Darjeeling, a Java Compatible Virtual Machine for Microcontrollers

Niels Brouwers
Delft University of Technology
The Netherlands
n.brouwers@student.tudelft.nl

Peter Corke
Autonomous Systems Laboratory
CSIRO ICT Centre, Australia.
peter.corke@csiro.au

Koen Langendoen
Delft University of Technology
The Netherlands
k.g.langendoen@tudelft.nl

ABSTRACT

The Java programming language enjoys widespread popularity on platforms ranging from servers to mobile phones. While efforts have been made to run Java on microcontroller platforms, there is currently no feature-rich, open source virtual machine available. In this paper we present Darjeeling, a system comprising offline tools and a memory efficient runtime. The offline post-compiler tool analyzes, links and consolidates Java class files into loadable modules. The runtime implements a modified Java VM that supports multithreading and is designed specifically to operate in constrained execution environments such as wireless sensor network nodes. Darjeeling improves upon existing work by supporting inheritance, threads, garbage collection, and loadable modules while keeping memory usage to a minimum. We have demonstrated Java running on AVR128 and MSP430 microcontrollers at speeds of up to 70,000 JVM instructions per second.

Categories and Subject Descriptors

D.1.5 [**Object-oriented Programming**]: Miscellaneous

General Terms

Languages

Keywords

Java, Sensor Networks

1. INTRODUCTION

Virtual machines (VMs) are a well known and powerful means of abstracting underlying computer hardware from an application, allowing portability across platforms without recompilation. A VM is an abstract machine for which code is compiled, and the run-time interpreter for the VM code is written specifically for each platform. The best known example is the Java language and the Java VM (JVM) but VMs also underpin many other common software systems, for example Microsoft's .NET, Python and Perl.

Virtual machines provide a means of overcoming the challenges of fault tolerance, cost and heterogeneity. Low-cost embedded systems often have no user interface and are deployed in remote or dangerous areas so must run autonomously throughout their complete lifetime, i.e. for several years. Microcontrollers lack advanced features such as a memory management unit and a single faulty process can potentially take down the entire software system. Virtual machines help to alleviate these problems by providing strong checking, memory management and error handling services that improve robustness and allow software faults to be handled appropriately before they become failures.

The total cost of ownership includes not only the price of hardware, but also other costs such as software development, software and hardware maintenance, testing and cost of failures. Virtual machines may help to cut costs in the areas of software development and testing. This effect can be attributed to a number of factors, such as increased maintainability and productivity [2].

Large systems deployed for long periods of time eventually face the problem of obsolescence. Virtual machines allow elements of the system to be replaced with different computation hardware yet still be able to run the original applications and relieve programmers from having to deal with this diversity. A virtual machine solves this problem by providing one execution model that is universal to all node platforms. This is illustrated by the success of Java in the mobile phone market [13].

Growing interest in Java virtual machines for embedded applications, including Wireless Sensor Networks (WSNs), reflected by recent efforts like [14, 7], shows a need for a more flexible and accessible programming abstraction. At the time of writing Java is one of the most popular programming languages [19]. This gives Java a significant advantage over other alternatives in terms of accessibility, integration with other network components and availability of tools such as IDEs and compilers, not to mention programmers.

On the technical side, the execution model of a Java virtual machine has numerous advantages over native code. Stack frames are allocated on the heap in an ad-hoc manner so threads can be very light-weight. The Java language and its compiler guarantee type safety, and common programming errors such as buffer overflows and null pointers are caught at runtime. Unreachable memory is automatically reclaimed by the garbage collector, greatly reducing memory leaks.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Middleware '08 Companion, December 1-5, 2008 Leuven, Belgium
Copyright 2008 ACM 978-1-60558-369-3/08/12 ...$5.00.

	Java Card	NanoVM	TinyVM	AmbicompVM	VM*
Stack	16-bit	32-bit	32-bit	32-bit (48)	32-bit
Multithreading	no	no	yes	yes	yes
Garbage	yes	yes	no	yes	yes
Loadable	yes	no	no	yes	yes
Open	no	yes	yes	no	no

Table 1: JVM comparison

In this paper we discuss a new virtual machine and toolchain that allows a significant subset of the Java language to execute on a microcontroller of the MSP430 or ATmega 128 class. These devices have 1-8 KB of RAM and about 16-128 KB of program memory so a low RAM footprint is critical.

