Automatic regression benchmark system

Nicolas Desprès

Technical Report n^o0513, June 2005
revision 921

Regression benchmark is a part of regression testing that aims at an automatic detection of performance regression during application development. The goal is to detect as soon as possible the smallest change of performance. We need precise measurement of small parts of the program to achieve that. Although, many benchmark systems already exists none are fully automated and/or adapted to wide range of applications. However, automation is a crucial requirement in order to detect regression as soon as possible. This paper tackles generalities about performance measurements, then gives the requirements of such a system, and finally proposes a modeling.

L'évaluation des régressions de performance d'une application fait partie intégrante de la phase de test de régression. Le but est d'exprimer le plus précisément possible les différences de performance entre deux versions. Par précision, nous entendons à la fois la pertinence de l'estimateur de temps utilisé et la granularité des parties évaluées.

Bien que de nombreux systèmes d'évaluation comparative des performances existent déjà, peu sont complètement automatisés et/ou adaptés à différentes sortes d'applications. Pourtant, l'automatisation de cette phase est cruciale afin de détecter le plus tôt possible les pertes de performance.

Cet exposé présente tout d'abord les prérequis à la mise en place d'un tel système. Puis son architecture ainsi que son application au projet Transformers. Ensuite, nous comparerons différentes techniques d'estimations du temps d'exécution. Et enfin, nous évoquerons les possibilités d'adaptation d'un tel environnement à d'autre sorte d'applications.

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

Contents

Chapter 1

Introduction

This technical report presents 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 frequent 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 it[2]. Moreover the amount of benchmark result 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 to not overload the whole development cycle.

This report describes the regression benchmark system under development in our laboratory. This system aims at solving the problem introduced above. It is still a draft but the main module specification, in terms of requirements and design, are defined.

First of all, the requirements of the system are detailed. Then, the whole project's architecture is described.

[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.

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)]:

- It must perform and collect the measurement results automatically.
- It must provide a unified result format.
- It must manage a result repository.
- It must feature a user-friendly interface for result 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 run, to the addition of the results into the repository.

2.3 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 such a comparison, the result must be stored using the same format to avoid the use of converter. Moreover, we want to be able to benchmark sub parts of a benchmark. So, 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.

2.4 Results repository

The amount of collected data may increase quickly because, we want to perform measurement for every revision. So, 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.

2.5 Results analyzes/visualization interface

The result analyze/visualization interface must provide an easy and user-friendly way to generate graphs 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.

2.6 Platform-dependency

The end-user may wish to see the difference of performance of its program from an architecture to another. So, 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

Design

In this chapter, we present the design chosen to develop the regression benchmark system. First of all, we give an overview of the whole system. Secondly, the structure of the database is detailed. Then, we describe the typical use case scenario. Finally, we detail our suggestion for a unified result format. Finally, we argue quickly the tool set we have chosen to implement it.

3.1 Overview of the regression benchmark system

The system is composed of several components listed below:

- A benchmark suite.
- A populate script.
- A collect script.
- A web interface.
- A database.

The relationships between each component are shown on the figure 3.1 [1] (on page 8).

3.2 The benchmark suite

The benchmark suite is the part of the system which actually performs the measurements. On the figure 3.1 (page 8), we call this part of the framework: the *bencher*.

Most of the existing projects implement their benchmark suite by writing a test suite dedicated to emphasis the performances of the program instead of the correct behavior of its features.

Developers are generally interested in measuring the amount of time and/or the memory usage their program needs. These two values are easily computable by means of a reusable extern program (such as `time` or `valgrind` [Nethercote (2004)]). These tools are convenient because they are not intrusive in the program code. In other word, they rather need meaningful test suites than code instrumentations in order to be relevant.

