RTJBench: A REAL-TIME JAVA BENCHMARKING FRAMEWORK

MAREK PROCHAZKA, ANDREY MADAN, JAN VITEK, WENCHANG LIU

Abstract. The paper gives an overview of RTJBench, a framework designed to assist in the task of benchmarking programs written in the Real-Time Specification for Java, but with potentially more general applicability. RTJBench extends the JUnit framework for unit testing of Java applications with tools for real-time environment configuration, simple data processing and configurable graphical presentation services. We present design principles of RTJBench and give an example of a benchmarking suite we have been using for daily regression benchmarking of the Open Virtual Machine.

Keywords: Benchmarking, regression benchmarking, Real-Time Specification for Java

1. Introduction

Tracking performance of large software systems in a consistent manner over time is a challenging problem. In the Ovm project [OVM], we have been developing an open source, production quality, Java™ Virtual Machine compliant with the Real-Time Specification for Java (RTSJ) [BGB+00]. The performance of a virtual machine is a function of many factors such as data layout, compiler optimizations and operating system interactions. For real-time applications both performance and predictability are crucial, and seemingly innocuous changes in the code base can make a significant difference in both dimensions.

In the context of a large software project as Ovm it is necessary to automate the tasks of regression testing and benchmarking of performance and predictability. None of the existing tools provides the support needed for both tasks (moreover, support for the RTSJ adds interesting problems that have not been addressed before). RTJBench is a testing and benchmarking framework designed to meet the following goals:

- Provide a simple API for writing and configuring individual benchmarks. Individual benchmarks can be run from command line as well as programmatically by creating benchmarking suites that group several benchmarks.
- Provide a simple data processing framework that allows persistent storage and processing of benchmarking data.
- Support automatic output generation with no human intervention. While it is acceptable to create one-off graphs by hand, this ceases to be the case when many similar graphs have to be generated.
- Virtual machine independence, the framework should not be tied to a particular virtual machine. In the case of RTJBench, we restrict ourselves to the RTSJ APIs.

2. The RTJBench framework

We have extended the well-know JUnit framework [JUn] with support for benchmark configurations. Similarly to JUnit, a benchmark is defined in a class that inherits from `OvmTestCase`. All methods whose name starts with the keyword `test` are considered benchmark methods. There are two ways how to configure a benchmark. The first of them is to define arguments, which are specific to each benchmark and allow it to run with different number of threads, locks, iterations, etc. An `Argument` object is either passed to the test case constructor or extracted from command line arguments.

The second way to configure a benchmark is to define its real-time execution environment. Every test method is executed by RTJBench in an environment defined in the `Configuration` object. `Configuration` is not benchmark specific and for example defines in which memory test methods are executed (either `HeapMemory` or `ScopedMemory`), which thread type is used (`RealtimeThread` or `NoHeapRealtimeThread`), which `NoiseMaker` implementation is used to generate background load during the benchmarking, how many background threads is started, etc. For example, an appropriate configuration, can generate benchmark results that either include garbage collection noise or pure, GC-free, performance results obtained with `ScopedMemory` and `NoHeapRealtimeThread`s.

Several benchmarks form a *benchmarking suite*. A benchmark could be added to a suite with different configurations. For example, a suite may contain the same benchmark running with or without background noise threads, running in heap or scoped memory, etc. If `ScopedMemory` is used, a fixed sized scope is allocated, and each benchmark in the suite enters the scope, executes the computation, and leaves the scope. This way the same memory is used for

all benchmarks, no garbage collection occurs in the middle of any benchmark, and hence memory intensive test suites that contain several benchmarks storing big amount of data can be executed.

2.1. Data processing

RTJBench defines a simple spreadsheet-like API to store all benchmark data in a simple file-based repository. Every benchmark execution stores a single data table with arbitrary numbers of rows and columns. Benchmark configuration and parameters are stored with the table as a list of name/value attributes. The API allows to find in the repository data related to a particular benchmark execution according to its name, time it was performed, as well as according to any attributes.

Data files are processed off-line after the suite execution is finished in order to avoid having to load processing code with the collection code[1]. For data processing we defined a simple *data filter* API. A data filter is able to parse a given data table or array of data tables and produce another table. Each benchmark has one or more associated data filters. For example, the *Yield Latency* benchmark produces a table with two columns and as many rows as there are iterations in the benchmark. The two columns contain timestamps obtained before and after the yield operation. A `MinMaxAvgYieldLatencyFilter` parses the table and produces another table with two columns and three rows. The first column contains textual result description and the second one minimal, average, and maximal yield latency measured during the benchmark run.

