Aalborg Universitet

AALBORG UNIVERSITY
DENMARK

Java for Cost Effective Embedded Real-Time Software

Korsholm, Stephan

Publication date:
2012

Document Version
Early version, also known as pre-print

Link to publication from Aalborg University

Citation for published version (APA):
Korsholm, S. (2012). *Java for Cost Effective Embedded Real-Time Software*. Department of Computer Science, Aalborg University.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal -

Take down policy
If you believe that this document breaches copyright please contact us at vbn@aub.aau.dk providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from vbn.aau.dk on: April 23, 2024

Java for Cost Effective Embedded Real-Time Software

Stephan E. Korsholm

Ph.D. Dissertation, August 2012

Abstract

This thesis presents the analysis, design and implementation of the **H**ardware near **V**irtual **M**achine (HVM) - a Java virtual machine for embedded devices. The HVM supports the execution of Java programs on low-end embedded hardware environments with as little as a few kB of RAM and 32 kB of ROM. The HVM is based on a Java-to-C translation mechanism and it produces self-contained, strict ANSI-C code that has been specially crafted to allow it to be embedded into existing C based build and execution environments; environments which may be based on non standard C compilers and libraries. The HVM does not require a POSIX-like OS, nor does it require a C runtime library to be present for the target. The main distinguishing feature of the HVM is to support the stepwise addition of Java into an existing C based build and execution environment for low-end embedded systems. This will allow for the gradual introduction of the Java language, tools and methods into a existing C based development environment. Through program specialization, based on a static whole-program analysis, the application is shrinked to only include a conservative approximation of actual dependencies, thus keeping down the size of the resulting Java based software components.

The Safety-Critical Java specification (SCJ), Level 0 and 1, has been implemented for the HVM, which includes preemptive task scheduling. The HVM supports well known concepts for device level programming, such as Hardware Objects and 1st level interrupt handling, and it adds some new ones such as *native variables*. The HVM is integrated with Eclipse.

The work presented here is documented in 5 conference papers, 1 journal article, and 1 extended abstract, which are all included as part of this thesis. A summary of these papers is given in a separate Section.

Contents

1 Introduction

Successful companies within the technology industry constantly monitor and optimize the work processes of their production. "Lean manufacturing" is one well known example of a production practice that constantly optimizes work processes in order to increase earnings.

Similarly for the development of software intensive products: new efforts are continually undertaken to increase the productivity and quality of software development.

This thesis presents technical solutions that may increase the usability and attractiveness of the Java programming language as a programming language for embedded software development. The principle aim of this effort is to offer tools that will increase productivity and quality of software development for embedded platforms.

The experimental work is embodied in the HVM (**H**ardware near **V**irtual **M**achine). The HVM is a lean Java virtual machine for low-end embedded devices. It is a Java-to-C compiler but it also supports interpretation. The main distinguishing feature of the HVM is its ability to translate a single piece of Java code into a self contained unit of ANSI-C compatible C code that can be included in an existing build environment without adding any additional dependencies. *The raison d'être of the HVM is to support the stepwise addition of Java into an existing C based build and execution environment for low-end embedded systems.* Other important features of the HVM are,

- Intelligent class linking. A static analysis of the Java source base is performed. This computes a conservative estimate of the set of classes and methods that may be executed in a run of the program. Only this set is translated into C and included in the final executable
- Executes on the bare metal (no POSIX-like OS required). The generated source code is completely self contained and can be compiled and run without the presence of an OS or C runtime library
- Hybrid execution style. Individual methods (or all methods) can be marked for compilation into C or interpretation only. Control can flow from interpreted code into compiled code and vice versa. Java exceptions are supported and can be thrown across interpretation/compilation boundaries
- 1st level interrupt handling. The generated code is reentrant and can be interrupted at any point to allow for the immediate handling of an interrupt in Java space
- Hardware object support. Hardware objects according to [57] are supported.
- Native variable support. Native variables as described in Section 5.2.13 are supported

- Extreme portability. Generated code does not utilize compiler or runtime specific features and can be compiled by most cross compilers for embedded systems e.g. GCC or the IAR C Compiler from Nohau [44]

- The HVM supports the SCJ specification [65] Level 0 and 1. It does not support garbage collection but relies on the SCJ scoped memory model for memory management. The HVM can execute the miniCDj benchmark from [52].

The design and implementation of the HVM is described in detail in Section 5.

1.1 Motivation

The Java programming language, a safe and well structured high level, object oriented development language has, since the mid 90's, been successfully applied on desktop and server platforms to cope with the increasing complexity of software systems. Empirical research has shown that this shift from previous languages and environments - mostly C - to a high-level language like Java significantly increases the productivity of the average software developer [48, 55].

