The embedded Java benchmark suite JemBench

Schoeberl, Martin; Preusser, Thomas B.; Uhrig, Sascha

Published in:
Proceedings of the 8th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2010)

Link to article, DOI:
10.1145/1850771.1850789

Publication date:
2010

Document Version
Early version, also known as pre-print

The Embedded Java Benchmark Suite JemBench

Martin Schoeberl
Department of Informatics and Mathematical Modeling
Technical University of Denmark
masca@imm.dtu.dk

Thomas B. Preußer
Department of Computer Engineering
Technische Universität Dresden, Germany
thomas.preusser@tu-dresden.de

Sascha Uhrig
Department of Computer Science
University of Augsburg, Germany
uhrig@informatik.uni-augsburg.de

ABSTRACT

Requirements to embedded systems increase steadily. In parallel, also the performance of the processors used in these systems is improved leading to multithreaded and/or multicore architectures. Depending on the type of the embedded system, using Java is a more and more popular way for software development. In this paper, we present a Java benchmark suite that enables the comparison of different embedded Java platforms while solely assuming the availability of a CLDC API, the minimal configuration defined for the J2ME. The core of the benchmark suite consists of adapted real-world applications. Furthermore, the suite contains benchmarks to explore multi-core/multi-threaded systems. Hence, it is possible to determine the gain of a parallel execution platform compared to sequential execution. Additionally, the penalty of a sequential program running on a parallel platform can be measured. Our benchmarks are structured in *micro*, *kernel*, *application*, *parallel*, and *streaming* benchmarks.

1. INTRODUCTION

Benchmarks are important for the development of embedded systems. While benchmarks for standard Java, server-oriented Java, and Java on mobile phones are available, no Java benchmark suite is available for classic embedded control systems. The presented benchmark suite JemBench fills this gap.

Embedded systems are often resource-constrained. A full JDK is usually too big for the embedded target. Our benchmark suite is designed to run with minimal JDK support. Furthermore, the benchmarks are designed to run with a small memory footprint (less than 1 MB) and do not require any file I/O. All benchmarks conform to the CLDC 1.1 [26][1] standard. Additionally, they can be executed on RTSJ and Safety-Critical Java (JSR 302) platforms.

Realistic benchmarks include real applications and not only toy kernels. For JemBench, we have adapted several embedded Java applications that are in industrial use [20]. Besides sequential benchmarks, we also developed several multithreaded benchmarks, which perform data-parallel and pipelined execution.

As the benchmark suite does not target consumer electronics, but embedded control systems, the benchmarks are mainly designed in a WCET (Worst Case Execution Time) analyzable way. Hence, a target platform can be examined concerning their real-time capability by performing a static WCET analysis and compare the results to the actual execution. Of course, the real-time capability of the used platform cannot be proven but an estimation of the WCET analysis' tightness can be obtained.

Depending on the platform, the execution time of an application may vary dramatically. To reach a feasible accuracy in combination with a reasonable execution time independent of the system's performance, the benchmarks adapt themselves to the capability of the target platform.

The provided benchmark suite contains sequential and multithreaded benchmarks, organized in five categories that can be started all together or individually. The first category includes simple micro benchmarks testing individual bytecodes of different complexity. Two typical kernel algorithms are packed together into the kernel category. Three applications from industrial use form the third category. These application benchmarks are the main benchmarks to measure single threaded performance. The multithreaded benchmarks consist of two groups: data parallel applications and a streaming benchmark. The number of threads to be used by the multithreaded benchmarks is defined in a utility class that is part of the benchmark suite.

The presented benchmark suite is freely available as open source under the GNU GPL license. Instructions how to access the source and run the benchmarks can be found in the abstract.

In the remainder of this paper, an overview over embedded Java systems is given first by Section 2 before Section 3 existing benchmarks are discussed. Section 4 introduces our JemBench benchmark suite whose characteristics are then be evaluated in Section 5. Finally, Section 6 concludes this paper.

2. EMBEDDED JAVA

In this section, we provide an overview of different Java runtime systems for the embedded domain, the domain JemBench was developed for. A Java virtual machine (JVM) can be implemented as a software environment executed on an arbitrary (embedded) processor or as a hardware solution using a processor that is able to execute Java bytecode natively.

2.1 Embedded JVM Implementations

A JVM implemented in software contains the Java library classes, several methods to connect the Java application to the underlying operating system and a bytecode interpreter and/or a just-in-time (JIT) compiler. Obviously, an operating system is required

[1]See also: `http://java.sun.com/javame/reference/apis/jsr139/`

JTRES'10 August 19–21, 2010 Prague, Czech Republic

that is responsible for the native I/O, such as file and network access. The supported Java standard depends on the available resources and the actual JVM implementation.

Examples of software implemented JVMs that can run in an embedded setting (among others) are the JamaicaVM [22], IBM J9 VM,[2] PERC Pico [2], KertasarieVM,[3] KVM [25], Squawk [23], CACAO [14], and Fiji VM [18].

2.2 Java Processors

In contrast to the software implementations, Java processors do not require any interpreter or compiler. In general, the operating system functionalities are also implemented in Java leading to a homogeneous control flow from the application to the I/O. Additional abstraction layers are not required. Moreover, the exception concept can be used also at the I/O level that means an explicit error handling in every hierarchy level is not required (`try/catch` blocks do not need extra bytecode instructions in the normal program flow).

Regarding the real-time capability, Java processors benefit from the omitted software JVM and operating system. Hence, it is sufficient that the processor and the JVM libraries are designed in a real-time capable way. JOP [17], SHAP [29], jamuth [28], aJile [1], and picoJava [9] are some examples of Java processors.

2.3 Real-Time and Safety-Critical Java

The Real-Time Specification for Java (RTSJ) [6] is an approach to enable Java for embedded systems. With a stricter definition of the scheduling algorithm and a memory model that avoids garbage collection, the RTSJ targets embedded real-time systems. RTSJ is defined under Java 2 ME with the restricted library of the CLDC. Therefore, JemBench shall be executable on all confirming RTSJ implementations. It has to be noted that most vendors of RTSJ implementations provide the JDK of the Java SE in version 1.5 or 1.6.

The Safety-Critical Java (SCJ) specification [12] is under development within the Java Community Process (JSR 302). A first draft for public review is expected this year. SCJ is intended for safety-critical applications that need to be certified. SCJ defines three levels of compliance: level 0 represents a cyclic executive, level 1 a static defined mission with preemptive scheduling, and level 2 allows dynamically created submissions. To simplify the certification process SCJ defines a very small subset of the standard library. As one of the authors is member of the SCJ expert group, we have access to early drafts of the specification. We developed JemBench according to the library definition of SCJ. All JemBench benchmarks shall be able to execute on all levels of SCJ.

SCJ implementations are currently under development, e.g., on top of the research JVM OVM [3]. A reference implementation of SCJ that executes on top of the RTSJ is under development by the SCJ expert group and will be freely available with the SCJ specification.

3. RELATED WORK

Several benchmarks for server and desktop JVMs are available, as described in this section. Benchmarks for embedded Java are not so common, and the few available target the mobile phone platform Java ME.

3.1 Java Standard Edition

DaCapo [4] is an open-source benchmark for Java that is intended to better serve the Java community than SPEC JVM98. The collection of benchmarks includes more complex code, more object-oriented applications, and puts more stress on the dynamic memory management. DaCapo has become the standard benchmark for the GC research community. DaCapo uses open-source tools and applications such as a parser generator, bytecode optimizer, graph plotting, Eclipse, an SQL database engine, a python interpreter, text index and search, source code analyzer, and an XSLT processor. The authors also argue for investing more effort in sound benchmarking methodologies [5]. The proposed embedded benchmark suite is our contribution to this field for a specific application domain, which is still underrepresented.

The SPECjvm2008[4] focuses on kernel applications for both client and server side Java systems. The benchmark results are obtained by single-threaded as well as multi-threaded applications using multiple logical processor cores (physical cores or hardware threads). Even though the benchmarks require little I/O and no off-chip network communication they are not well-suited for embedded devices because the full J2SE standard API is required. The SPEC website highlights that the SPECjvm2008 uses real-world applications and also domain specific benchmarks (e.g., XML processing, cryptography). Furthermore, the benchmark results are influenced by the operating system and the hardware platform where the JVM executes.

The SPECjbb2005[4] benchmark emulates a three-tier business application. Multiple warehouse databases are implemented as Java Collection objects containing about 25 MB of data. So-called customer threads place new requests or status updates. Additionally, internal stock updates and payment reports are generated. The focus of this benchmark is the performance and the scalability of the middle-tier, which is stressed by the customer threads with new requests. Besides the JRE, no additional libraries or system capabilities are required.

Doherty [8] extended the SPECjbb2005 to the so-called SPECjbb2005rt. The modified benchmark measures the response times of the requests to the middle-tier and delivers a histogram. Besides the average time also the maximum observed response time is reported. This benchmark is intended to be used by JVM suppliers to demonstrate the real-time capabilities of their products. Of course, this benchmark is neither suitable for embedded systems with restricted resources nor can it be used to proof any hard real-time capability of a system.

All described Java SE benchmarks are too heavy for embedded Java. They need a full Java environment with file I/O and usually several MB of heap memory, which both is seldom available in embedded systems.

3.2 Embedded Java

The Embedded CaffeineMark [16] was introduced by the Pendragon Software Corporation. It is an adopted version of the complete CaffeineMark suite reduced to the following benchmarks: *Sieve*, *Loop*, *Logic*, *Method*, *String* and *Float*. The *Graphics*, *Image*, and *Dialog* benchmarks are omitted because they require a complex graphical interface that is not present on most small embedded systems. Embedded CaffeineMark is a typical collection of kernel benchmarks, which can give some guidance in performance improvement, but shall not be used for system comparisons.

GrinderBench[5] targets the mobile phone platform. It contains

[2] `http://www.ibm.com/developerworks/java`
[3] `http://www.kertasarie.de`
[4] `http://www.spec.org/jbb2005/docs/WhitePaper.html`
[5] `http://www.grinderbench.com/`

five different kernel benchmarks: *Chess*, *Crypto*, *KXML*, *Parallel*, and *Png*. Because the benchmark suite focuses on CLDC and CDC devices, only kernel algorithms are tested. Any kind of I/O is not considered. The algorithms are taken from typical mobile phone applications i.e., they are based on entertainment and not on industrial/real-time requirements.

Another benchmark suite targeting devices with restricted resources is the PennBench [7]. It offers 12 benchmarks that mainly represent complete applications like a calculator, an email viewer, a jpeg viewer, a video stream viewer, and a web browser. The MIDP capability of the target platform is required.

The Mälardalen Real-Time Research Center provides open-source C benchmarks as WCET analysis tool challenges [15]. Those benchmarks have been ported to Java by Trevor Harmon [10]. The benchmarks are basically kernel benchmarks to stress WCET analysis tools and not real applications. Many of them consist just of a single function. Furthermore, as the original code is in C the benchmarks are not object oriented.

The benchmark that is closest related to JemBench is CDx [13]. CDx is a collision detector implemented as test case for the RTSJ. Besides targeting the RTSJ, CDx is also available for standard Java. However, in both cases the JDK of Java SE is assumed. CDx contains a single periodic task that implements the collision detection. Another thread that generates the radar frame can be started to generate garbage to test real-time garbage collection.

4. THE JemBench SUITE

The JemBench benchmark suite is implemented in plain Java against a J2ME API, considering also the restrictions of CLDC 1.0 and the available library in SCJ. The classfiles of the benchmark are generated by a standard Java source compiler. It does not contain classfiles assembled or manipulated manually.

4.1 Benchmark Architecture

The performance of embedded systems can vary in the order of several magnitudes. As to extract valid results in a reasonable time with reasonable accuracy, the benchmark needs to adapt dynamically to the underlying platform. JemBench performs this adaptation by exponentially increasing the number of iterations of a benchmark until a minimum execution time threshold is exceeded. The performance is then reported based on this conclusive run and normalized to iterations per second. The greater the reported number is, the higher is the performance.

The JemBench suite includes several types of benchmarks. On a first level, sequential benchmarks can be distinguished from parallel ones. While the former focus on the evaluation of a single execution engine, the latter target systems with hardware support for parallel thread execution. Both groups of benchmarks are further subdivided by the structure of the workloads of the comprised individual benchmarks.

4.2 Sequential Benchmarks

The sequential benchmarks measure the performance of a single execution engine on different levels of abstraction:

Micro Benchmarks
: target individual bytecodes or short typical bytecode sequences;

Kernel Benchmarks
: comprise a tight computational kernel; and

Application Benchmarks
: execute a small application that is structured into multiple methods of several classes.

4.2.1 Micro Benchmarks

The micro benchmarks target the evaluation of the execution performance of single bytecodes or short bytecode sequences that cover all features of Java execution individually. The obtained results allow the comparison of the implementations of the same feature across different platforms. It is important to note that these benchmarks do not measure a workload that comprises any realistic instruction mix. Quite on the contrary, the employed measurement approach is particularly designed to extract the performance figures for the targeted bytecode sequence only. Even the necessary benchmark overhead is explicitly discounted from the measurement.

The micro benchmarks are useful for evaluation of Java processors or interpreting JVMs. A compiling JVM will optimize the loop kernel in a way that the results will be meaningless. The main motivation for the micro benchmarks stems from the intention to evaluate the optimization of individual bytecodes of the author's Java processors.

The micro benchmarks contained in the JemBench suite cover all relevant aspects of Java execution:

- arithmetic operations,
- array access,
- class and instance field access,
- branching (taken and not taken),
- static, instance and interface method invocations,
- type checking, and
- synchronized block and method executions.

Aiming at a measurement on the level of individual bytecodes, the micro benchmarks require a sophisticated measuring approach. In particular, this must discount the control overhead induced by the benchmarking itself, which would otherwise dominate the execution time. Thus, JemBench cannot simply execute the target bytecode within a loop but rather employs a full measurement loop and an overhead loop. The bodies of these merely differ in a minimal bytecode sequence containing the targeted instruction. The relevant measurement is then obtained as the difference of the execution times of these two loops. A small example illustrating this approach is shown in Figure 1 for the GETFIELD bytecode instruction.

The benchmarks calculate a result within the benchmark loop and the overhead loop that is returned from the method. Local optimization cannot optimize away the instructions in the loop. This is the very reason why the example of Figure 1 must also contain a GETFIELD bytecode in the overhead loop. While a plain assignment of t=t would certainly suffice, it would establish a rather obvious empty operation and, thus, an easy optimization target. This is prevented by the indirection through a heap object.

The micro benchmarks are very concise. Their execution exposes a great degree of locality. Therefore, the obtained measurements typically resemble best cases benefitting from code and data caching. We currently do not consider hotspot execution with dynamic code recompilation as a relevant technique for small embedded devices. It is, nonetheless, important to be aware of the fact this technique may not only introduce execution jitter but that very aggressive optimizers might even fail these simple benchmarks.

It has to be noted that the synchronized block and method benchmarks are executed in a single thread setting. A common technique in JVMs is to optimize the common case (no lock contention) and defer the full implementation of synchronization either when the

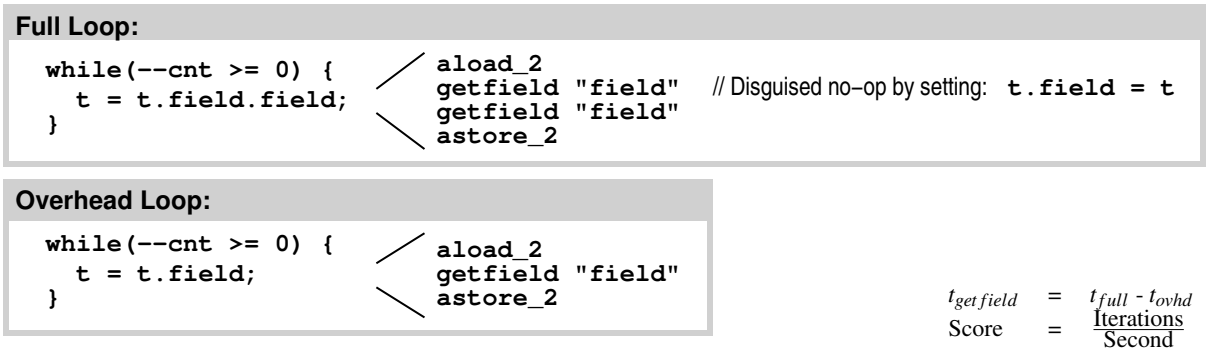

Figure 1: Discounting of the overhead in the measurement of a micro benchmark for the bytecode GETFIELD

lock is contended or when thread switching occurs. In that case, only the best case is measured.

4.2.2 Kernel Benchmarks

The workloads of these benchmarks consist of compact computational kernels. In contrast to the micro benchmarks, these kernels already implement full algorithms. Their focus of measurement is, however, the plain computational power observed for very basic algorithmic workloads. There is no noteworthy workload diversity within a particular benchmark. The kernel benchmarks are also structurally simple with a self-contained algorithmic loop. Lacking complex class hierarchies and invocation patterns, these kernel workloads are still far away from real object-oriented applications.

The performance measurement for the kernel benchmarks is implemented straightforward. The loop of the benchmark iteration counter is simply included in the time measurement as the isolated execution time of the contained kernel is of no particular interest.

The current JemBench suite contains two kernel benchmarks:

Sieve
: which computes all the primes below 100 using the Sieve of Eratosthenes, and

BubbleSort
: which initializes an integer array bitonically before sorting it by the bubble sort algorithm.

It has to be noted that kernel benchmarks shall only be used for some initial performance measurement. They are no substitute for complete application benchmarks. The issues with kernel benchmarks, such as Dhrystone or Whetstone, are discussed in a fallacy at p. 63 in [11].

4.2.3 Application Benchmarks

The application benchmarks aim at a realistic approximation of real-world workloads. Two of them are adapted versions of applications that are in industrial use [20]. They implement simple but already structured tasks. This is reflected in the executed code base, which is factored into several methods distributed over a few classes. Their overall complexity remains, however, far below that of standard desktop applications. As for the kernel benchmarks, the performance measurement is again based on the full runtime of the benchmark iteration loop.

The current JemBench suite contains the following application benchmarks:

Kfl
: In rail cargo, a large amount of time is spent on loading and unloading of goods wagons. The contact wire above the wagons is the main obstacle. Balfour Beatty Austria developed and patented a technical solution, the so-called *Kippfahrleitung*, to tilt up the contact wire. Each mast contains a tilt mechanism and a motor that is controlled by an embedded Java system. This was the first commercial project where JOP had to prove that a Java processor is a valuable option for embedded real-time systems. A simulation of both the environment (sensors and actors) and the communication system (commands from the master station) forms part of the benchmark, so as to simulate the real-world workload. The application is written in a very conservative, static programming style. That means, all methods and fields are static – no objects are allocated.

Lift
: Lift is a lift controller software that is installed in an automation factory in Turkey. The hardware is based on a JOP in a Cyclone FPGA device and an I/O print that was originally developed for the SCADA device TeleAlarm. The controller has just a few inputs (command buttons and input sensors for the hight measurement) and a simple motor control. The I/O devices are simulated in the benchmark environment. Compared to the Kfl benchmark, Lift is written in a more object oriented style, but still avoids to generate garbage at the mission phase.

UdpIp
: The third application benchmark uses an embedded UDP/IP stack that is in use in two industrial applications. The UDP/IP stack was developed to be time-predictable, to avoid garbage generation, and to consume minimal resources. In the version that is included in JemBench the UDP/IP stack is small enough to be executed on leJOS [24], a tiny interpreting JVM for the RCX robot controller from the LEGO MindStorms series. (The packet buffers and their size need to be reduced to fit into the 32 KB memory of the RCX controller.) In the benchmark setting, a client and a server exchange UDP messages via a loopback driver. UdpIp is, similar to the Lift benchmark, written object oriented, but avoids garbage generation. A pool is used to manage the packet buffers. As most code in the UDP/IP stack manipulates the packet buffer, the benchmark stresses integer array accesses.

All three benchmarks contain loop bound annotations and are WCET analyzable. Therefore, they are also used as test cases for WCET analysis of Java programs [21].

The application benchmarks of the current JemBench suite are all inherited from JBE[6] (version 1.1). JBE was first presented in [19], where JOP was compared with several embedded Java solutions. Since then, it has been used for performance evaluations in several papers by the authors and others. The results obtained for Kfl, Lift, and UdpIp are comparable with JemBench. It is the intention that future versions of JemBench will contain the unchanged source of all former benchmarks, so published results from different versions are comparable.

4.3 Multithreaded Benchmarks

There is an increasing support of multiple threads in hardware, either with chip-multithreading (CMT) or chip-multiprocessing (CMP). Therefore, multithreaded benchmarks for embedded systems are needed. Even small embedded Java processors provide support of real concurrency. The three Java processors JOP [17], SHAP [30], and jamuth [27] are all able to work in a multiprocessor configuration. To hide memory latency cycles, jamuth is additionally designed as a multithreaded processor.

All of the multithreaded benchmarks are new to the JemBench suite. Two classes of multithreaded benchmarks are distinguished:

Parallel Benchmarks
with a homogeneous workload, which is processed by a number of concurrent worker threads executing the same algorithm (data partitioning), and

Streaming Benchmarks
with a heterogeneous workload, in which each thread implements a different processing step on the same original data (process partitioning, pipelining).

4.3.1 Parallel Benchmarks

The homogeneous workloads of the parallel benchmarks are computations that can be parallelized easily. With a suitably fine granularity of subproblems, these benchmarks scale very well to an arbitrary number of worker threads. This allows the JemBench framework to spawn the computation to exactly one thread of execution for each execution engine[7]. These benchmarks, thus, enable the measurement of the performance of the complete system and its scalability and efficiency under full parallelization.

The parallel benchmarks of the JemBench suite are designed to stress the parallel processing. Nonetheless, they involve some synchronization and inter-thread communication in the distribution of the workload.

The measurements obtained from the parallel benchmarks are necessarily associated with a quantization error, which results from the subdivision into atomic serial subproblems. This error is bounded by the execution time of the subproblem completed last – potentially by the lone work of a single core. As to produce valid results, the fine-granular subproblem division is, therefore, an imperative countermeasure also taken by the benchmarks of the JemBench suite.

Currently, the JemBench suite comprises three parallel benchmarks:

Matrix Multiplication
which implements the multiplication of two integer matrices where the computation of each result row represents one subproblem.

NQueens
which computes all the solutions of the N-Queens Puzzle for $N = 13$. The search tree is divided into separate exploration problems after the successful placement of queens within the first four columns.

Ray Tracer
which uses ray tracing to compute the projective image of a three-dimensional scene containing two triangles and two balls. The calculated projection screen has a size of 2x3 pixels resulting in the maximum of six parallel execution threads. This benchmark is optional and requires the examined platform to provide floating-point arithmetic for doubles including implementations for the trigonometric functions sin and cos.

4.3.2 Streaming Benchmarks

Streaming benchmarks establish a more realistic scenario with a heterogeneous workload. Inter-thread communication should not dominate the computation but is, nonetheless, an essential part of these benchmarks for forwarding the data from one processing stage to another. The number of processing stages and, thus, threads working on a streaming benchmark is pre-defined. Consequently, these benchmarks do not scale to arbitrary numbers of execution engines but will force some cores to execute multiple threads or to run idle if the benchmark depth does not match the processor count.

The streaming benchmarks are currently represented by a single AES benchmark, which implements a four-stage pipeline:

1. The first stage generates well-defined pseudo-random data block,
2. which are AES-encrypted by the second pipeline stage before
3. being decrypted by the third.
4. The fourth stage, finally, computes a simple checksum over the received decrypted data.

This benchmark integrates the AESLightEngine from the Bouncy Castle Crypto Library,[8] which is available under a free open-source license.

4.4 Release Policy

JemBench is an extension of the benchmark collection JBE, which contains most of the micro benchmarks, one kernel benchmark, and the application benchmarks. JBE is available in version 1.0 and 1.1. As JemBench is a major update, the source is released under version 2.0 with the final version of this paper. For the review process the version 1.8 was available.

The intention of JBE and JemBench is to provide a platform and releases that stay comparable and compatible to earlier versions of the benchmark suite. Therefore, results with newer embedded Java platforms can be compared against results from earlier publications. Having release versions of JemBench allows us to extend and enhance the benchmark suite in the future.

JemBench is released in source form for easier handling and usage. However, to report results based on JemBench, the benchmarks shall not be changed, except adapting the number of processors for scalability measurements. Furthermore, the release version of the suite shall be included in the report.

We are very interested in collecting numbers for various embedded Java systems and have setup a Wiki page[9] to collect the results.

[6] `http://www.jopwiki.com/JavaBenchEmbedded`
[7] as reported by Runtime.availableProcessors()
[8] See the project website: `http://www.bouncycastle.org/`
[9] `http://www.jopwiki.com/JemBench`

Table 1: Platform support required by the JemBench suite

Basics

- Integer arithmetic (`int`, `long`).
- Classes: `Object`, `String`, `StringBuffer`.
- `System.arraycopy()`.

Reporting

- `System.currentTimeMillis()`
  – may be replaced within the execution framework.
- `System.out : java/io/PrintStream`.

Concurrency

- `Runnable`.
- `Thread` – may be replaced within the execution framework.

Particularities

AES
- `java.util.Random`.

RayTracer
- Standard double-precision floating-point arithmetic.
- `Math`: `sqrt()`, `sin()`, `cos()`, `toRadians()`.
- `java.util.Vector`.

The page is freely editable (after registering a user name to block spam bots).

5. BENCHMARK EVALUATION

It is not the intention of this paper to provide benchmark results of individual embedded Java platforms. We want to provide a set of benchmarks that can be executed on a great variation of platforms. Therefore, the usage of language features and the Java library needs to be restricted. As the minimum platform, we selected the CLDC 1.1 [26]. For a sanity check that JemBench only uses libraries that are available within the CLDC, we run the benchmarks on the Squawk JVM, a certified CLDC-compliant JVM from Sun/Oracle [23]. Furthermore, the benchmarks where tested on the Java processors jamuth, SHAP, and JOP.

Table 1 summarizes the library requirements a platform has to meet for the execution of the JemBench suite. Basic integer arithmetic and simple string processing must, of course, be available. JemBench further relies on System.currentTimeMillis() for its time measurement and on the standard printing routines invoked on System.out for the result output. While the default implementation also uses the standard Thread class to implement the concurrency for the multithreaded benchmarks, any other thread abstraction can be easily substituted for it as long as it can execute the Runnables provided by the concrete benchmarks.

Some particular benchmarks require some library support in addition to these basics. None of it goes beyond the capabilities of the CLDC 1.1. However, the floating-point support and the trigonometric functions required by the RayTracer benchmark were not required for the CLDC 1.0. Note that the AES and RayTracer benchmarks are also the only ones generating continuous garbage collection work.

6. CONCLUSION AND FUTURE WORK

In this paper we presented the embedded Java benchmark JemBench. The target systems for this benchmark are classic embedded systems, programmed in Java. As those systems are often resource constraint JemBench needs only the minimal Java standard (CLDC 1.1) to execute.

The benchmark suite contains micro benchmarks for measurements of JVM implementation details, kernel benchmarks that can run even on an incomplete JVM, real-world applications, and parallel benchmarks. The real-world applications are the core benchmarks to compare the performance of embedded JVMs. The parallel benchmarks are intended to evaluate possible speedups gained by multithreaded or multiprocessing platforms.

As future work, we intend to add adapted real-world applications that are multithreaded to the parallel workload. The single-threaded benchmarks can be execute on an RTSJ or an SCJ platform. We plan to add the needed wrapper classes for RTSJ and SCJ to the multithreaded benchmarks.

Furthermore, it is not yet clear how a real-time system can be benchmarked. The outcome of a benchmark for a hard real-time system is basically a boolean value: either all deadlines are met or not. We are working on a benchmarking methodology for those real-time systems. The idea is to measure the minimum available slack time in the worst case.

7. REFERENCES

[1] aJile. aj-100 real-time low power Java processor. preliminary data sheet, 2000.

[2] Aonix. Perc pico 1.1 user manual. http://research.aonix.com/jsc/pico-manual.4-19-08.pdf, April 2008.

[3] Austin Armbruster, Jason Baker, Antonio Cunei, Chapman Flack, David Holmes, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. A real-time Java virtual machine

with applications in avionics. *Trans. on Embedded Computing Sys.*, 7(1):1–49, 2007.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[5] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovik, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. Wake up and smell the coffee: evaluation methodology for the 21st century. *Commun. ACM*, 51(8):83–89, 2008.

[6] 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.

[7] G. Chen, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Pennbench: A benchmark suite for embedded java. In *in the IEEE 5th Annual Workshop on Workload Characterization*, 2005.

[8] Brian P. Doherty. A real-time benchmark for java™. In *JTRES '07: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 35–46, New York, NY, USA, 2007. ACM.

[9] Sudheendra Hangal and Mike O'Connor. Performance analysis and validation of the picojava processor. *IEEE Micro*, 19:66–72, 1999.

[10] Trevor Harmon, Martin Schoeberl, Raimund Kirner, and Raymond Klefstad. A modular worst-case execution time analysis tool for Java processors. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2008)*, pages 47–57, St. Louis, MO, United States, April 2008. IEEE Computer Society.

[11] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach, 3rd ed.* Morgan Kaufmann Publishers Inc., Palo Alto, CA 94303, 2002.

[12] Thomas Henties, James J. Hunt, Doug Locke, Kelvin Nilsen, Martin Schoeberl, and Jan Vitek. Java for safety-critical applications. In *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*, York, United Kingdom, Mar. 2009.

