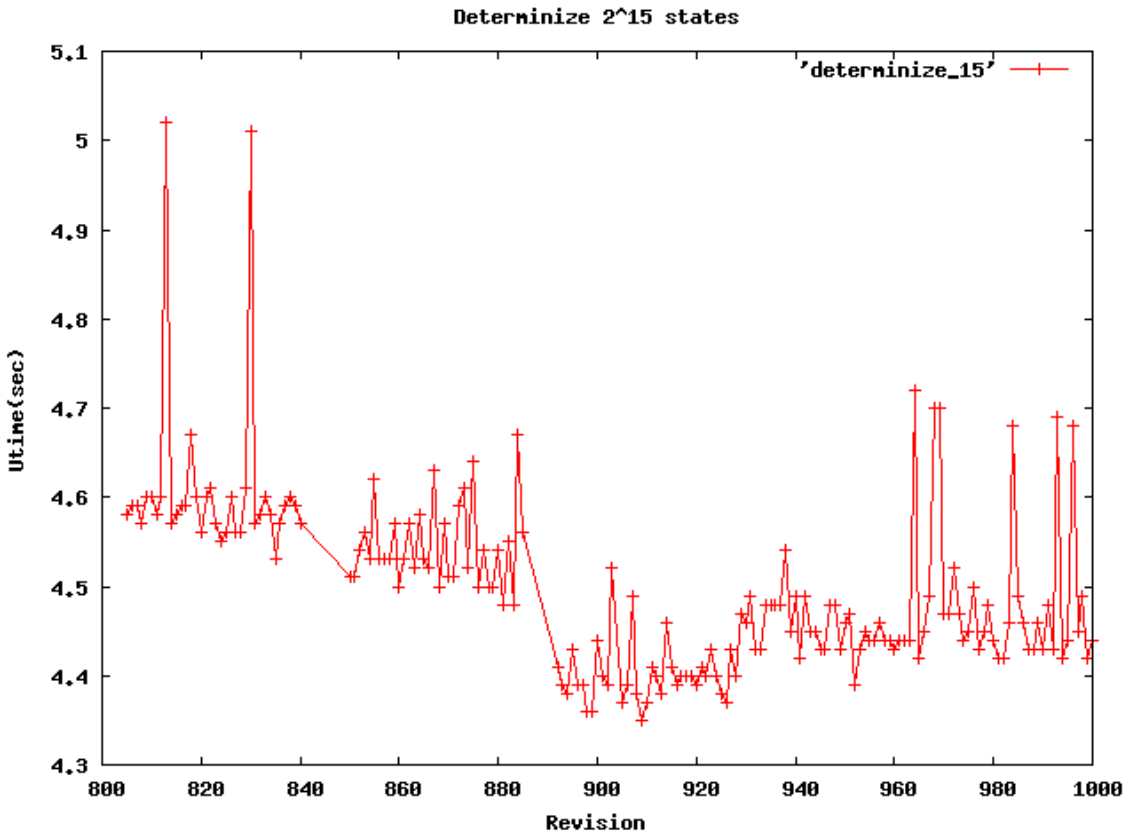
In the case of *Vaucanson* the task was simpler since it is fully static library written only in header files. *Vaucanson* doesn't distribute any binary library either dynamic or static. Thus, we don't need to compile *Vaucanson* before compiling and linking with our benchmark program. We only need to try to compile the benchmark program, which save us a lot of time. Actually, since *Vaucanson* uses very specific features of the `C++` compiler, it was harder to find a compatible compiler for the maximum number of revision, rather than a compatible system.

We haven't been able to compile our benchmark program with a revision prior to 805, because after this point, it was necessary to change the benchmark code to adapt it to the old interface of the library. We didn't do so, by fear of influencing the benchmark results, and so, to break the regression.

The results obtained by this benchmark are reported on the figure 4.2 on page 18. As you can notice, the global shape of the curve is very noisy. This is due to the load average of the machine. Even if the script was the only user of the machine, there are always others running programs. As you can notice, there are two important punctual regression between revisions 805 and 840. Obviously, they are not due to a real performance regression. Since the machine processor supports the hyper threading technology, we think that they represent executions where the processor has been able to optimize in this particular context. Obviously, if we execute again the program, the result is around the same value as the neighbor revisions. The same remark can be applied to the regression between revisions 960 and 1000.

However, a very interesting point can be noticed between revisions 880 and 900. Although, the differences between the revisions 885 and 892 are very small ($4,56$ seconds and $4,41$ seconds), we can see that the rest of the curve after the revision number 892 stay below $4,5$. We conclude this represents a real progression in the performance of the implementation of this algorithm.

Figure 4.2 Determinize algorithm performance from revision 805 to 1000 on an BiXeon CPU 3.20Ghx with 4Go of RAM



Chapter 5

Conclusion

In this report, we have presented a `C++` library that allows to gather and report measurement data for a benchmark program. This library is reusable in any benchmark program and provides domain independent methods to collect elapsed user time and memory space usage. It has been designed to be extensible by means of sub-classing.

We also have introduced a database and a web application structured to store and visualize collected benchmark values.

A real case example of a curves obtained with our system have proved the relevance of such a system.

There are still several issues to address in *Ranch*, such as registering new projects, writing more benchmarks and making the overall process more robust. Specially, the noise visible on the curve should be moderated.

Chapter 6

Bibliography

Ben-Kiki, O., Evans, C., and Ingerson, B. (2004). Yaml ain't markup language.

Claveirole, T. (2004). An overview of vaucanson 0.6.

