Embedded Java – too fat and too slow?

Abstract

In software developer communities it is one of the most often discussed questions: “Is Java really to fat and too slow?”. For embedded devices this discussion is usually held very lively but largely without objective arguments. In this article we will first have a little look at the main reasons for using Embedded Java. After a brief introduction to the J2ME we present the “Embedded Java Assessment Suite” EJAS which was developed at 3SOFT GmbH in Erlangen. The results of some of the benchmarks may be quite astonishing when we have a look at a Java vs. C comparison of the Quicksort algorithm. After a glimpse at the structure of a JVM, its footprint and startup time, we conclude with the presentation of some coding and design guidelines for Embedded Java.

1. Why Embedded Java

In embedded environments we usually have very restricted resources concerning memory consumption, CPU performance and battery lifetime. Additionally there are often other restrictions like real time requirements. Considering this restricted environment one would normally not say that Java is the optimal language for programming such systems. But Java has a lot of features that make it very attractive:

- Technology:
    - JVM / platform independence
    - Security (features / API)
    - Dynamic
    - Memory Management (GC)
- Language:
    - Simple (similar to C / C++)
    - Object oriented
    - Elegant
    - Well known
- Project:
    - Time to market
    - SW quality
    - Development costs

So to choose whether or not to use Embedded Java is a very difficult trade off process. To ease this process a little bit we provide some fact and benchmark results later on.

2. J2ME

Since the normal desktop Java Runtime environment is much too big for embedded devices Sun Microsystems provides a version of Java, which has been scaled down to the needs of small devices.
The Java 2 Micro Edition – J2ME is intended to be appropriate for different kinds of embedded Devices, hence it is divided into two Configurations. Configurations are the fundamental components of the runtime environment. They are composed of a virtual machine and a minimal set of class libraries that enable them to provide the base functionality for a particular range of devices that share similar characteristics, such as network

connectivity and memory footprint. Currently, two J2ME configurations are defined: the Connected Limited Device Configuration (CLDC) and the Connected Device Configuration (CDC).