Many projects also need more specific information. For instance, our project *Olena* [Duret-Lutz (2000)], an image processing library, can compute the number of times an algorithm accesses to a pixel of an image. This information is very interesting in order to optimize an algorithm. Contrary to the time and memory usage values, the computation of such a value implies to instrument the library code. This example illustrates that it is very hard with common languages

[1]The CRUD abbreviation is the Create, Read, Update and Delete actions sequence that is usually performed on a database.

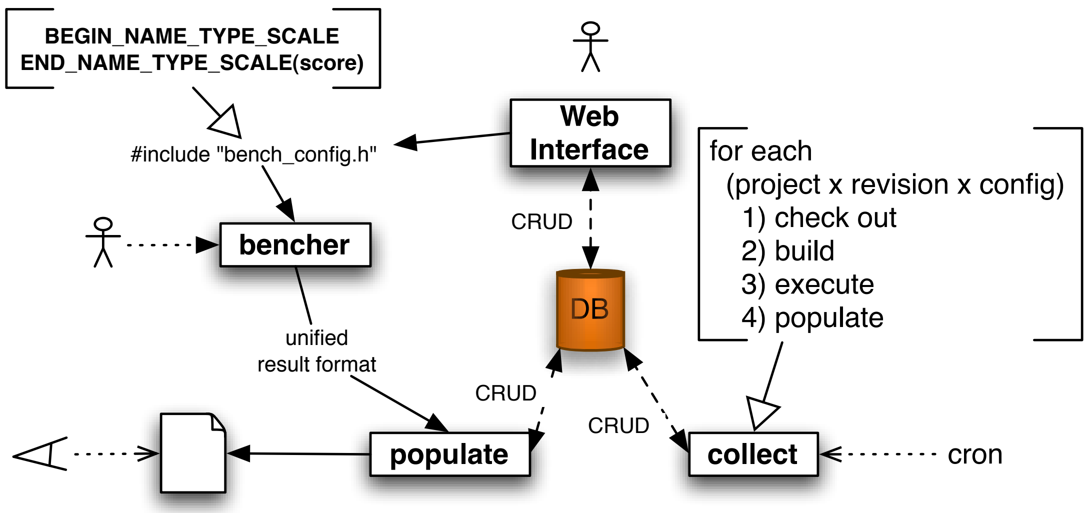

Figure 3.1: Regression benchmark system overview

to develop a generic tools that can help to compute any measure one may need. That's the reason why our regression benchmark system does not feature a generic tool to ease the writing of benchmark measurements. Nevertheless, this point is tackled in [Kalibera et al. (2004)].

However, as discussed in the previous chapter, we want to unify the format used by the benchmark suite to print its results. This format is not only a matter of layout. It asserts that necessary information is present. The information is the description of every benchmark of the project and their results. The description must be non ambiguous, in order to ensure that the insertion of the result in the database won't need any human interactions during the entire process. The materials provided to help the developers to print the right information using the right format is described in the section dedicated to the unified results format 3.8 on page 11. This material is provided as a library and it is generated from the information contained in the database.

3.3 The populate script

The goal of the populate script is to take the benchmark results (written using the unified results format) as input and populate them into the database. This script is called either by the *collect* script (see section 3.4) or by a human operator.

The benchmark suite of a given project may be run without our project installed on the machine. That's why the module that commit the information into our database is embedded into the *populate* script instead of the benchmark suite.

Moreover, in case of an unexpected ambiguous benchmark description which could not be committed for sanity reasons into the database, the *populate* script keeps track of the result until a human intervention. Thus, the automatic process is not interrupted and no data are lost.

3.4 The collect script

The goal of the collect script is to periodically run the benchmark suite of every revision of every project registered in the database and on every configuration mentioned. Thus, it fills the database and ensures that none revision measures are missing.

3.5 The web interface

The web interface allows the user to draw graphs and charts based on the measures stored in the database. We have chosen a web based application instead of an X window one for portability reasons.

3.6 The database

The database is the corner stone of the system. It is designed to avoid duplicated field and to support every benchmark type. The figure 3.2 (page 9) represents the relationship between the tables of the database.

