The Kertasarie VM

S. Gatzka, Th. Geithner, Chr. Hochberger
geith@informatik.uni-rostock.de
[Stephan.Gatzka, Christian.Hochberger]@inf.tu-dresden.de

University of Rostock, 18059 Rostock, Germany
TU Dresden, 01062 Dresden, Germany

Abstract. Currently, Java in traditional embedded systems is not very widely adopted due to the required resources and the lack of realtime capabilities. The Kertasarie VM was designed to get a feature rich implementation with a small memory footprint. In this paper we present a typical scenario from which we derive the requirements for a Java virtual machine implementation. The concept and features of the Kertasarie VM are presented in detail. It is related to other VMs by a comparison of speed and code size. Overall, the Kertasarie VM turns out be a very good choice for Java in embedded systems.

1 Introduction

The programming language Java is widely accepted for software development in desktop and server systems. Yet, there are areas where Java has even started to appear (e.g mobile phones) and other areas where Java does not play any role at all (e.g. traditional embedded systems). These critical environments are typically characterized by resource or timing constraints. In this contribution we will discuss the problems that arise if Java is used in this type of environments and we will show that nevertheless Java can be used with full functionality on such systems.

Java as a platform for applications and software systems can be categorized by four different aspects:

- The language definition. It specifies the syntax and semantics of the language elements like data types and language constructs.
- The API definition(s). It is important that a modern language concept does not only come with a syntax and semantics of the language itself, but rather also defines a minimal set of supporting functionality that can be used by the programmer independent of the operating system and processor architecture on which the program runs. In recent times this very general claim has been replaced by a more subtle view which is explained in section 1.2.
- The definition of an abstract intermediate program representation. This so called bytecode is intended to run on a virtual machine and thus does not contain any platform dependencies.
- The Java Virtual Machine itself. Many different implementations provide a very different spectrum of code size, execution speed and functionality.

The usage of Java as a programming language in various environments has always been a question of the availability of a suitable virtual machine implementation. First generation Java implementations were very memory consuming since all possible Java features were implemented in the JVM. Thus, these implementations were far to heavy for small systems. The Java 2 standard aimed at resolving this conflict by specifying different editions of Java for different target systems, J2EE (Java 2 enterprise edidition), J2SE (standard edition), J2ME (micro edition) and J2CE (card edition).

Through the remainder of this paper we will focus on the Java 2 Micro Edition, but we will consider J2SE or J2EE aspects where applicable.

1.1 The Java Programming Language

The great success Java had in the last ten years has many reasons. Some of them are related to the language itself, whereas others are more related to non technical issues. In this paper we would like to discuss just three of them:

- Object orientation. Java enforces strict object orientation and thus programmers have to adapt this design methodology for their own projects. This methodology comes with a number of positive side effects. Firstly, it supports a high degree of reusability of the code. Secondly, it permits a better distribution of the development process in a group.
- Program stability. Java uses an implicit memory management scheme (i.e. garbage collection) and also implies a consequent usage of exception handling. Both features make Java programs very stable even in the early stage of development.
- Platform independence. Java programs are translated into a machine independent code. This bytecode is the form in which Java programs are shipped in most cases. It is executed on the target machine by a Java Virtual Machine (JVM). JVM implementations exist for almost any operating system and almost any processor architecture. Especially, in the server and desktop area very good implementations are available that provide equal if not better performance than other languages do.

1.2 Java in Embedded Systems

Although many good reasons promote the usage of Java in software development projects, it has taken a considerable amount of time before Java appeared in embedded systems. The main reason for this delay was the lack of good JVM implementations that fit the requirements of embedded systems.

Configurations and Profiles With the introduction of the J2ME version of Java things changed dramatically. The main improvement of this specification was the introduction of configurations and profiles. These configurations allow vendors to develop different JVM implementations that are suitable for a wide spectrum of embedded systems. In J2ME the following two configurations are specified:

– Connected Limited Device Configuration (CLDC). This configuration is intended for small devices with very limited resources. Thus, a major part of the J2SE functionality is optional and need not be implemented to simplify the internal structure of the JVM. Together with the omission of the corresponding API classes this leads to a great reduction of memory requirements. Yet, these omissions concern such important things like sockets, reflection and serialization. Typically, the overall memory requirement of a running Java application should be well below one MByte of memory.
– Connected Device Configuration (CDC). This configuration is intended for small devices with less resources than typical desktop systems. The major omission in this configuration is a full featured GUI like AWT or SWING. Also, JNI is optional and may be left out. Typically, the overall memory requirement of a running Java application should be in the range of one to eight MByte.