Thanks to the fact that a filter parses a table or array of tables, and produces another table, several filters can form a *hierarchical filter*. For example, an array of tables produced in different runs by `MinMaxAvgYieldLatencyFilter` are parsed by `RegressionMinMaxAvgFilter` that creates a table with history of minimal, maximal, and average yield latencies measured. Reusability is another advantage as some filters are used by different benchmarks, for example tables of before- and after-timestamps are used by many latency microbenchmarks.

2.2. Output

A *plotter* is an entity that is able to plot data stored in a table. Several plotters can be used with a single benchmark. Since a plotter takes a table or an array

[1] In space constrained environment, it is crucial to reduce the footprint of the instrumentation. For example, RTJBench is also run on an embedded PPC board with limited resources. Moreover, the Ovm ahead-of-time optimizing compiler is able to generate better code if fewer classes are included in an image.

of tables as its input, it can be used with different filters, as well as a filter can be used to pass data to different plotters. Currently we support two plotters. `TerminalPlotter` prints results and descriptions stored in the input table on the screen. `GnuPlotPlotter` is a Java library on top of the *GnuPlot* plotting program [Gnu], and supports displaying graphs on the screen and storing generated graphs in various graphical formats.

To generate number of graphs based on the same run of a benchmarking suite, we once again use the JUnit framework – we define so called *display suites*. Each display suite defines all outputs by identifying appropriate benchmarks, filters, and plotters. By executing a suite we get all results and graphs in desired formats. Currently we have three display suites that process data collected by a single run of our *OvmSuite*. *OvmDisplaySuite* displays various graphs on the screen, *ReportSuite* generates dozens of postscript files for research reports, and *WebSuite* generates files in *png* format to be displayed on our web site.

3. Case study: Benchmarking Ovm

Since April 2004 we use RTJBench for daily automatic regression benchmarking of Ovm[2]. We use *OvmSuite* that contains several latency microbenchmarks and *SpecSuite* suite that parses SPEC JVM98 benchmarks [Spe]. Based on one execution of our regression benchmarking suite, we daily generate the following output:

- regression graphs presenting evolution of Ovm minimal, maximal, and average latencies over time,
- comparison of Ovm minimal, maximal, and average latency with a commercial implementation of RTSJ,
- graphs presenting individual latency values measured in every iteration of each benchmark (currently we use 11000 iterations and the first 10% iterations are considered a warmup),
- a regression graph of Ovm performance measured using the SPEC JVM98 benchmark,
- a graph comparing performance of two different configurations of Ovm with three other Java Virtual Machines using the SPEC JVM98 benchmark,
- a regression graph with tracing sizes of Ovm image and executable files over time,

[2] All results are available at `http://ovmj.org/bench`

- for all the graphs that do not show Ovm regression, we generate browsable history of all historical results.

We use the same benchmarking suite on three various platforms (Timesys Linux RTOS, Mac OSX, and Embedded Linux BSP on an embedded board). Examples of generated graphs are on Fig. 1, 2, 3.

Our regression benchmarking helped us many times to quickly observe that some recent development in Ovm added undesirable performance overhead or spoiled its predictability. Despite the fact that we do not benchmark middleware but Real-Time Java, our experience confirms many of the issues pointed out in [BKT04] and [KBT04] dealing with middleware *regression benchmarking*. We measure much smaller absolute values (order of microseconds) and therefore the influence of the operating system is much higher and there are strong requirements for the precision and predictability of timers we use. Since none of our benchmarks is distributed, we do not have to deal with appropriate network setting, use of multiple nodes, and less predictable network latency, which all are major issues in middleware benchmarking.

Benchmarking a JVM does not require so long warmup periods, as the only warmup operations are related to loading hardware and software caches, which in general takes only few instants. In real-time systems, we are not so much interested in average or median results and filtering out exceptional outliers in benchmarked system performance. Instead, we are interested in the approximation of the worst case execution time, so we often use the maxima of measured results as the most significant metrics.

The latency benchmarks we use are similar to those used in other RTSJ benchmarking projects ([SPLI03], [CS02], [BLMW03]). The main contribution of RTJBench is to have not only a collection of benchmarks for getting some numbers, but a framework for composing different benchmarks to suites, running benchmarks in different configurations, and last but not least, having efficient tools for processing collected data and automated generation of human understandable outputs.