[13] Tomas Kalibera, Jeff Hagelberg, Filip Pizlo, Ales Plsek, Ben Titzer, and Jan Vitek. Cdx: a family of real-time java benchmarks. In *JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 41–50, New York, NY, USA, 2009. ACM.

[14] Andreas Krall and Reinhard Grafl. CACAO – A 64 bit JavaVM just-in-time compiler. In Geoffrey C. Fox and Wei Li, editors, *PPoPP'97 Workshop on Java for Science and Engineering Computation*, Las Vegas, June 1997. ACM.

[15] Mälardalen Real-Time Research Center. WCET benchmarks. Available at `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`, accessed 2009.

[16] Caffeinemark 3.0 benchmark. Available at `http://www.benchmarkhq.ru/cm30/info.html`, 1997.

[17] Christof Pitter and Martin Schoeberl. A real-time Java chip-multiprocessor. *Trans. on Embedded Computing Sys.*, accepted for publication, 2010.

[18] Filip Pizlo, Lukasz Ziarek, and Jan Vitek. Real time java on resource-constrained platforms with fiji vm. In *JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 110–119, New York, NY, USA, 2009. ACM.

[19] Martin Schoeberl. Evaluation of a Java processor. In *Tagungsband Austrochip 2005*, pages 127–134, Vienna, Austria, October 2005.

[20] Martin Schoeberl. Application experiences with a real-time Java processor. In *Proceedings of the 17th IFAC World Congress*, pages 9320–9325, Seoul, Korea, July 2008.

[21] Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40/6:507–542, 2010.

[22] Fridtjof Siebert and Andy Walter. Deterministic execution of Java's primitive bytecode operations. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01): April 23–24, 2001, Monterey, California, USA. Berkeley, CA*, pages 141–152. USENIX, 2001.

[23] Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White. Java on the bare metal of wireless sensor devices: the squawk Java virtual machine. In *Proceedings of the 2nd international conference on Virtual execution environments (VEE 2006)*, pages 78–88, New York, NY, USA, 2006. ACM Press.

[24] Jose Solorzano. leJOS: Java based os for lego RCX. Available at: http://lejos.sourceforge.net/.

[25] Sun Microsystems. *J2ME Building Blocks for Mobile Devices, White Paper on KVM and the Connected, Limited Device Configuration (CLDC)*, May 2000.

[26] Sun Microsystems. *Connected Limited Device Configuration Specification, Version 1.1*, March 2003.

[27] Sascha Uhrig. Evaluation of different multithreaded and multicore processor configurations for sopc. In *SAMOS '09: Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 68–77, Berlin, Heidelberg, 2009. Springer-Verlag.

[28] Sascha Uhrig and Jörg Wiese. jamuth – an IP Processor Core for Embedded Java Real-Time Systems. In *The 5th International Workshop on Java Technologies for Real-time and Embedded Systems - JTRES 2007*, Vienna, Austria, September 2007.

[29] Martin Zabel, Thomas B. Preusser, Peter Reichel, and Rainer G. Spallek. Secure, real-time and multi-threaded general-purpose embedded Java microarchitecture. In *Prceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, pages 59–62, Lübeck, Germany, Aug. 2007.

[30] Martin Zabel and Rainer G. Spallek. SHAP — scalable multi-core Java bytecode processor. Technical report, Fakultät Informatik, Technische Universität Dresden, 2009. `ftp://ftp.inf.tu-dresden.de/pub/berichte/tud09-13.pdf`.

APPENDIX

JemBench is available under the GNU GPL and can be downloaded from

`https://sourceforge.net/projects/jembench/`.

or with following svn command:

svn co https://jembench.svn.sourceforge.net/svnroot/jembench jembench

The driver class to execute all benchmarks is jembench.Main. Compiling and running JemBench on a desktop JVM is basically:

```
javac jembench/Main.java
java jembench.Main
```

By default floating-point operations are excluded as they are optional in CLDC 1.0. To include the floating-point benchmarks set the constant USE_FLOAT to true in Main.java.

On a CLDC platform it is not possible to query the number of processors. Therefore, the default configuration uses a single thread for the multithreaded benchmarks. The number of threads can be changed in Util.getNrOfCores().

Please help us to collect results on various embedded Java platforms and report the results on:

`http://www.jopwiki.com/JemBench`