Profiles refine the functionality that comes with a JVM of a certain configuration. One of the most popular profiles for the CLDC is the Mobile Information Devices Profile (MIDP), which is intended for devices like mobile phones or organizers. Typical profiles for the CDC are the RMI and Foundation profile. The RMI profile is intended for networked devices without graphical capabilities (e.g. telecommunication equipment). The Foundation profile extends the RMI profile with basic GUI functionality and thus can ideally be used in set top boxes or comparable devices.

Resource Requirements Memory is only one of the constrained resources in embedded systems. Other resources also have to be treated carefully. Electrical power is an important resource, especially in mobile systems. Computing power is typically very much smaller than in desktop and server systems. Communication is restricted in most cases. Bandwidth as well as the number of concurrent sockets are much lower. Other operating system resources may also be limited.

It should be noted, that configurations and profiles only deal with the amount of memory. Other resources are not taken into consideration.

Realtime Capabilities Since environments of embedded systems are often time critical, embedded systems have to react in realtime to external events. Realtime means that the system must produce a response in a predefined, guaranteed time interval. Realtime systems are often categorized into soft and hard realtime systems. Hard realtime means, that it is fatal for the system if any response deadline is missed. Soft realtime means, that only the overall system performance is degraded if occasionally a response deadline is missed.

Many Java implementations are not realtime capable and can therefore not be used in time critical environments. A more thorough discussion of these aspects is given in section 3.4.

1.3 Related Work

Many JVM implementations for embedded systems have been developed in recent times. The KVM[14] and CVM[13] are the CLDC and CDC implementations of Sun

Microsystems. The J9 VM by IBM[7] implements both configurations by dynamically loading the native functionality according to the configuration. Several smaller companies offer J2ME implementations: Aicas offers the Jamaica VM, Insignia offers the Jeode VM. Esmertec offers the Jbed VM. Throughout this paper we only compare the Kertasarie VM with Suns and IBMs implementations. Other implementations were not considered since they either have a much worse performance or cannot compete in terms of features.

The usage of middleware is also discussed in this paper. Although we only discuss RMI, other technologies should be noted here: CORBA[11] is a very general approach to middleware and can be used on almost any operating system and with many programming languages. Yet, it is not as closely integrated into the language as RMI. SOAP is another base technology for middleware, but it has severe drawbacks like very high bandwidth requirements and latencies[5].

1.4 Paper Outline

After this introduction, we present a real life scenario to discuss the practical requirements in the next section. The Kertasarie VM is presented in section three, as well as a comparison with other popular JVMs for embedded systems. Section four shows the future development of the Kertasarie VM and section five gives a short conclusion.

2 Practical Requirements

This section provides the reader with a thorough discussion of the practical requirements of Java implementations in arbitrary types of systems or system components. We present a scenario for the usage of Java in order to discuss arguments on a real life example.

2.1 Scenario

The presented scenario takes place in a distributed environment since networking has become very popular in embedded systems in the past and will become even more popular in the future. The scenario is depicted in figure 1.

It shows a production facility in which many machines run concurrently. All these machines consume electrical energy which is produced at a power plant. One of the common price models for this energy is that the consumer has to pay according to the peak energy consumption that occurs in a given time interval (a week or a month). Thus, it is very interesting for the production plant manager to keep this peak consumption as low as possible. For this purpose each machine is equipped with a small networked controller (NC). This controller monitors the amount of energy that is consumed by the machine and it also controls basic operation of the machine.