4. Conclusion

We have presented our RTJBench framework for benchmarking Real-Time Java implementations. We have provided a brief overview of the RTJBench architecture and shared our experiences from benchmarking Ovm, our implementation of the Real-Time Specification Java. In the future we would like to focus on the regression benchmarking features of RTJBench and consider the use of hardware counters to reason about measured results more precisely.

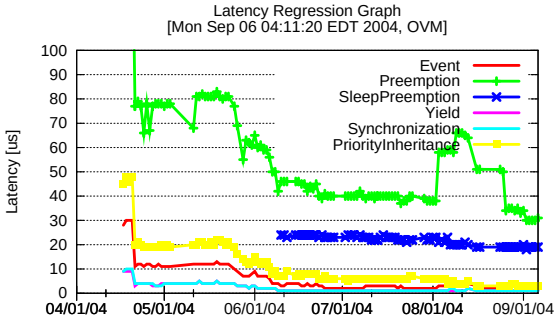

Figure 1: Latency regression graph.

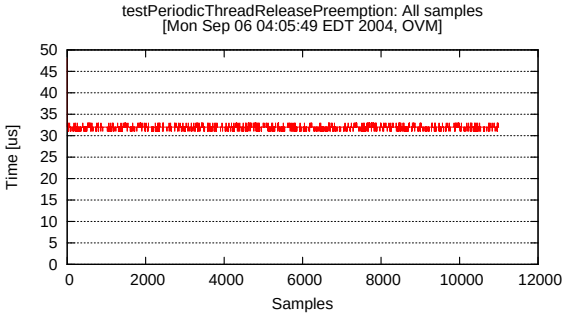

Figure 2: Preemption latency all-samples graph.

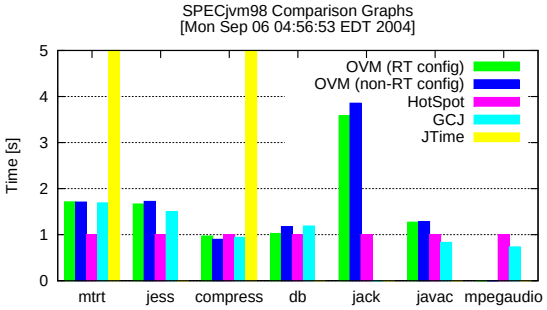

Figure 3: SPEC JVM98 comparison graph.

References

[BGB+00] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000. `http://www.javaseries.com/rtj.pdf`.

[BKT04] Lubomir Bulej, Tomas Kalibera, and Petr Tuma. Regression Benchmarking with Simple Middleware Benchmarks. In *Proceedings of International Workshop on Middleware Performance (IPCCC 2004), pp. 771-776, Phoenix, AZ*, April 2004.

[BLMW03] Greg Bollella, Krystal Loh, Graham McKendry, and Thomas Wozenilek. RTJ Experiences & Benchmarking with JTime. Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), International Federated Conferences (OTM '03), Catania, Italy, November 2003.

[CS02] Angelo Corsaro and Doug Schmidt. The design and performace of the jRate Real-Time Java implementation. In *Proceedings of the 4th International Symposium on Distributed Objects and Applications (DOA'02)*, November 2002.

[Gnu] Gnuplot. `http://www.gnuplot.info`.

[JUn] JUnit regression testing framework. `http://junit.org`.

[KBT04] Tomas Kalibera, Lubomir Bulej, and Petr Tuma. Generic Environment for Full Automation of Benchmarking. To appear at the First International Workshop on Software Quality (SOQUA 2004), Erfurt, Germany, September 2004.

[OVM] The Open Virtual Machine Project. `http://ovmj.org`.

[Spe] The Standard Performance Evaluation Corporation: SPEC JVM98 Benchmarks. `http://www.spec.org/jvm98`.

[SPLI03] David C. Sharp, Edward Pla, Kenn R. Luecke, and Ricardo J. Hassan II. Evaluating Real-Time Java for Mission-Critical Large-Scale Embedded Systems. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, May 2003.

Authors addresses:
Marek Prochazka, Andrey Madan, Wenchang Liu, Jan Vitek
Purdue University
Department of Computer Sciences
250 N. University Street
West Lafayette, Indiana, 47907-2066
{marek,jv}@cs.purdue.edu
andrey.madan@medtronic.com
liu27@purdue.edu