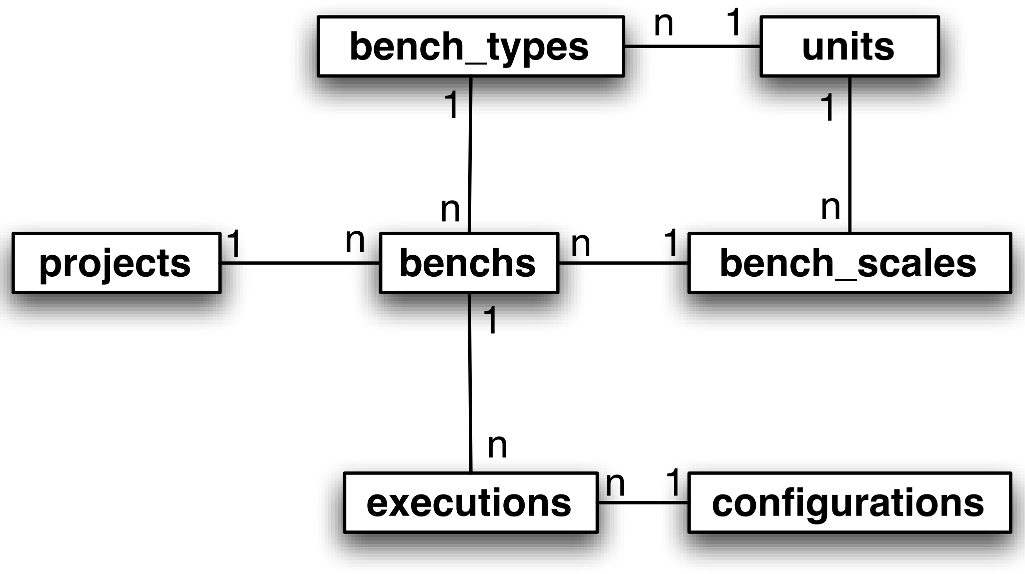

Figure 3.2: The results database description

3.6.1 Tables description

The central table, called `benchs`, stores every single benchmark. In this database, a benchmark represents one measure of one feature of one project. A measure is qualified by a type and a scale.

The measurement type tells us, for instance, if the benchmark measures the memory usage or the duration of the program. Thus, the type has a unit such as bytes or seconds.

The measurement scale codes the program input size used by the benchmark. This allows us to perform scalability benchmark. A scale is also qualified by a unit.

The `benchs` table has the following fields: an id, a name, a project id, a type id, a scale id, etc...

Because a benchmark may not be available from the beginning (first revision) of a project to its end, there are two more fields called `start_revision` and `stop_revision`. They indicate between which revision interval the benchmark may be perform.

The `executions` table stores the result of a benchmark collected for every valid revision and every available configuration. The benchmark is mentioned in this table by means of its `id` in the `benchs` table. The `executions` table allow us to easily check the performance regressions for a given benchmark of a project.

3.6.2 Table alteration versus numerous records

We have designed this database to avoid having to alter a table while the system is running. We have also paid attention to not duplicate data. Thus, the table relationships may seem complex, but it is not relevant since it is maintained by the system and not by the users. Currently, we prefer to have a table with many records instead of creating new tables on the fly.

3.7 Typical use case scenario

The typical use case scenario is shown on the figure 3.3 and is detailed here:

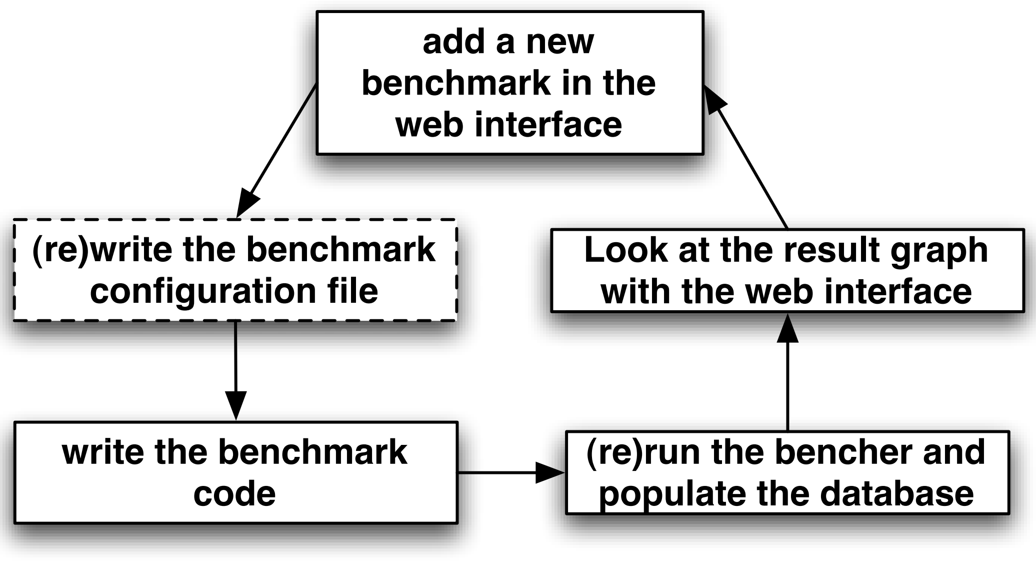
add a new benchmark in the web interface
(re)write the benchmark configuration file
Look at the result graph with the web interface
write the benchmark code
(re)run the bencher and populate the database

Figure 3.3: A typical use case scenario

1. Register a new benchmark in the database via the web interface. This includes the addition of necessary new types or scales or units.
2. Ask the system to regenerate the benchmark configuration file. Basically, this file features materials to help the developer to print the results using the expected format.
3. Write the code needed of the new benchmark in the project's benchmark suite.
4. Run again the benchmark suite and redirect the results to the populate suite. This stage is optional since people may wait for the periodical benchmark execution.
5. Finally, once the population process is finished you can watch the results by means of charts using the web interface.

3.8 The unified results format

This format aims at representing every benchmark results type. It is, so-called, unified because every project may use it as the output format of their benchmark suite.

3.8.1 Featured materials

The developers of the benchmark suite should not know the unified format that we provide. First of all, it may be very cumbersome for them to print it properly. Secondly, if we change it, they will have to adapt their benchmark suite. Thirdly, they don't have to know all the information we require to describe without ambiguity a benchmark and its result.

So, we provide a library which contains all the information needed which are registered in the database for a given project. Basically, the benchmarks name, type and unit are available. Then, the library interface features mainly two functions with the following prototypes:

```
void begin_benchmark(const char *name,
                     const char *type,
                     const char *unit);
void end_benchmark(double score);
```

For instance, the developers of the benchmark suite may use these functions this way:

```
#include "benchmark.h"

static double do_the_bench(void)
{
  double score;

  /* compute the value of the score variable... */

  return the_score;
}

int   main(void)
{
  double score;

  begin_project(MY_PROJECT);
  begin_benchmark(MY_BENCH_FOO, MY_TYPE_BAR, MY_INPUT_BAZ);
  score = do_the_benchmark();
  end_benchmark(score);
  end_project();
  return 0;
}
```

Figure 3.4: An example of a benchmark code

The `begin_benchmark` function prints the description of the benchmark which is going to be run. The description is the `id` of the benchmark in the `benchs` table of the database. This `id` is computed by the hash of the concatenation of the benchmark name, type and input strings. Since the type name and input name are kept unique in the database and the benchmark name for a given project as well, there are no ambiguities. After calling the `begin_benchmark` function, we really compute our benchmark and then, give the score as argument to the `end_benchmark` function call. The `begin_project` function call indicates that the benchmarks written until the `end_project` function call, belong to the project `MY_PROJECT`. The purpose of these four functions is only to print the needed information to the standard output. Of course, you may nest benchmarks and projects.