The overall power consumption of the facility can now be controlled by the facility server. The policy of the server can be controlled by the manager by tuning the parameters of the software on the facility server. Moreover, the manager can control and monitor the power consumption of each individual machine. This type of access can

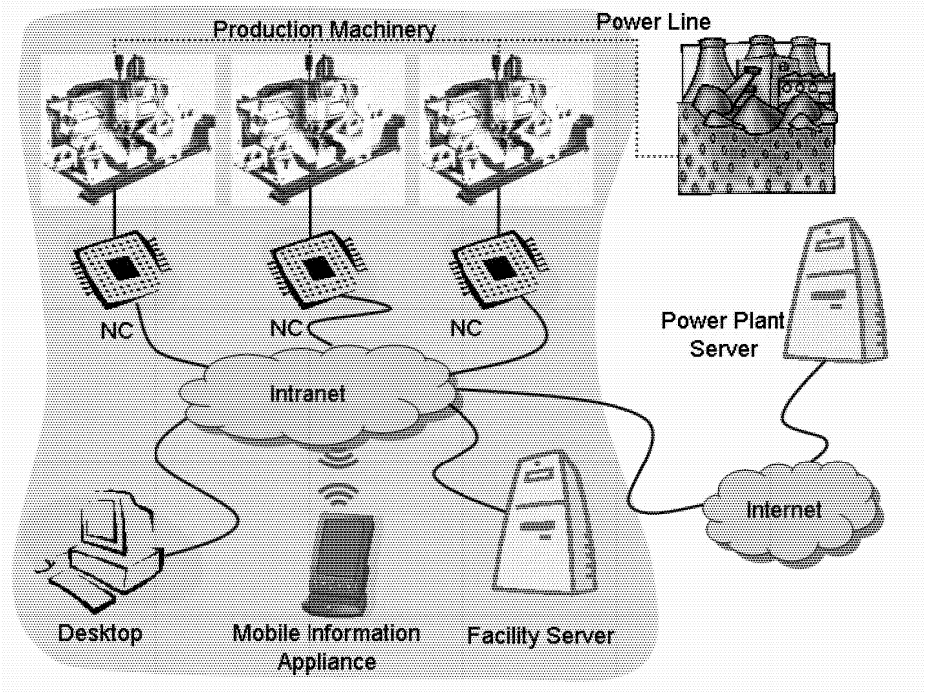

Fig. 1. Distributed power management in a production facility

be done from normal desktop systems connected to the intranet or it can be done with mobile information appliances through a wireless network.

To further extend the options of this scenario, let us assume, that the facility server also communicates with a server of the power plant. In this case it would be possible for the power plant to offer excessive power to the facility at low cost in times, when the overall power consumption of the power plant is lower than planned. The production facility could increase its productivity temporarily to use such offers.

2.2 Implementation Issues

Distributed environments like the presented scenario require a thorough engineering of the overall software architecture. In the following we will discuss some of the implementation issues of the presented scenario.

Object Orientation An object oriented approach to this problem is strongly indicated, since many different parties will participate in the development process. Using a distributed object approach will greatly simplify the development process, since accessibility of attributes and methods can be controlled very well. Java has established itself as a programming language for distributed objects, due to its many network related API classes.

Middleware Communication between the different systems should not be carried out with proprietary protocols, but should rather use some abstract mechanisms. Typically, such an abstraction layer is called a middleware. Using Java as programming language, RMI is the preferred middleware technology, because it is seamlessly integrated into the language[1] (see section 1.3).

Realtime One further aspect of the presented scenario should be noted. Controlling of machines often requires realtime capabilities of the controlling hardware/software. Thus, the software of the networked controllers must be realtime capable.

2.3 Device Properties

The devices participating in this scenario all have different properties. Server and desktop systems offer enough memory and processing power, that they can be implemented using the enterprise or standard edition of Java. The mobile information appliances and the networked controller have only limited resources so they need the Java 2 micro edition. Typical mobile information appliances have available memory of 16 to 64 MByte and a small display. In contrast, typical controllers have considerably less than 16 MByte. Also, 16 bit processors are still used in these systems.

3 The Kertasarie VM

3.1 History of the Kertasarie VM

The development of the Kertasarie VM started in April 2000. Initially, the project started as a "'proof of concept"' study. The aim was to implement a web based management of a telecommunication equipment with less than 500 KByte of memory (including native code and runtime memory). The communication between the browser and the JVM inside the embedded system should be done using RMI.

At that time no JVM implementation that offered RMI was embeddable with such a small memory footprint. Thus, we decided to write our own implementation of the VM and to develop our own variant of RMI.

In the meantime, the development focus changed from an extremely small memory footprint to higher performance.

3.2 Supported Features

In the field of embedded systems usually the goal is to handle a given task with minimal expenses. In general, the resources (processing power, memory) will be limited to the required minimum.

The differences between the various Java editions (CLCD, CDC) do not only affect the API but also the complexity of the underlying virtual machine. For example, some

[1] With RMI you can use remote objects similar to local objects. The effort for requesting remote objects is very small.