The next sections of the paper discuss prior work on VMs for embedded microcontrollers, introduce the Java VM, and then outline our solution for executing Java on embedded microcontrollers. Our focus is the application in WSNs but our Java system can be used for any embedded microcontroller application.

2. PRIOR WORK

Reported VM projects can be roughly divided into two groups: traditional VMs such as Java VMs and so called Application Specific Virtual Machines, or ASVMs. Application specific VMs abstract common operations as instructions in a stack-based virtual machine. They trade off flexibility for code footprint, both in the VM and the application code. This approach has been popular especially in the field of wireless sensor networks, where applications are transmitted wirelessly and small code size means reduced power consumption for communication. Examples of such VMs are Maté [8] and VMSCRIPT [12].

A multitude of projects have focused on making Java execute on tiny platforms. An overview of these projects and their features is given in Table 1. We have looked at stack width as this influences memory consumption, support for multithreading and garbage collection, support for runtime loadable modules (libraries) and whether these projects are open source.

Java Card [10] is an effort by SUN, the creators of Java, that targets smart cards. These platforms typically have 1-2 KB of memory and about 16 KB of program memory. An interesting feature of Java Card is that it uses a stack width of 16 bits rather than the standard 32. This essentially means it is a 16-bit JVM, which makes it more suitable for microcontrollers.

NanoVM [11] is an open-source, limited JVM written for Atmel's AVR series of microprocessors. It can run in less than 8 KB of program flash, and 1 KB of memory. It was designed for robot platforms and does not support more advanced features such as exceptions and threads and has a limited inheritance model. TinyVM [18] was developed to bring Java support to the Lego Mindstorms RCX brick. It has many features including multithreading but unfortunately lacks garbage collection.

Java has been reported for WSN applications, notably AmbicompVM [14] and VM* [7]. The AmbiComp VM is a flexible and complete JVM. It can operate in a distributed fashion using key-based routing to address remote objects. Stack elements carry two additional bytes with type information, making the stack width 48 bit. VM* features a few interesting performance optimizations. Memory limitations are addressed by synthesizing a VM for a specific application.

Of the examined projects only NanoVM and TinyVM are open source. Upon examination neither of these proved a suitable starting point for further work.

3. DARJEELING

The Java Virtual Machine (JVM) [9] was designed specifically to execute compiled Java programs. It uses a stack-based instruction set, called *bytecode* to reflect the general size of opcodes. The stack is 32 bits wide, but 64 bit data types are supported by having them occupy two stack slots. Because it is a stack-based VM, code density is generally greater than with register-based instruction sets [16]. Java is an object-oriented language and the VM supports features such as class inheritance, interfaces, and virtual method invocation.

The design constraints for many embedded systems, particularly wireless sensor networks, are quite different to those for desktops and mobile phones. One important difference is that system RAM is limited, ranging from about 1-8 KB on most sensor node hardware. Another is that sensor node applications spend most of their lifetime in sleep mode, periodically executing small segments of code. Execution speed of non time-critical applications is therefore less important. This observation has led us to actively trade off clock cycles for bytes on the heap, the exact reverse of many optimizations found in desktop and mobile JVMs where memory is available by the megabyte.

When designing our Darjeeling Virtual Machine (DVM) we have chosen not to implement the full Java virtual Machine Specification. It is our philosophy that instead of scaling down the JVM to fit on our target processors, it is better to design a new VM from the ground up and map appropriate parts of the Java platform onto it. Important tradeoffs have been compatibility, features, and performance versus code complexity and memory usage.

Darjeeling does not use the class loader system present in Java, and as a result memory overhead is greatly reduced. Darjeeling uses a static linking model discussed in Section 3.1. Modifications have been made to the instruction set to accommodate this new linking model, support packing of heap objects and static variables. Support for 64-bit datatypes and floating point has been omitted.

Darjeeling does not support the full standard class libraries, but rather provides a small footprint, bare-essentials system module ("infusion"). Libraries can be added to give the application developer control over peripherals such as the radio. This is discussed in more depth in Section 3.4.