There are many reasons why the use of Java, as opposed to C, will increase productivity. Some important reasons are:

- Java expresses important object oriented concepts, such as encapsulation and modularization, through simple language constructs. High level languages in general invite, through their design, the developer to write software that is amenable to reuse and easy to maintain and extend

- The way Java manages and accesses memory prevents a number of common mistakes that are easier to make in C

- But equally important as the language itself, Java, and other high-level object oriented languages, are usually accompanied by a range of open-source, efficient development tools such as Eclipse. Eclipse supports a wide range of tools and frameworks that may help the software developer to develop better software, by using e.g. Unit testing and UML modeling.

Yet, the advantages of, and experiences from, high level programming languages and object oriented design principles and best practices, as they have been used on the desktop and server platforms, have so far not found widespread use on the embedded platforms used in industrial settings. Even today the C, C++ and assembly languages are in total by far the most predominant languages ([56], page 27) when programming small resource constrained devices to control e.g. machinery in a production line or sensors and actuators in the automotive industry. Some of the main reasons are:

- *Not incremental.* Java environments tend to require the inclusion of a significant amount of functionality even when this functionality is not used by a given program. E.g. running a HelloWorld type of application

using a standard Java execution environment requires the loading of many hundreds of classes. As a result the embedded programmer will often find that adding a small piece of Java functionality requires the use of a disproportionate amount of memory resources (both RAM and ROM). It is natural to expect that when you add a piece of Java software to an existing code base you pay for what you actually use, but no more than that. If a Java environment lives up to this expectation, it is incremental, otherwise it is monolithic. Until now almost all embedded Java environments lean strongly towards being monolithic

- *Not integratable.* Integration with existing RTOS build and execution environments written in C is difficult, since Java environments tend to be based on a closed world assumption: the JVM is the main execution controller and only limited amounts of C are used for specific purposes. The contrary is the case for companies contemplating the use of Java: the existing RTOS execution environment is the main execution controller. Additionally, Java software modules cannot dictate build procedures but rather have to be added to the existing build and execution environment. Also, the Java language shields the programmer from the lower levels of the software and hardware stack. Direct access to e.g. memory and device registers, and direct access to data in the underlying RTOS (usually written in C) is not part of the language itself and makes it difficult to integrate Java with a particular hardware environment. This is especially troublesome for embedded systems, that usually tend to communicate with and control hardware to a much greater extent than non-embedded environments

- *Not efficient.* Embedded software developers expect that functionality written in Java will execute as fast, or almost as fast as C, and they will expect that the size required to store Java objects in memory is approximately the same as the size required to store them using legacy languages. Recently embedded Java environments has proven their efficiency, but until now concerns about efficiency have been important to embedded developers contemplating the use of Java for embedded systems.

C environments are better at managing dependencies and they offer a larger degree of incrementality than Java environments. The baseline offset in terms of program memory of simple C functionality is very small. C can directly access memory and device registers and C can easily be integrated, in an incremental manner, with any build and execution environment. C compilers have developed over many years and produce efficient code for a large range of platforms, both in terms of execution speed and memory requirements. C is the standard for any other execution environment in terms of efficiency. So C has been a natural choice of language for embedded software engineers, since C, to a higher degree than higher level languages, enables the engineer to control, in detail, the use of the limited memory- and computational resources available.

An underlying assumption of this thesis is that high level languages like Java are better programming languages than low level languages in terms of programmer efficiency and in terms of software quality in general. This claim is supported by research for desktop and server platforms [48, 55] and it is an assumption here that it holds true for embedded platforms as well. As the following sections will show, the latter problem with lack of efficiency of embedded Java has been solved already - and today's state-of-the-art embedded Java environments execute almost as efficiently as C - but the former two issues concerning incrementality and integratability remains open issues making the utilization of Java difficult for low-end embedded systems.

So how can the drawbacks of higher level languages be eliminated while keeping their advantages? If this problem can be solved, the architectural power of object oriented environments can be used at the level of embedded systems development. If the embedded systems engineer can be empowered through the use of object oriented methods and best practices to conquer the growing complexity of embedded systems, while maintaining the ability to control the hardware in detail, the industry as such will be able to develop software faster and to increase the quality of the software.

The Problem

Let us go back in time a couple of decades and consider a company that has done most of its embedded development in *assembler*, but now seeks to use the higher level programming language C for some parts of new functionality. They would expect the following:

- *Incrementality.* If they add a certain limited set of functionality in C they would expect to pay a cost, in terms of memory requirements, proportional to the functionality added - in other words, they will expect to pay for what they use, but not more than that
- *Integratability.* They should not have to change to a new build environment or build strategy. It should be possible to add C software artifacts to their current build environment. Also, they should not have to change their RTOS or scheduling policy. New code produced in C should be able to be executed by the existing execution environment and be easily integratable with existing assembler code
- *Efficiency.* In terms of execution time, they may accept a limited degradation for C code, but not by an order of magnitude.