JVMs, which implement the J2ME/CLDC specification, do not offer features like sockets, serialization and reflection, which will be discussed in the following. In contrast, the Kertasarie VM implements all features that are required for a J2ME/CDC respectively J2SE virtual machine.

Sockets The scenario presented in 2.1 shows a networked environment. The package `java.net` defines a standard way to access TCP/IP communication. It is part of the J2EE, J2SE and of the J2ME/CDC, but missed in the J2ME/CLDC. This package offers the possibility to communicate over TCP connections. On top of this, it is easy to implement higher level protocols like HTTP, FTP or proprietary protocols.

Reflection Two additional features (serialization and reflection) are required to use middleware. Reflection offers the possibility to deal with Java objects of unknown type. It provides access to all fields and methods of an object with information about their according types and attributes.

Serialization The second required feature for middleware systems, serialization, offers the possibility to transform a Java object from a running application into a byte stream. This stream can be saved into a file, stored in a database or transfered over a network connection. Later, the byte stream can be reconverted into a Java object which will be integrated into a running Java environment.

Serialization offers an easy way of communication between Java applications. It is efficient because it does not need to parse complex protocols and data (e.g. HTTP/XML) on Java level and it is not necessary to develop proprietary protocols.

Together, sockets, serialization and reflection, provide the basis for a middleware like RMI.

Other Features Originally, the Kertasarie VM was designed for classical embedded systems without user interfaces. But some applications require an user interaction, e.g. remote control of processes from mobile devices like PDAs. For such applications, the Kertasarie VM optionally offers a simple GUI API. It uses only few essential GUI elements (windows, menus, buttons, input fields etc.). It makes use of the system GUI elements to reduce the consumption of system resources, e.g. the buttons of the Win32 API on Windows CE. In contrast to other GUI APIs like AWT or Swing, the Kertasarie GUI API does not guarantee the usability of one interface design on different platforms without modifications. But this solution has some important advantages: Java applications look like native applications and it is very small and very fast.

One additional feature is an optional integrated remote debugger. With it, Java applications can be debugged inside the target system over a TCP connection. It is possible to control the debugger with a telnet client or a graphical front end application, written in Java (requires J2SE).

3.3 TinyRMI

The Kertasarie VM comes with its own RMI implementation. The development goals of TinyRMI are similar to the ones of the Kertasarie VM: minimal size and high performance. TinyRMI is source compatible to the RMI reference implementation from Sun Microsystems. It runs on the Kertasarie VM and on J2SE (tested with the Sun implementation).

TinyRMI uses an abstract transport layer which implements the major part of the transport functionality. Specific implementations for different transfer methods are easy to implement. Currently, transport layers for TCP and UDP are implemented. But the transport is not limited to IP based techniques. In the future, a CAN implementation is planned (see 4.2).

By comparison with Sun RMI, TinyRMI has some essential advantages. It is significantly smaller (a quarter of size) and faster[3].

3.4 Realtime Capabilities

Thread model Java threads can be implemented in a software VM in two ways: using the operating systems (native) threads or implementing threads inside the JVM (green threads). Each model has its advantages and disadvantages.

Green threads require the execution engine to check for thread scheduling in each interpretation loop pass. However, the porting effort for green thread written in ANSI C is much less then the adaption to each new operating system and not every operating system provides threads. In fact, the standalone version of the VM as described in section 3.5 is only possible by implementing threads inside the virtual machine. The Java execution performance could be slightly faster using native threads because the overhead to check for thread scheduling is dropped. The Kertasarie VM applies the green thread model for one more reason: to gain maximum control over thread scheduling.

The thread scheduler inside the Kertasarie VM implements a strict *highest priority first* scheduling. A runnable thread with a high priority prevents a runnable thread with a low priority to be scheduled. This is the best strategy to guarantee realtime response for important tasks, but allows an unexperienced programmer to produce deadlocks[2].

Priority inversion/inheritance The simple but powerful scheduling mechanism may introduce problems if an application consists of more than two threads. Threads with low priority can prevent high priority threads from running (priority inversion). A solution for this is priority inheritance[12]. The thread synchronization mechanisms (monitors) inside the Kertasarie VM are implemented using priority inheritance.

Garbage Collection Garbage collection is often another problem concerning the realtime capabilities of a JVM. Garbage collection (GC) cycles are very time consuming (up to several seconds depending on processor speed and the memory currently used)

[2] The Java Language Specification[10] enforces the Java programmer not to rely on scheduling policies.

and simple implementations do not allow the interruption of a GC cycle. The garbage collector inside the Kertasarie VM is a typical incremental mark–and–sweep GC[9].

During the mark phase of the GC the Java object heap is searched for living objects. Objects are alive, if they are reachable from a so called root set. The root set consists of the Java stacks, classes etc. The mark phase is the most time consuming phase of the GC. To allow the interruption of the mark phase, the objects are colored black, gray, and white. Black objects are inspected as well as all descendants. Objects, whose descendants are not completely inspected, are marked gray. White objects are not inspected yet. Now we can interrupt the mark phase and continue the mark phase on the gray objects. Between every black and white object must always be a gray one. Problems arise, when the Java application wants to change a reference from a black object to gray into a reference to white one. This situation is handled using write barriers. The write barrier will change the color of the black object into gray so the descendants must be inspected once more. The drawback of write barriers is the performance loss [15] when writing object references. We implement different write barriers for different garbage collection states to minimize the negative performance effects. Our measurements show a performance loss of only 1%. The garbage collector in the Kertasarie VM will definitely give the control back to the Java application after 175 μs on a 50MHz ARM7TDMI.

The Real-Time Specification for Java (RTSJ)[1] describes some API extensions to get more control over threads, scheduling policies, synchronization mechanisms, memory management etc. Although the Kertasarie VM currently does not support this API extensions, the VM contains many of the required features. For better synchronization mechanisms the specification demands priority inheritance, which is already implemented in the Kertasarie VM. Moreover, the RTSJ requires a PriorityScheduler class, which is the standard scheduler in our VM. Currently the Kertasarie VM does not support similar mechanisms for memory management (ImmortalMemory, RawMemoryAccess etc.) and for asynchronous interruption of threads.

3.5 Platform Independence

Code Portability In the field of embedded systems exist a large variety of processor types (architecture and variants), memory (size and type) and the peripheral components. Furthermore, there is quite a number of operating systems and C compilers. Hence, the Kertasarie VM was developed with high portability in mind.

All source code conforms to the ANSI C standard to avoid compiler dependencies. On most platforms, the Kertasarie VM uses the GNU C Compiler (GCC) combined with other GNU tools (binutils, make). This depends on the target platform and the host development system, e.g. the Windows CE version needs the Microsoft Visual C++ environment. The widely used GCC offers some proprietary improvements, which are not ANSI C compliant. Some other compilers, e.g. the Intel C Compiler (ICC), are supporting these features too. The Kertasarie VM makes use of these features to improve the performance, but these parts are encapsulated into preprocessor macros. So, the Kertasarie VM can be standard compliant or it uses these features if they are available.

The independence of the processor architecture in this context means to support different word sizes and word respectively byte orders. The Kertasarie VM supports

32 bit processors (like x86 and ARM), 16 bit processors (C167) and mixed 16/32 bit architectures (M68000). It runs on little endian as well as on big endian machines.

The biggest differences between various platforms originate in the operating system. For this reason, the Kertasarie VM consists of a large, platform independent part and only a small system specific module. Depending on the platform, the percentage of platform specific code varies in the range of 4% (Windows CE, including GUI) to 1.7% (eCos, without GUI). The system dependent module contains functions like standard IO functions, socket access, timer and clock handling and functions to access Java class files (e.g. from filesystem, database, memory, network). Currently, the following platforms are supported by the Kertasarie VM: Linux (x86, ARM, PowerPC), Windows CE (StrongARM), eCos (ARM), PalmOS (M68000) and Euros (C167). A special platform is an ARM7TDMI based board, which is supported by a standalone version of the Kertasarie VM. It does not need an underlying operating system. The dependent functions (startup code, device driver) were integrated into the JVM.

IO Model To reach this platform independence, the Kertasarie VM uses a green thread model (discussed in section 3.4). This model avoids the usage of system threads. In Java, there are two common IO models, event driven IO and synchronous IO. The first one creates a Java event object, if an external event occurs. It is often used in GUI frameworks, e.g. to initiate an action if a button was pressed. The second model, synchronous IO, is used e.g. for data transfers over sockets. In this case, the execution of a Java thread will be suspended until the action (reading or writing) is possible.

Because of the green thread model of the JVM, this kind of IO can not be used on the system level. The preferred model inside the Kertasarie VM, is asynchronous IO. If an IO operation is not possible in a moment, the execution continues and the operating system notifies the VM, if the operation can be completed (e.g. with signals on Linux). The VM has to handle the Java threads (suspend, awake, reschedule) according to these IO operations. On some platforms, asynchronous IO is not available (e.g. Windows CE, eCos). In such cases, the Kertasarie VM uses additional operating system threads to handle IO operations. Such a thread blocks, until the IO operation can be done. After that, it notifies the execution thread of the VM similar to the asynchronous IO model. The only difference between both is the interface to the operating system.

A special case is PalmOS. It neither supports asynchronous IO nor user level threads. So, the socket implementation uses polling. For this reason, Java sockets are only limited usable on this platform.

The Kertasarie VM offers a very flexible IO model with a high portability and a fast reaction with realtime capabilities.

Native Code Some applications and API classes require the usage of native code. They offer the possibility to access hardware and system specific functions or to accelerate applications by sourcing out performance critical modules. For this purpose, the Java Native Interface (JNI) was defined. It allows to link native code at runtime and to call native functions from within Java and vice versa.

This has two big disadvantages. Firstly, dynamic code loading is required. In many embedded systems, this feature is not available. Hence, a JNI implementation is not

possible on all systems. Secondly, the complexity to call Java routines from native code results in a big code size.

For these reasons, the Kertasarie VM does not implement JNI, but it supports native code. These functions are statically linked into the JVM binary at compile time. The Kertasarie VM allows the function call from Java programs and the generation of Java objects and the thread management from within the native code.

3.6 Performance

A Java program consists of Java bytecode. The task of a JVM is to execute this bytecode. There are different ways to do this.

The first possibility is to interpret each bytecode separately. This technique is easy to implement and requires few memory. It can be implemented platform independent. Although this implementation does not offer the fastest execution it is the preferred type of execution engine for embedded systems.

In contrast to interpreters, just in time compilers (JIT) translate Java classes (or parts of it) into other code, usually machine code of the target system. As a result, the execution of Java applications is much faster. On the other hand, such a solution is difficult to implement and contains usually a lot of platform dependent code. The size of the JVM will be increased considerably.

Besides these solutions, there are some special possibilities like ahead of time compiler (offline compilation into machine code) and hardware supported execution. But this loses a lot of flexibility because it is not possible to load additional Java code at runtime (in the first case) or special hardware is required (second case).

Due to the focus on a small memory footprint, the Kertasarie VM uses a bytecode interpreter (in the current version). The size of the Kertasarie VM is between 80 and 170 KByte, depending on the platform. This size includes the native methods for sockets, serialization and reflection. To illustrate the advantages of the Kertasarie VM, it is compared with the KVM (designed for J2ME/CLDC) and the CVM (designed for J2ME/CDC) from Sun Microsystems and the J9 from IBM. Compared to this JVMs, the Kertasarie VM is very small in consideration of the implemented features (see figure 2).

Besides the memory footprint, the execution speed is an essential property. To examine the performance the UCSD Java Benchmark[6] was used. It is a low level benchmark, which consists of nine parts to test different properties: bytecode execution (1 to 3), array and field access (4 and 5), method calls (6 and 7), exceptions (8) and thread switching (9). Table 1 shows the results (runtime in ms on an Athlon/1GHz). The distribution of the values gives some indication about the internal operations. For example, the J9 is by far the fastest JVM in calling methods. This gives an advice, that the JIT accelerates these functions enormously. In contrast to that, the priority inheritance mechanisms in the Kertasarie VM (see section 3.4) led to slightly increased thread switching times[3].

[3] The priority inheritance mechanism described in section 3.4 requires the usage of priority ordered lists for the thread queues. Insertion in such sorted queues is more expensive than only to append threads to the end of an unsorted queue.