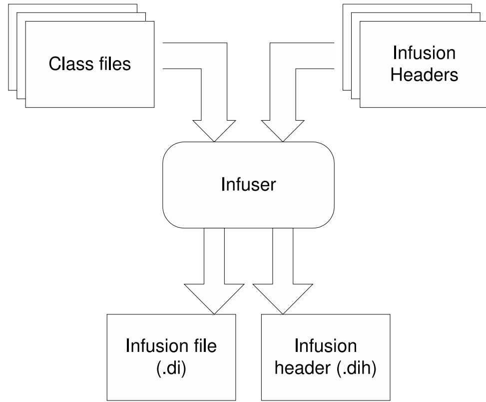

Figure 1: Infusion process

3.1 Linking model

In Java, every class is treated as a dynamically linked, loadable module. Entities, such as fields and methods, are referenced by name, and these names are stored as string values in the class files. While this model is very flexible and allows for code updates on a per-class basis, it is also very costly in terms of storage. Other embedded JVMs have implemented a static linking model where all application class files are linked together into a single binary [11], resolving each reference into an integer identifier. This results in binaries that are considerably smaller, but the downside of this approach is that common code cannot be shared across applications.

To achieve a small code footprint while ensuring modularity, Darjeeling uses a model where groups of classes are statically linked into loadable modules called *infusions*. Infusions may reference each other, can be loaded and unloaded at run time, and support versioning.

An application or library consisting of a group of classes is transformed in a post-compile step by a tool called the *infuser*. The infuser statically links Java `.class` files together and resolves names into identifiers. Classes, methods and fields are sequentially numbered. A header file is produced that contains the mapping between the original names and the identifiers that replace them. These header files are used when one infusion imports another, and to make sure that new versions of infusions do not break previous dependencies by changing identifiers.

The process is illustrated in Figure 1. The Java class files that make up the application or library are input to the infuser, along with the header files of imported infusions. The output consists of two files, a Darjeeling infusion file (`.di`) that contains the actual bytecode and a Darjeeling infusion header (`.dih`) that contains a mapping between the original Java entity names and the generated identifiers.

The identifiers that are found in the `.di` files are called *local IDs* and consist of two parts, a *local infusion ID* and an *entity ID*. The first element refers to an item in the import list of an infusion. The second element refers to an entity within that imported infusion. Local IDs are stored as a two-byte tuple. Figure 2 shows how a method is resolved at runtime. In this example a method inside the 'motor' infusion is called from the 'car' infusion. First, the local ID is partially resolved into a *global ID* by looking up the infusion in the import list. A global ID is a tuple of a pointer to a loaded infusion, and an entity ID. The method itself can now be retrieved from the infusion's method list.

3.2 Executable size

By removing the naming information we can substantially reduce the size of executables. Table 2 shows file sizes of four applications. The Car application is a single class that controls an r/c car, SimpleApp is a token-ring application that uses the radio, the system library is the system infusion that is required by all applications to run and lastly the regression tests are a group of classes used to test Darjeeling. The first column shows the sum of the class file sizes, with the code column showing how much of that is used for storing actual bytecode. The code/class ratio shows how much overhead dynamic linking information contributes for each case, and illustrates why we decided to remove it from our format. The jar column shows the compressed size of the classes, with no manifest files included. The system library consists of many classes which explains why the jar file is actually larger than the uncompressed class files, as file path information was added to the archive. The size of the `.di` files and the ratios versus `.class` and `.jar` files show that even though we do not use code compression our executable format produces files that are considerably smaller than the corresponding `.jar` files.

3.3 Memory model

Java is in its core a 32-bit virtual machine. On the stack, in local variables, and in objects, values are stored in 32-bit slots. This is beneficial for performance on 32-bit architectures but leads to memory wastage on 8 and 16-bit microcontrollers. Most applications can use `short` instead of `int` for counters, temporary variables and so forth. Ideally variables of type `byte` or `short` should only occupy 1 or 2 bytes respectively instead of 4.

Darjeeling packs the fields of objects on the heap, with references being separated from integer fields to help with garbage collection. The instructions `getfield` and `setfield` have been replaced by the new instructions `getfield_<T>` and `setfield_<T>`, where T is one of `int`, `short`, `byte` or `ref`. The offset of the field inside the object is stored as an immediate value in the instruction. A similar method is used to pack static fields, which are allocated on the heap as a part of the loaded infusion.

Darjeeling uses a simple mark & sweep garbage collector. We chose this algorithm because of its simplicity, and because it does not move objects. This allows allocated Java objects to coexist with native objects such as stack frames. A downside is that non-compacting collectors typically cause fragmentation on the heap. Another is that the mark phase is usually implemented using recursion, which can potentially cause stack overflows. We are therefore implementing a slower but non-recursive marking algorithm.