Claveirole, T., Lombardy, S., O'Connor, S., Pouchet, L.-N., and Sakarovitch, J. (2005). Inside Vaucanson. In Springer-Verlag, editor, *Proceedings of Implementation and Application of Automata, 10th International Conference (CIAA)*, volume 3845 of *Lecture Notes in Computer Science Series*, pages 117–128, Sophia Antipolis, France.

Collins-Sussman, B. (2002). The subversion project: buiding a better cvs. *Linux J.*, 2002(94):3.

Courson, M., Mink, A., Marçais, G., and Traverse, B. (2000). An automated benchmarking toolset. In *HPCN Europe 2000: Proceedings of the 8th International Conference on High-Performance Computing and Networking*, pages 497–506, London, UK. Springer-Verlag.

GNU (2002). *Linux Programmer's Manual: malloc_hook(3)*. Free Software Foundation.

Gorshkova, E. A., Novikov, B. A., Belov, D. D., Gurov, V. S., and Spiridonov, S. V. (2005). A uml-based modeling of web application controller. *Program. Comput. Softw.*, 31(1):29–33.

Gournet, O. (2005). Autobuild project home page.

Kalibera, T. (2004). Regression benchmarking environment. In *WDS'04*, pages 174–178, Charles University. MatfyzPress.

Kalibera, T., Bulej, L., and Tuma, P. (2004). Generic environment for full automation of benchmarking. In *Net.ObjectDays 2004, First International Workshop on Software Quality (SOQUA 2004)*, pages 35–41, Ilmenau, Germany. tranSIT GmbH.

Roy, G. S. (1997). Mpatrol home page.

Thomas, D., Hansson, D., Breedt, L., Clark, M., Fuchs, T., and Schwarz, A. (2005). *Agile Web Development with Rails: A Pragmatic Guide*, volume 1. Pragmatic Bookshelf.

Thomas, D. and Hunt, A. (2000). *Programming Ruby: the pragmatic programmer's guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Vaughan, G. V., Elliston, B., Tromey, T., and Taylor, I. L. (2000). *GNU Autoconf, Automake, and Libtool*, volume 1. Sams.

Wall, L. and Schwartz, R. L. (1991). *Programming perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Appendix A

Web application screen shot

This appendix contains a captioned screen short of the web application of *Ranch*.

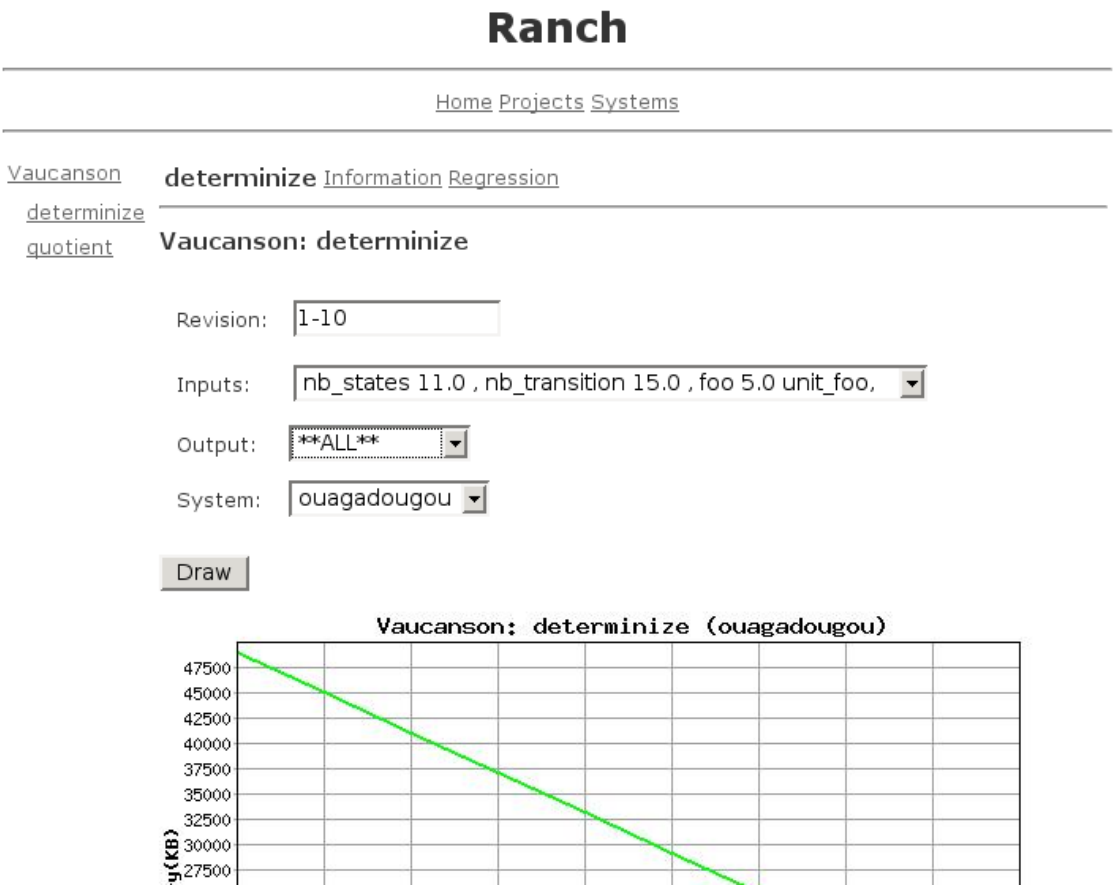



Figure A.1 Ranch web application regression form