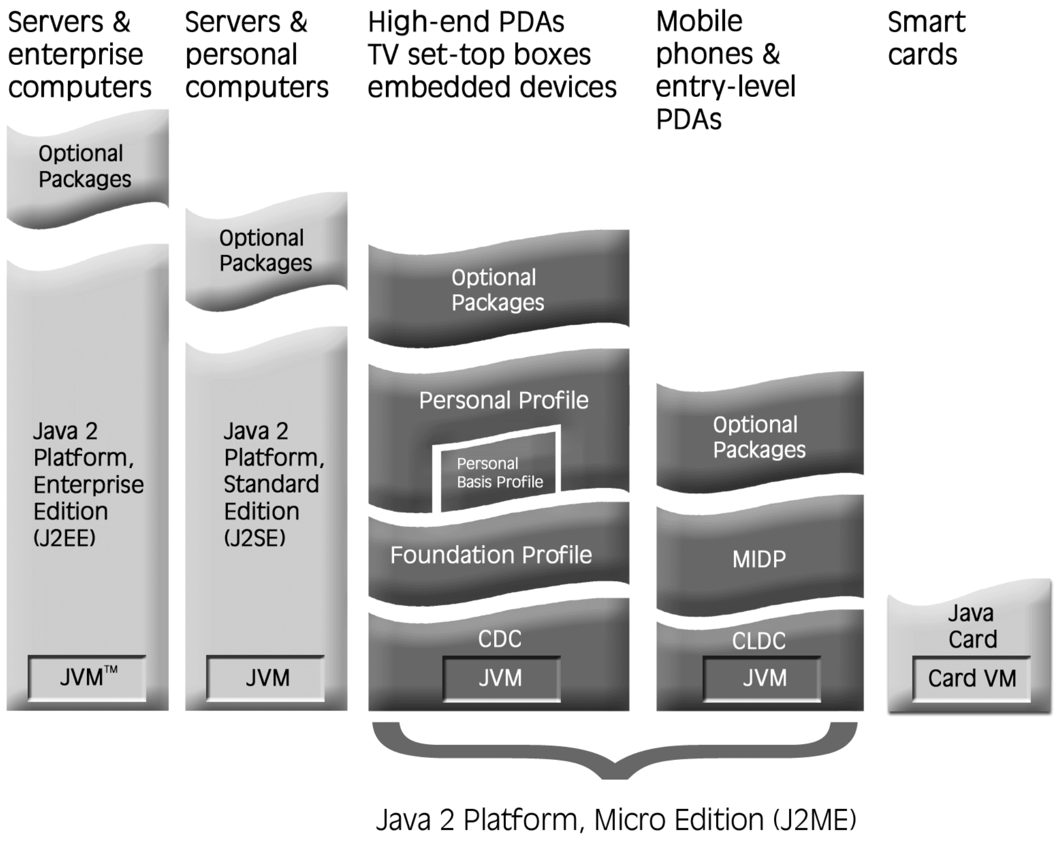
- CLDC

  CLDC is the smaller of the two configurations; designed for the devices with intermittent network connections, slow processors and limited memory – devices such as mobile phones, two-way pagers and PDAs. These devices typically have either 16- or 32-bit CPUs, and a minimum of 128 KB to 512 KB of memory available for the Java platform implementation and associated applications.

- CDC

  CDC is designed for devices that have more memory, faster processors and greater network bandwidth, such as TV set-top boxes, residential gateways, in-vehicle telematics systems and high-end PDAs. The CDC includes a full-featured Java virtual machine, and a much larger subset of the J2SE platform (81 classes for the CLDC 1.1 while 305 classes for the CDC 1.0) than the CLDC. As a result, most CDC-targeted devices have 32-bit CPUs and a minimum of 2 MB of memory available for the Java platform and associated applications.

Figure 1 The Java Platforms


Servers & enterprise computers
Servers & personal computers
High-end PDAs TV set-top boxes embedded devices
Mobile phones & entry-level PDAs
Smart cards
Optional Packages
Optional Packages
Optional Packages
Optional Packages
Java 2 Platform, Enterprise Edition (J2EE)
Java 2 Platform, Standard Edition (J2SE)
Personal Profile
Personal Basis Profile
Foundation Profile
MIDP
CDC
CLDC
JVM™
JVM
JVM
JVM
Java Card
Card VM
Java 2 Platform, Micro Edition (J2ME)

The configurations serve the basic operations. However, to provide a complete runtime environment, to further define the application life cycle model, the user interface, and the access to device specific properties, configurations must be combined with a set of higher-level APIs, or profiles. From the Figure 1 we can see that there are four profiles available in

J2ME: the Mobile Information Device Profile (MIDP), the Foundation Profile (FP), the Personal Basis Profile (PBP) and the Personal Profile (PP).

3. Java VM

Without knowledge about the runtime environment of Java programs it is not possible to argue why Java may be slow and what can be done to improve its performance.

3.1 Architecture of the JVM

All JVMs on the market have to obey the rules given in the Java Virtual Machine Specification [JVM_SPEC] from Sun Microsystems. But the JVM spec leaves a lot of room for optimizations as we experienced in the last years with the upcoming technology of Just In Time Compilation. Where the first JVMs just interpreted the compiled byte codes of Java class files the JIT-VMs transparently compile frequently used parts of the code and executes them natively.

The JVM defines various runtime data areas for the execution of a program. The life cycle and usage of these areas differs, the JVM level (created on JVM start-up and destroyed on JVM exit) and the thread level (created at the same time as the thread and destroyed on thread exit).

PC Register
Since Java has built in support for multithreading it provides one program counter for each thread.

JVM Stacks
Each Java virtual machine thread has a private Java virtual machine stack, created at the same time as the thread. A Java virtual machine stack stores frames. JVM stacks are analogous to the stack of a conventional language such as C: it holds local variables and partial results, and plays a part in method invocation and return. Frames may be heap allocated and the memory for a Java virtual machine stack does not need to be contiguous.

Heap
The heap memory of a JVM is shared among all threads. The heaps life cycle is JVM level; it is used to allocate memory for Java objects and arrays. The memory of the heap is reclaimed by the Garbage Collector GC.

Method Area
The method area is also shared among all threads in a JVM. It is used to store per-class structures such as the runtime constant pool, the code for methods and constructors. Although the method area is logically a part of the heap, how to implement the location of it or the policies used to manage compiled code is up to VM vendor and leaves space for optimizations.

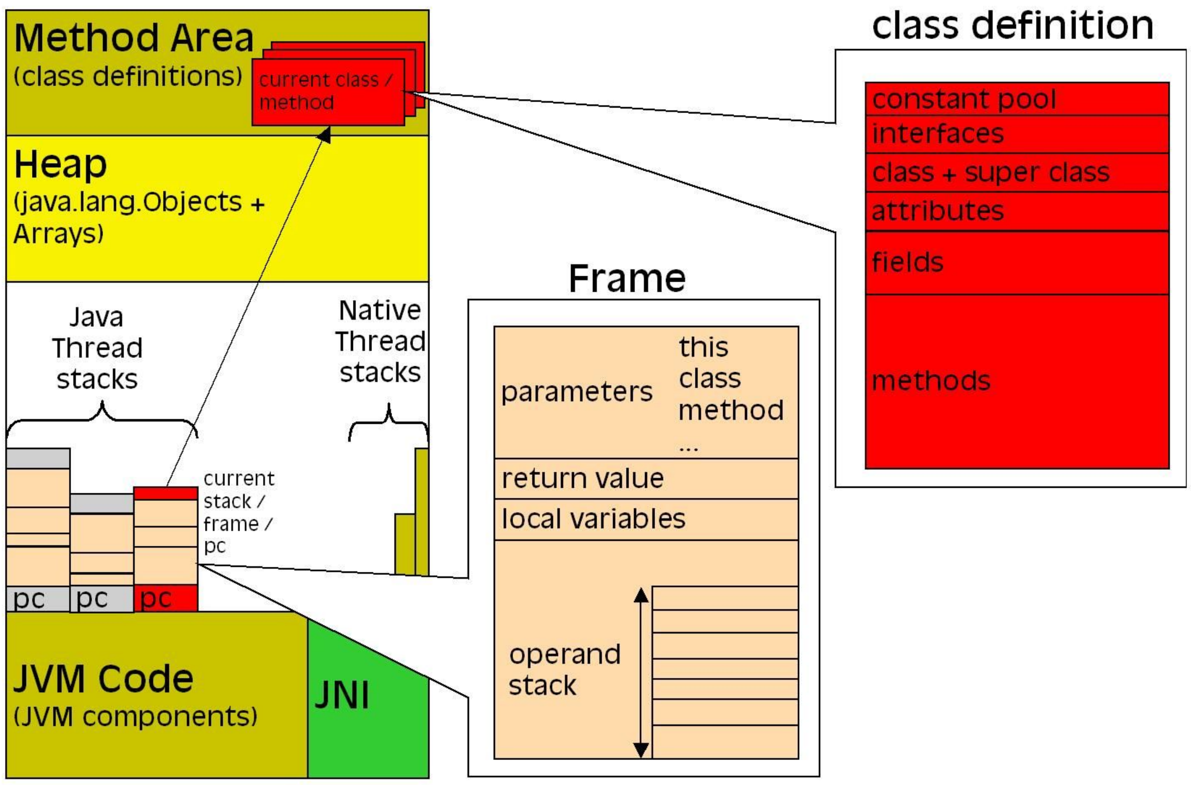


Figure 2 JVM Structure

3.2 Startup time and runtime

The startup process of a Java Virtual Machine is dominated by three major tasks before the `main()` method of the Java program is invoked. The `main()` method drives all further execution.

Loading
If there is no class (native or byte code) in the JVM present, it uses a Class Loader to load the desired class.

Linking
Before a class can be executed it has to be linked. Linking is the process of taking a class or interface and combining it into the runtime state of the Java virtual machine so that it can be executed. The linking process has three phases: 1. *Verification* to ensure the programs binary representation is structurally legal. 2. *Preparation* to create static fields and initialize them. 3. *Resolution* to dynamically determine the concrete value of symbolic references in the runtime constant pool.

Initializing
The classes' static initializers and the initializers for static fields declared in the class are executed.

Of course all the work done at the startup time of a JVM and at each time a class is loaded dynamically costs some time. And of course this is one reason why Java is widely considered to be slow. The startup time for the JVM itself also depends on what features shall be used. The usage of a JIT compiler does not only have memory cost but also slow down the time until the first command of the main method can be processed. But there are techniques to reduce the consumed startup time. With an appropriate JVM some of the described tasks can be done at compile time:

- *Pre linking*: saves some time of the linking process
- *Pre verification*: no code verification has to be done at startup
- *Ahead of time compilation*: reduces the time consumed by the JIT compiler
- …

3.3 Footprint

The memory footprint of Java runtime environment depends of course on the numbers of class libraries and features supported by the environment

Item	**Used Memory**
Java Class Libraries Configuration + Profile (platform abstraction + helper classes)	iPAQ e.g. J9 CLDC/MIDP 2MB
JVM .JIT / HotSpot / Interpretation Class Loading Memory Management Object Creation Array Handling Synchronization / Threading Exception Handling Casting / Reflection	iPAQ e.g. J9 VM 325KB CLDC +1MB JIT, +50KB JNI

Table 1 Example footprint of J9 on an IPAQ

To reduce the memory footprint of a runtime environments there are several possibilities. One is to remove all not used classes and methods from the libraries and the code itself. This can be done by “stripping” tools automatically. Of course dynamic class loading is very hard with that because not all methods defined in an interface may be present at runtime.

4. EJAS – Embedded Java Assessment Suite

The EJAS project was started to serve the following aims:

- Performance benchmarking
- Acquire know how for optimization
- Measurements of memory consumption
- Functionality checks

The architecture of EJAS makes use of the Plug-In Pattern and the External Configuration Pattern. The assessment suite has a “core” part, which is always the same regardless of the current runtime environment. The used assessments can be plugged into the core and can be configured using an external configuration file. The output of the assessments can be received through various Outputter objects, as there is no command line interface on most embedded devices. Therefore EJAS is very flexible and can easily be adapted to serve newly developed benchmarks or unforeseen environments.