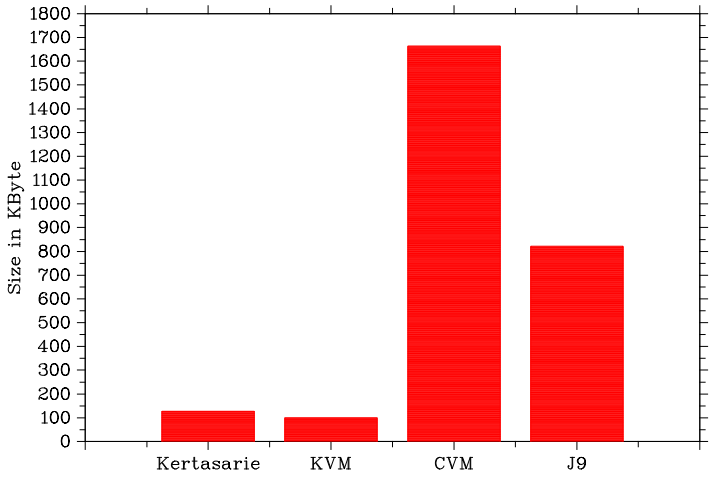

Fig. 2. Sizes of different JVM implementations

The performance ratio between the different JVMs depends on the hardware platform. The comparison on a PalmOS PDA in [2] shows, that on this platform Kertasarie VM performs better than the J9 and KVM. Probably the speed benefit of the J9 (which uses a JIT) will be larger on powerful RISC architectures.

Speed and size of an implementation are the most crucial properties for the choice of a JVM. They must be combined to a single value to give a ranking of JVMs. One possibility could be a product of size and runtime. It should be handled with care, but it gives a comprehensive reference value with a statement like “The double speed legitimates the double size”. Possibly, it could be expanded by weights (e.g. speed is more important than size) or the usage of nonlinear factors (e.g. a logarithmic scale for size). But according to the simple product (size in KByte and cumulative runtime in s), the Kertasarie VM is the best one (191) followed by the KVM (265), the J9 (1085) and the CVM (2017).

This comparison shows two important results:

1. Only the KVM is smaller, but it does not offer sockets, serialization and reflection.
2. The Kertasarie VM is slightly slower than the CVM and the J9 but significantly smaller.

3.7 Possible Use Cases

The previous sections showed that the Kertasarie VM offers all the features to implements scenarios like mentioned in section 2.1. The available features, the memory con-

Table 1. UCSD Java Benchmark on Linux/x86

		Kertasarie VM	KVM	CVM	J9
1	empty loop	70	154	44	49
2	add	191	365	126	135
3	mul	186	368	132	137
4	array assign	168	269	124	122
5	object fi eld access	218	298	93	87
6	method call - same object	158	340	161	76
7	method call - titer object	178	389	178	87
8	exception	209	445	260	547
9	thread switching	72	14	54	42
10	cumulative runtime	1494	2656	1212	1320

sumption and the performance make it a very good choice for Java in embedded systems. Additionally, with the simple GUI API the Kertasarie VM can be used in mobile information appliances.

4 Future Development

4.1 JIT Compilation

Performance is always an important concern in Java execution. The stack machine for executing Java bytecode is not very efficient on modern, register based architectures. Almost every operation belongs to the stack and results in a memory access. The large register files of modern microprocessors are not adequately used. We thoroughly analyzed different kinds of bytecode modifications: instruction folding[4], bytecode specialization[5] and the mapping of Java bytecode to a register based opcode. The statistical analysis yielded to some interesting results. Instruction folding and bytecode specialization will not lead to a significant performance gain. A very optimistic estimate gives a maximum gain of 3–6% for specialization and 5% for instruction folding. The investigations for mapping Java bytecode onto a register machine showed that 94% of all methods of the Kertasarie API could be executed with only 10 registers in the processor. Moreover, many stack related instructions can be omitted, which decreases the amount of instructions by 34%. Currently we are implementing two versions of JIT compilation: a platform dependent and a platform independent one. Both new execution engines are integrated into the Kertasarie VM. The minimal expected performance gain is about a factor of two for the platform independent JIT and a factor of five for the platform dependent JIT.

[4] Several instructions are combined to a new opcode.

[5] Some bytecodes cover some sub variants. The check for some conditions could be omitted using specialized bytecodes.

Platform dependent JIT To achieve a maximum performance gain we are developing a JIT for the ARM architecture[8]. This processor platform was chosen because it is very wide-spread in embedded systems. This JIT translates Java bytecode directly into ARM instructions. The platform dependent JIT promises the highest performance gain. A big disadvantage is the complete loss of realtime capabilities. Each compiled Java method will run until it blocks in a system call or at the end of the method. This is an implication of our green thread model (see section 3.4). If we enhance the Kertasarie VM to support native (operating system) threads, we will get the realtime capabilities back. Another drawback is, that the JIT compiler must be adapted to every new microprocessor platform.

Platform independent JIT A Java class running inside the Kertasarie VM translates Java bytecode into a new, register oriented instruction set. The new opcodes are now 16 bits wide. A new, register based virtual machine written in C will be much more efficient on a register architecture. The main advantage of the platform dependent JIT is the complete preservation of the realtime capabilities. The same mechanisms from the original bytecode interpreter are applicable here. In addition, this JIT compiler is very portable because no adaption is required for a new processor architecture. On the other hand, the performance gain will be noticeably smaller than the compilation into machine instructions.

4.2 Alternative Transport Layers for TinyRMI

TinyRMI is also subject of further improvements. In [4] we presented a transport layer relying on UDP. The introduced concepts could be adapted to every message based communication mechanisms like CAN (Controller Area Network) or even SMS (Short Message Service). Currently, we are developing a new transport layer based on CAN. This will enable Java based middleware for industry automation systems.

5 Conclusion

In this paper we discussed the major problems and design issues of JVMs for embedded systems. With the Kertasarie VM we have shown that it is possible to use Java in embedded systems even under hard memory constraints.

The Kertasarie VM offers CDC functionality at a size that is typical for CLDC implementations. This functionality enables the usage of middleware with the Kertasarie VM. Thus, it is a very good basis for distributed objects in embedded systems.

Realtime plays an important role in the embedded systems area. The Kertasarie VM is one of the few JVMs that offer realtime capabilities. Other realtime Java implementations are either much larger or much slower.

The development of JIT compilers for the Kertasarie VM will give a substantial performance gain. The ARM version will have a much higher speedup at the expense of realtime capabilities.

References

1. G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Addison Wesley Longman, Reading, Massachusetts, 2000.
2. F. Burchert, S. Gatzka, C. Hochberger, H. Kopp, and D. Tavangarian. Providing Java Based Middleware for PalmOS Devices. In U. Büker, H.-J. Eikerling, and W. Müller, editors, *Proceedings International ITEA Workshop on Virtual Home Environments*, volume 12, pages 113–120, Aachen, 2002. Shaker Verlag.
3. S. Gatzka, C. Hochberger, and H. Kopp. Deployment of Middleware in Resource Constrained Embedded Systems. In *Tagungsband der GI/OCG-Jahrestagung 'Informatik 2001'*, pages 223–231, Wien (Österreich), September 2001. Österreichische Computer Gesellschaft.
4. S. Gatzka, C. Hochberger, and H. Kopp. A Datagram Based Middleware for Embedded Systems. In K. Irmscher and K.-P. Fähnrich, editors, *KiVS 2003*, pages 197–202. VDE Verlag, 2003.
5. M. Govindaraju, A. Slominski, V. Choppella, R. Bramley, and D. Gannon. Requirements for and Evaluation of RMI Protocols for Scientific Computing. In *Proceedings of the Scientific Computing Conference 2000*, 2000.
6. W. G. Griswold. Bill and Paul's Excellent UCSD Benchmarks for Java. http://www-cse.ucsd.edu/users/wgg/JavaProf/javaprof.html.
7. IBM. Websphere Studio Device Developer. http://www-3.ibm.com/software/wireless/wsdd/.
8. D. Jagger and D. Seal. *ARM Architecture Reference Manual*. Addison Wesley Longman, Reading, Massachusetts, second edition, Dec 2000.
9. R. Jones and R. Lins. *Garbage Collection. Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Ltd., San Francisco, 1999.
10. B. Joy, G. Steele, J. Gosling, and G. Bracha. *The Java Language Specification, Second Edition*. Addison Wesley Longman, Reading, Massachusetts, second edition, Jun 2000.
11. Object Management Group. CORBA 2.4 Specification. OMG Web Server, 2001. http://www.omg.org/technology/documents/formal/corba_2.htm.
12. L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, pages 1175–1185, Sep 1990.
13. Sun Microsystems. The CVM. http://java.sun.com/products/cdc/.
14. Sun Microsystems. The KVM. http://java.sun.com/products/cldc/.
15. P. R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proc. Int. Workshop on Memory Management*, Saint-Malo (France), 1992. Springer-Verlag.