A function call causes a new stack frame to be allocated on the heap. This stack frame contains bookkeeping such as a pointer to the parent frame, a stack, and local variables. When slots are 32-bit wide, each stack element and local variable occupies four bytes. This causes an average stack frame to occupy in the order of 30-50 bytes. To reduce this overhead, it is possible to use 16-bit slots instead of 32. This requires bytecode analysis and changes to the instruction set

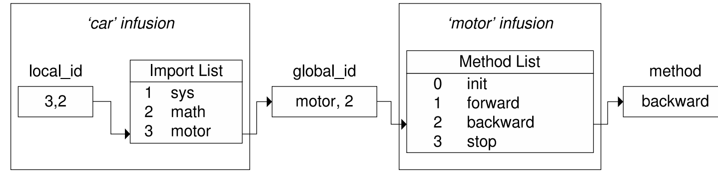

Figure 2: Resolving a method reference

Application	Class	Code	Code/Class	Jar	DI	DI/Class	DI/Jar
Car	699	108	0.15	601	179	0.26	0.30
SimpleApp	2,205	623	0.28	1,593	813	0.37	0.51
System library	2,264	84	0.04	3,320	724	0.32	0.22
Regression tests	28,998	10,798	0.37	18,349	12,337	0.43	0.67

Table 2: File size comparison

[17]. We are planning on implementing this technique in the future.

The `.di` files are placed in different memory sections on different platforms. The AVR and MSP430 platforms both have a certain amount of program flash, but access to them is quite different. On the MSP430 for instance, reads have to be word-aligned. On the AVR program flash is not mapped into the general address space and reading is done through a special instruction. To resolve these differences a thin layer of macros is written for each platform to access a `.di` file and its elements.

3.4 Interaction with native code

A mechanism for calling native methods is implemented to allow Java programs to make use of the platform hardware and interact with the virtual machine. When a method is declared with the `native` keyword, calling that method in a Java program will result in the virtual machine resolving the corresponding native (written in C) implementation of that method and executing it.

The mapping between a native method declaration in Java and a native method is generated by the infuser tool. It can generate a `.h` file that contains definitions for native methods found in an infusion and the identifiers that were assigned to them. Parameters to, and return values from the native methods are passed through the runtime stack.

3.5 Multithreading

As previously mentioned, Darjeeling was developed in the context of wireless sensor networks. Allowing multiple tasks such as a network stack, routing layer and data processing application to be executed simultaneously is a challenge for which different solutions have been proposed in the field.

The traditional method of supporting concurrency is by means of multitasking. Operating systems such as Mantis OS [1] and FOS [3] provide this functionality. A drawback of this technique is that every thread requires a separate, pre-allocated stack, the size of which is equal to the worst-case usage. TinyOS [6] and Contiki [20] solve this problem by using event-driven concurrency. The event model has proven difficult for developers to work with however, especially as application complexity grows [5].

The lack of support for light-weight threads in existing operating systems is partly due to how C code is compiled and executed. The Darjeeling runtime allocates stack space dynamically rather than statically. More specifically, each stack frame is allocated as a single heap object. This might seem inefficient in terms of memory, since in most JVM implementations the stack frame of the caller is overlapped with the stack frame of the callee. In scenarios with many threads however, the benefit of not having to pre-allocate stack space for each thread quickly outweighs this drawback. If the application programmer decides to use thread synchronization to to keep threads from allocating a lot of stack space all at the same time, these benefits can be even more considerable. This is illustrated in section 4.

Test	Time	Heap	VM Instr.	Instr/sec
Tree sort	136s	890 bytes	3,652,006	26,533
Tree sort	128s	1200 bytes	3,652,006	28,531

Table 4: Tree sort test

When there is not enough heap space for a function to be called, an `OutOfStackException` is thrown, and the system can decide how to appropriately deal with the situation. Possible actions might be restarting the thread, blocking the thread until more memory is available, or killing a lower-priority task.

Darjeeling implements preemptive multithreading with atomic JVM instructions. Timeslicing is done by performing a context switch every n instructions, where n can be chosen freely and currently defaults to 256. Darjeeling supports thread synchronization with the Java `synchronized` keyword. Because the vast majority of objects will never act as monitors, monitor counters are not stored in the object headers as in some other implementations but rather live in a separate construct to further save space.

4. RESULTS

In order to quantify the performance of Darjeeling we have written two test applications and measured their execution times. The Bubblesort test is a standard $O(n^2)$ sorting algorithm, which we tested with an array of 500 values in reverse order (worst case). We also tested an 8×8 vector convolution [4] executed 10,000 times. Test were written in both Java and C and timed on an ATmega128 running at 8MHz.

Test	C	Java	VM Instr.	Instr/sec	AVR/VM	Java/Native
Bubble sort	0.74s	72s	5,134,766	71,316	112.18	97.30
Vector Convolution	2.97s	421s	28,650,085	68,052	117.56	141.75

Table 3: Performance comparison

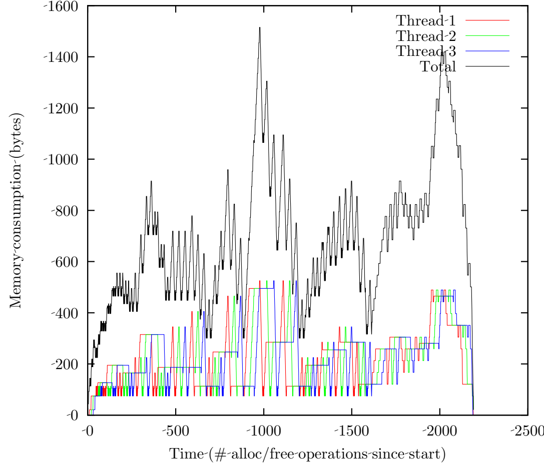

Figure 3: Binary tree test, unsynchronized

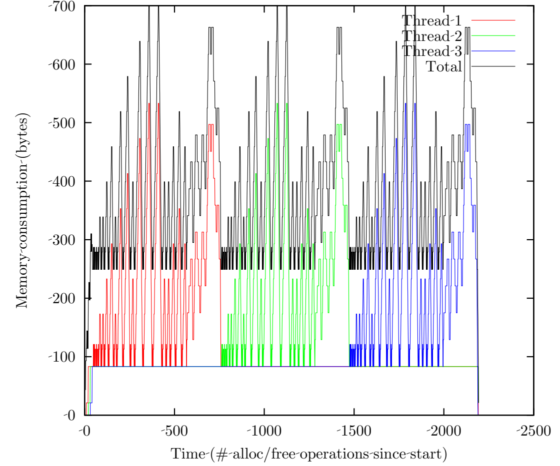

Figure 4: Binary tree test, synchronized

The compilers were GCC 4.2.1 for AVR and Javac from the Sun JDK 1.6.

The results are shown in Table 3. The first two columns show execution times for the C and Java programs respectively. The third column displays the number of instructions that were executed by the JVM, the next shows the number of instructions per second. The AVR/VM column shows the number of AVR clocks per VM instruction, and the Java/Native ratio quantifies the performance overhead of Java over C. The tests show a performance of about 70,000 JVM instructions per second on these benchmarks.

We performed a worst-case test of the JVM with a binary tree implementation. Each tree node is represented by an object that has a byte value and references to the left and right child nodes. The tree is constructed by inserting 20 random numbers, after which it is walked and the numbers are placed into an array in ascending order. This is repeated 1,000 times. Both the insert and walk operations are implemented using recursion.

Obviously this is not an effective way to sort, but it provides a good stress test for the virtual machine because it generates a large number of small objects and virtual method calls which triggers the garbage collector frequently. The worst-case from the perspective of memory consumption occurs when the tree is being walked and the deepest node is being visited, as this causes a large amount of stack space to be allocated by the main thread.

From Table 4 we see that this application can run in 890 bytes. In order to do that, the garbage collector is invoked twice per iteration which reduces performance. Still the VM is able to execute about 26,500 instructions per second. When the heap is increased to 1200 bytes, the collector is invoked only once per iteration and performance increases to about 28,500 instructions per second.

In order to illustrate how stack space is allocated dynamically rather than statically and how this effects memory consumption, we wrote an application that runs the tree sort test concurrently in three threads. We measured the *total stack size* for each thread, which we define as the size of the thread object plus the total size of the individual stack frame objects. The overhead of the heap manager in the form of chunk headers is included in this number. Figure 3 shows the total stack size of each thread over time. The three individual threads each peak at 525 bytes. The total stack space of all threads in this test peaks at 1515 bytes.

The application is run again, but slightly altered. A `synchronized` block is added in the code so that only one thread at a time is allowed to execute the tree sort test while the other two threads remain blocked. The result is shown in figure 4. In this case the total peak consumption is 699 bytes, less than half of the unsynchronised case.