The macros `MY_PROJECT`, `MY_BENCH_FOO`, `MY_TYPE_BAR` and `MY_INPUT_BAZ` are also provided by the generated library. Their name is almost equivalent to the string they replace. They are provided to avoid the developer to misspell some of the names written in the database. If the macros are misspelled the compiler will warn the developer whereas no error will be raised if the developer misspells a string. This kind of errors may be difficult to investigate and may break the automatic process.

3.8.2 Chosen language

According to the requirements mentioned in 2.3, there are two obvious data representation languages available: XML and YAML [Ingerson et al. (2002)]. Both languages are easy to read and write using a script: support for many script languages already exists. XML supports nested data by means of nested tags whereas YAML supports it by means of indentation. The interface we have chosen for our library makes harder to manage indentation since we must memorize the indentation level over the different calls to the `begin_xxx` and `end_xxx` functions. That's why; we prefer to use the XML language.

Here an example of a benchmark output:

```
<project name="my_project">
 <benchmark>
  <id>012345678901234567890123456789O1</id>
  <starttime>2005-12-14 22:15:43.10 -05:00</starttime>
  <benchmark>
   <id>abc3456e890f234b67c901eda5678fbc</id>
   <starttime>2005-12-14 22:15:43.10 -05:00</starttime>
   <stoptime>2005-12-14 22:17:43.10 -05:00</stoptime>
   <score>5100.98</score>
  </benchmark>
  <benchmark>
   <id>abc3456e890f234b67c901eda5678fbc</id>
   <starttime>2005-12-14 22:18:43.10 -05:00</starttime>
   <stoptime>2005-12-14 22:58:43.10 -05:00</stoptime>
   <score>10</score>
  </benchmark>
  <stoptime>2005-12-14 22:59:43.10 -05:00</stop_time>
  <score>52.1</score>
 </benchmark>
 <benchmark>
   <id>98765432109876543210987654310320</id>
  <starttime>2005-12-14 22:59:45.10 -05:00</starttime>
  <stoptime>2005-12-14 23:01:43.10 -05:00</stoptime>
  <score>51.02</score>
 </benchmark>
</project>
```

Figure 3.5: An example of the unified results format

Obviously, the `begin_project` function prints the `<project name="my_project">` tag and the `end_project` function prints the `</project>` tag.

The `begin_benchmark` function prints the following tags:

```
<benchmark id="01234567890123456789012345678901">
 <id>01234567890123456789012345678901</id>
 <starttime>2005-12-14 22:15:43.10 -05:00</starttime>
```

The `end_benchmark` function prints the following tags:

```
 <stoptime>2005-12-14 22:59:43.10 -05:00</stop_time>
 <score>52.1</score>
</benchmark>
```

Chapter 4

Conclusion

In this report, we have described the design of a regression benchmark system. Even if we have tried to design it flexible and automated enough, some parts of it are not yet well handled. This is the case of the bencher module which is still mainly written by the project maintainers.

Moreover, we haven't tackled the accuracy of the benchmark measurements which in some cases may be very noisy and so may introduce wrong data in the regression system (Bulej et al. (2004)). The automation of the writing of benchmark suite hasn't been covered too (see Kalibera (2004)).

Chapter 5

Bibliography

Bulej, L., Kalibera, T., and Tuma, P. (2004). Regression benchmark with simple middleware benchmarks. In *the 2004 Internation Performance Computing and Communications Conference (IPCCC 2004)*, pages 771–776, Charles University. IEEE.

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.

Duret-Lutz, A. (2000). Olena: a component-based platform for image processing, mixing generic, generative and OO programming. In *Proceedings of the 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE)—-Young Researchers Workshop; published in "Net.ObjectDays2000"*, pages 653–659, Erfurt, Germany.

Ingerson, B., Evans, C., and Ben-Kiki, O. (2002). Yaml ain't markup language.

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.

Nethercote, N. (2004). *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge.