Demo Abstract: A Java Compatible Virtual Machine for Wireless Sensor Nodes

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 has potentially significant advantages for wireless sensor nodes but 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 run-time. 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 and supports inheritance, threads, garbage collection, and loadable modules. 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 network

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. For sensor networks they provide a solution to challenges such as fault tolerance, total cost of owernership and heterogeneity. Sensor nodes have no user interface, cannot be conveniently reset, typically lack memory management hardware yet must run autonomously. Virtual machines provide 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 (TCO) includes not only the price of hardware, but 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 contributed to a number of factors, such as increased maintainability and productivity [1].

SenSys'08, November 5–7, 2008, Raleigh, North Carolina, USA.
ACM 978-1-59593-990-6/08/11.

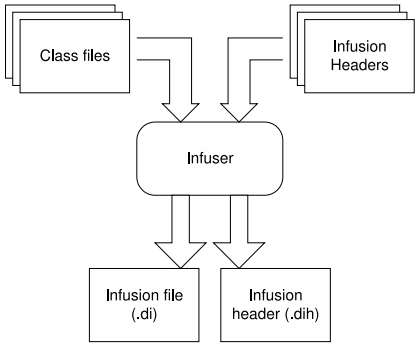

Figure 1: Infusion process

Large sensor networks deployed for long periods of time will face the problem of obsolescence. Virtual machines allow nodes to be replaced with different 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.

Growing interest in Java virtual machines for Wireless Sensor Networks (WSNs), reflected by recent efforts like [2, 3], 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. 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.

In this demonstration we will show 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.

Table 1: Performance comparison

Application	C	Java	VM Instr.	Instr/sec	AVR/VM	Java/Native
Worst-case bubble sort	0.74s	72s	5,134,766	71,316	112.18	97.30
8x8 Vector Convolution	2.97s	421s	28,650,085	68,052	117.56	141.75

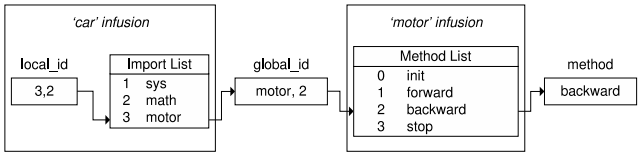

Figure 2: Resolving a method reference

2. DARJEELING

A sensor node applications spend most of its lifetime in sleep mode, periodically executing small segments of code. Execution speed is typically not important which has led us to actively trade off clockcycles for bytes on the heap, the opposite of optimizations found in desktop and mobile JVMs.

When designing our Darjeeling Virtual Machine (DVM) we have chosen not to implement the full Java virtual machine specification. Instead we have taken a subset and applied modifications to the memory layout and instruction set. The bytecode emitted by the Java compiler is post-processed by the Darjeeling toolchain into Darjeeling bytecode. Of the 227 original instructions we kept 94, modified 6, and introduced 22 new ones, bringing the total number of opcodes in the DVM to 122. Important tradeoffs have been compatibility, features, and performance versus code complexity and memory usage.

Darjeeling does not support the full standard class libraries, but rather provides a small footprint, bare-essentials system module ('infusion').

2.1 Linking model

In Java, every class is treated as a dynamically linked, loadable module. Entities, such as fields or 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 and in terms of radio transmission time and energy.

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. Other embedded JVMs have also implemented a static linking, eg. [4].

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.

2.2 Memory model

Java is in its core a 32-bit virtual machine. 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. 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.

3. RESULTS

We evaluate performance with two applications and the results are shown in Table 1. The Bubblesort app is a standard $O(\frac{1}{2}n^2)$ sorting algorithm, which we tested with an array of 500 values in reverse order (worst case). We also tested an 8×8 vector convolution executed 10,000 times. Applications were written in both Java and C and timed on an ATmega128 running at 8MHz. The compilers were GCC 4.2.1 for AVR and the Eclipse built-in Java compiler. 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 tests.

4. REFERENCES

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

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

[3] 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.

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