These tests show how thread synchronization can be used effectively to automatically serialize complex computations and prevent programs from running out of memory.

5. CONCLUSIONS

We have presented the motivation and design principles for Darjeeling, a system that allows Java to run on a small embedded microcontroller. The system comprises offline tools, the Infuser, and a memory efficient run-time which implements a modified Java VM. Darjeeling supports a significant subset of the Java language including inheritance, threads, native methods and garbage collection, as well as allowing loadable modules. Results have been presented for the AVR128 microcontroller and the system also runs on the MSP430.

Our plans for future work are focused on issues regarding memory. The current mark & sweep garbage collector is recursive which is problematic in a memory constrained environment, and we will investigate alternatives including incremental schemes. We also plan to add memory compaction to eliminate some issues we have observed with fragmentation. Finally we plan to change the stack slot width to 16 bits and support 32 bit int and float types which would occupy two slots, and this would also add support for float types which is currently missing. We do not see a requirement for double precision floating point arithmetic in the targeted applications. For sensor network applications we will add support for secure 'over the air' code loading, memory management for loaded infusions, and 'hot' infusion linking.

6. REFERENCES

[1] Bhatti, S. et al. (2005): MANTIS OS: an embedded multithreaded operating system for wireless micro sensor platforms. Mob. Netw. Appl. 10(4):563-579.

[2] Butters, A. M. (2007): Total Cost of Ownership: A Comparison of C/C++ and Java. Evans Data Corp, www.evansdata.com

[3] P. Corke, P. Sikka, W. Hu, S. Sen, P. Valencia, C. Crossman (Feb. 2007): a Sensor Network Architecture for Software Environments, CSIRO ICT Centre Technical Report

[4] Dunkels, A. et al. (2006): 'Run-time dynamic linking for reprogramming wireless sensor networks'. In SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems, pp. 15-28, New York, NY, USA. ACM.

[5] Dunkels, A. et al. (2006): 'Protothreads: simplifying event-driven programming of memory-constrained embedded systems'. In SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems, pp. 29-42, New York, NY, USA. ACM.

[6] Hill J. et al (2000): System Architecture Directions for Networked Sensors. In Architectural Support for Programming Languages and Operating Systems, pages 93-104, 2000.

[7] Koshy, J., Pandey, R. (2005): VMSTAR: synthesizing scalable runtime environments for sensor networks. In SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems, pp. 243-254, New York, NY, USA. ACM.

[8] Levis, P., Culler, D. (2002): 'Maté: a tiny virtual machine for sensor networks'. SIGOPS Oper. Syst. Rev. 36(5):85-95.

[9] Lindholm T., Yellin F. (1999): The Java(TM) Virtual Machine Specification (2nd Edition). Prentice Hall PTR.

[10] Lindholm, T., Yellin, F. (March 2006): Virtual Machine Specification, Java CardTM Platform, Version 2.2.2., Sun Microsystems Inc.

[11] http://www.harbaum.org/till/nanovm

[12] D. Palmer, et al. (2005). An Optimising Compiler for Generated Tiny Virtual Machines. Embedded Networked Sensors, 2005. EmNetS-II. The Second IEEE Workshop on pp. 161-162.

[13] Porthouse, C., Butcher, D. (August 2004): Multitasking JavaTM on ARM platforms. Whitepaper, ARM Limited, http://www.arm.com/pdfs/MVMWhitePaper.pdf

[14] B. Saballus et. al.: Towards a Distributed Java VM in Sensor Networks using Scalable Source Routing

[15] Sen, S. & Oliver, C. R. (2006): A Rule-Based Language for Programming Wireless Sensor Actuator Networks using Frequency and Communication. EmNetS-III. The Third IEEE Workshop on Embedded Networked Sensors .

[16] Shi, Y. et al. (2005): Virtual machine showdown: stack versus registers. In VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments, pp. 153-163, New York, NY, USA. ACM.

[17] Sun Microsystems (2002): Java Card™2.2 Off-Card Verifier. Whitepaper, Sun Microsystems, June 2002

[18] http://tinyvm.sourceforge.net/

[19] http://www.tiobe.com

[20] Voigt (16-18 Nov. 2004): Contiki - a lightweight and flexible operating system for tiny networked sensors. Local Computer Networks, 2004. 29th Annual IEEE International Conference on pp. 455-462.