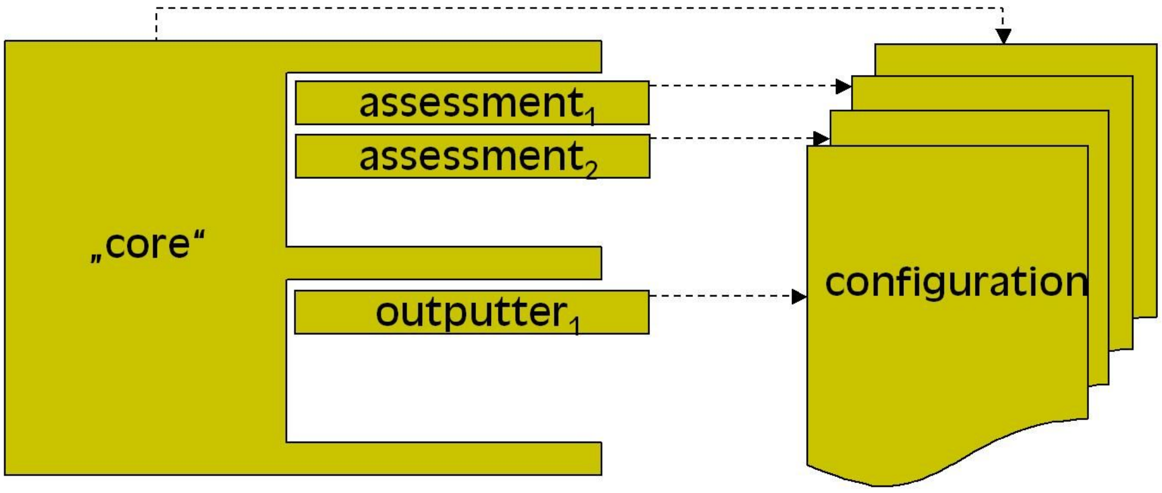
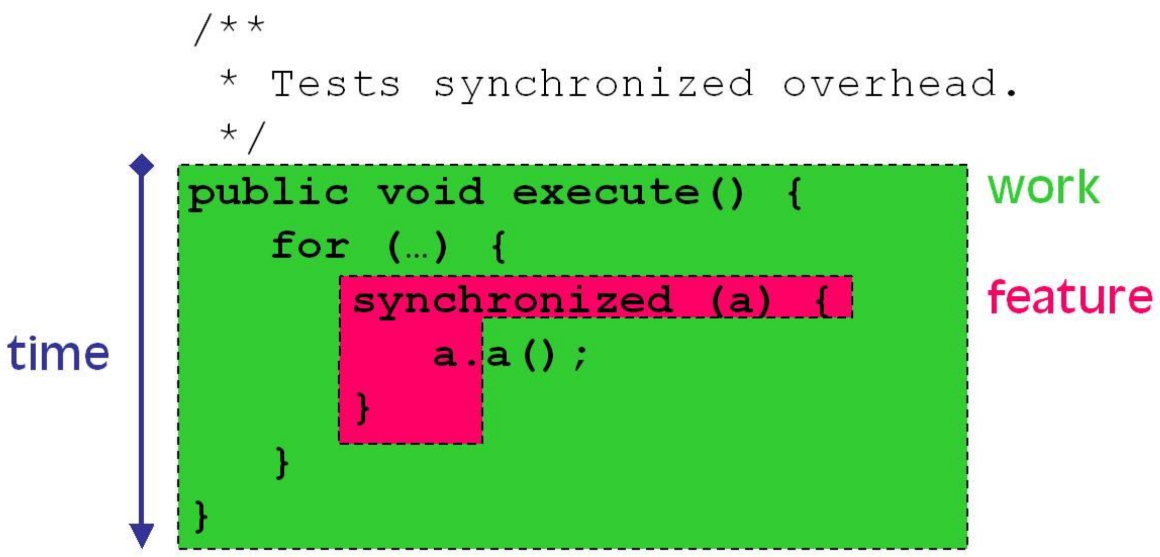


Figure 3 EJAS Architecture

In EJAS the “time per constant work” type of benchmark is used. That means the time consumed for performing a certain work is measured on different platforms.
Other possibilities for benchmarks would be “work per constant time”. This type is rather difficult to use for objective measurements because a second thread is needed to stop the working thread after the constant time has past by. This stopwatch thread of course influences the measurement result. A third kind of benchmark is to measure the time consumed by a certain feature like synchronized code in Figure 4. One could obtain the time spent for the marked feature (synchronized block) by first quantifying the time needed for the green part and afterwards subtracting the time consumed for the same loop without the synchronized block.



```
/**
 * Tests synchronized overhead.
 */
public void execute() {
    for (…) {
        synchronized (a) {
            a.a();
        }
    }
}
```

Figure 4 Simplified benchmark code

The benchmarks executed with the assessment suite can be classified in three kinds:

- **Low level operations** (operators, array handling, assignments, casting, I/O, loops, Math operations, method calls, thread handling)
- **Algorithms** (Triple-DES, FFT, RSA, matrix multiplication, sorting)
- **Domain Applications** (Address book, games, calendar)

The result of the benchmarks on the three different platforms for the EJAS benchmark for low-level operations can be found in Table 2. But always be careful to use the results of one test for a general statement, which JVM is the fastest. Usually the assessment result is very dependent on the hardware, the OS, VM version and operations carried out.
The benchmark results are the higher the better.

System-Type	**Hardware**	**EJAS Benchmark (low-level op‘s)**
Desktop PC	Pentium 3, 32 bit, 650 MHz, 512 MB RAM, 5400 rpm HD, Windows XP, J9 JVM, J2SE	JDK 1.3.01: 11.342.000 J9 5.1: 77.880.417
iPAQ	H3900, Intel PXA250, 32-bit, 400 MHz, 64 MB DRAM, 32 MB Flash, Windows CE, J9 JVM	912.875
P800	ARM9, 32 bit, 156 MHz, 12 MB RAM, KVM (?) JVM integrated in Symbian OS 7.0, CLDC 1.0 / MIDP 1.0	309.126

Table 2 Benchmark Results for low-level operations

4.1 Java vs. C

For the Java vs. C comparison an algorithmic problem has been chosen. 1.2 million random integer numbers had to be sorted using a quick sort algorithm. The source code of the C and the Java Program are very similar except for the file IO part of course. Here the results are also very depended on the used JVM.
The following results in Table 3 are represented in milliseconds used for reading the numbers from a text file and afterwards sorting them on a Desktop PC (P4 1,6 GHz, Win XP SP1). In order to reflect the performance effects of just in time compilation the test has been done five times. The C benchmark has been compiled with Microsoft Visual C++ 6.0 compiler with standard optimization options enabled.

Round	**C**		**JDK 1.4.2**		**IBM J9 5.1**	
	File IO	Algorithm	File IO	Algorithm	File IO	Algorithm
1	141	422	375	547	328	750
2			344	485	313	453
3			344	485	297	438
4			344	562	266	469
5			328	500	281	406

Table 3 Results Java vs. C: Quicksort

The results of the Java versus C benchmark are displayed in Table 3. It is obvious that in the first round the J9 takes some extra time compared with the Sun JVM to do some optimization that improves the sorting later on. Comparing the fastest Java round with 406 milliseconds with the usual C result of 422 one can see that doing pure algorithmic work without IO is not really slower in Java, sometimes even faster.
Also obvious is that File IO in Java is about two times slower than in C with the tested JVMs.

5. Assessment results

5.1 Guidelines for Embedded Java

As a first conclusion from the assessments made with Java on different platforms we would like to present some guidelines for the usage of specific features of the Java programming language and runtime environment presented in the following table.

Feature	**Direction**
Classes associations & hierarchies	Use it – but prevent dynamic method dispatching if optimization is necessary
Interfaces	Use it – optimize away if optimization is necessary for non-hot-spot
JNI	Use it if necessary or if tests say its worth it: Down-calls (J to C) are much faster than up-calls (C to J)
Long names	Use it – prevent dynamic method dispatching
Objects	Optimize here, use as few as possible
Method calls / granularity	Use it – optimize if necessary
Reflection	Use it only if necessary

Table 4 Feature guidelines for embedded Java

5.2 Benchmark results

Afterwards we present the results of some measurements made on a Win XP system with a P4 1,6 GHz processor. To have a comparison all benchmarks have been carried out on the JVM delivered with the Sun JDK 1.4.2 and on the J9 5.1 from IBM. In the following tables there is a column for the used JVM, one column for the ratio: (time with feature) / (time without feature) and one for T_F the time consumed one execution of the feature. Also the clock cycles of the CPU consumed for one execution of the feature are displayed. The clock cycles and T_F are average values of about 10^9 executions.

Please always keep in mind that the presented values have only example character. These results do **not** allow a generalized statement like “This JVM is faster than that.”; they are only spot checks.

Assessment results for exceptions:

Exceptions are an error-handling feature of Java. In theory the usage of exceptions should only consume time in the error case, which means catching an exception. In reality each try-catch clause costs time. Exceptions can gainfully be used for error handling where exceptions should be thrown only in “bad paths”. Assessment results can be found in the following table:

Feature	JVM	Ratio	T_F	Clock cycles used for the feature
Exceptions (good path)	J9	3,98	4,21ns	6 cycles
Exceptions (good path)	Sun	9,81	34,38ns	55 cycles

Table 5 Assessment results exceptions

Assessment results for synchronization

Synchronization is a built in Java feature for mutual exclusion of threads in a multithreaded environment. Synchronization is done via the keyword `synchronized`. The results are displayed in Table 6.

Feature	JVM	Ratio	T_F	Clock cycles used for the feature
Synchronization	J9	11,5	18,125ns	30 cycles
Synchronization	Sun	5,6	29,093ns	46 cycles

Table 6 Assessment results for synchronization

Assessment results for type casting

Type conversion in Java is done by using the so-called cast-operator `(Type)`. The cost of casting one object to another type is represented in Table 7.

Feature	JVM	Ratio	T_F	Clock cycles used for the feature
Casting	J9	1,112	0,406ns	1,6 cycles
Casting	Sun	1,42	1,640ns	2,6 cycles

Table 7 Assessment results for casting

Assessment results for array handling

Using arrays in Java is always linked to creating new objects on the heap memory, as all arrays are objects in Java. The main direction here is to use arrays but to prevent polymorphic array objects. See results in Table 8.

Feature	JVM	Ratio	T_F	Clock cycles used for the feature
Array access (int)	J9	3,2	1ns	0,6 cycles
Array access (int)	Sun	1,02	0,62ns	1,5 cycles
Array access (Object)	J9	1,32	6,56ns	10,4 cycles
Array access (Object)	Sun	1,6	37,5ns	60 cycles

Table 8 Assessment results for array handling

Accessing the elements of an array using a for-loop is very often used in Java programs. The following code snippet displays a common for-loop (see code 1):

Code 1	Code 2 (manually optimized)

`for (int i = 0; i<array.length; i++) {`
`      doWork();`
`}`
	`int l = array.length;`
	`for (int i = 0; i<l; i++) {`
	`      doWork();`
	`}`

In may Java performance guidelines we find the advice to improve the performance of for-loops in the way shown on the right side (see Code 2). But whether or not this is really an improvement for a specific JVM can be found out by a benchmark. In Table 9 the time consumed for loops accessing arrays with 10 million elements are displayed. We can see that that the hand-optimized code (code 2) has nearly equal time consumption on both JVMs whereas the "normal" code (code 1) has different results. The J9 behaves like expected and is a bit slower with the not optimized code (code1). The Sun JVM seems to have some special optimization for these loops, which causes it to perform better than the optimized code.

Feature	JVM	Time (code 1) (10^9 elements)	Time (code2) (10^9 elements)
Optimized for-loop	J9	250ms	203ms
Optimized for-loop	Sun	188ms	203ms

Table 9 Assessment results for-loop optimization

6. Conclusion

Using the Embedded Java Assessment Suite EJAS has produced interesting results. It is possible with this suite to judge the performance and memory consumption of language and JVM features on specific platforms. With some of the benchmark results we saw that Java reaches about the same performance as C in algorithmic tasks like sorting where no Java native interfaces have to be used. Another very important aspect of the overall results of the benchmarks is that an optimized JVM is one of the most important ingredients for an Embedded Java system with good performance.
As a summarized result of our assessments we can say that all JVMs have great performance differences depending on the platform they are running on. So for projects using embedded Java we have two advices:

- It is a good idea to do some performance before choosing a platform and a JVM.
- Tailor your assessments to the needs of your project to gain significant results.
- After the platform decision is made: provide programming guidelines to your developers based on assessments carried out on the selected platform with the selected JVM.

References

[JVM_SPEC]	Java VM Specification http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html