These assumptions hold for the C programming language and the tool chains supporting it. Now consider the same company today exploring the options of using the even higher level language Java for writing new software components. They will find that no existing Java environment will be able to meet all of the above expectations.

1.2 Contribution

The main contribution of this work is the HVM. It is an efficient, integratable and incremental execution environment for low-end embedded systems. The HVM can execute the example code shown in Figure 1 (compiled against e.g. the latest JDK1.7) on a low-end embedded platform with 256 kB Flash and just a few kB of RAM.

```
ArrayList<String> list = new ArrayList<String>();

list.add("foo");
list.add("horse");
list.add("fish");
list.add("London");
list.add("Jack");

Object[] array = list.toArray();
Arrays.sort(array);
```

Figure 1: HVM Example

The HVM compiler is implemented as an Eclipse plugin but may also run from the command line. Figure 2 shows how the Java-to-C compilation can be activated from inside Eclipse. An additional view, entitled 'Icecap tools dependency extent' below, shows the user all the dependencies that will be translated to C.

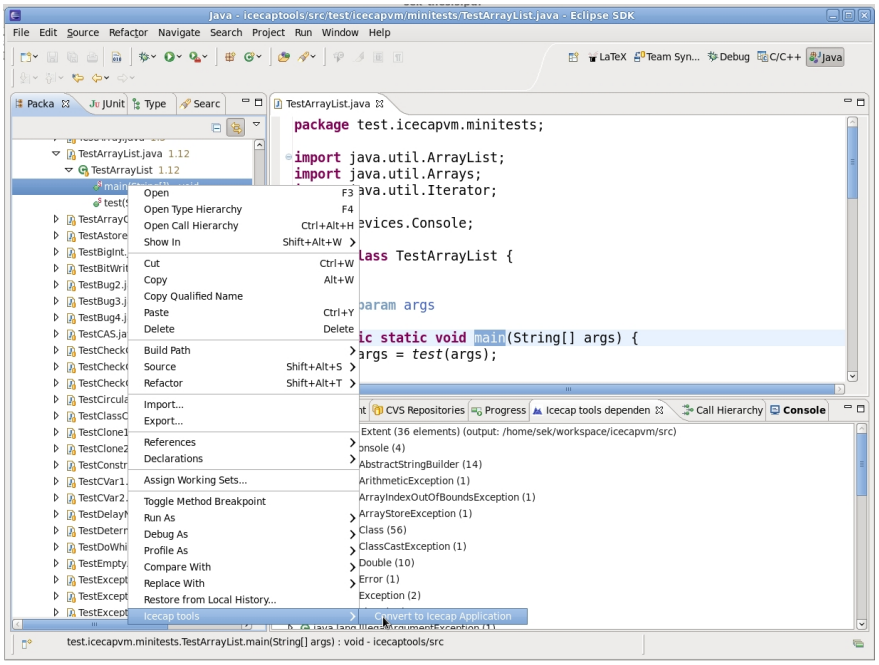


Figure 2: HVM Environment in Eclipse

Static methods implemented in Java and translated to C can easily be called from existing C code, thus supporting the seamless integration between C and Java. Since Java is translated into C, a high level of efficiency is achieved, on some platforms within 30% of native C.

Yet, this is just one step forward. There is still the important task of making the ensemble applicable to hard real-time safety critical embedded systems. In the course of its short life time the HVM has already been extended with tools to start this work:

- SCJ (**S**afety-**C**ritical **J**ava) profile. To support the SCJ, features have been added to the HVM to support preemptive task scheduling, scoped memory allocation and a real-time clock. On top of these features the SCJ Level 0 and 1 specification has been implemented and is available as part of the HVM distribution

- WCET (**W**orst **C**ase **E**xecution **T**ime) analysis. In their thesis work and paper [23] the authors present the tool *TetaJ* that statically determine the WCET of Java programs executed on the HVM. TetaJ is based on a model checking approach and integrates with the UPPAAL [2] model checking tool.

In their paper [8] the authors lay out a vision for a complete environment comprised by a set of tools for supporting the development and execution of hard

real-time safety critical embedded Java. The HVM is a candidate for a virtual machine executing the resulting Java byte code on a variety of embedded platforms. Additional to WCET analysis - which has already been implemented for the HVM - the authors also advocate the development of tools for (1) *Conformance checking*, (2) *Exception analysis*, (3) *Memory analysis* and (4) *Schedulability analysis*. Using the WALA [1] and UPPAAL [2] framework the authors have developed tools for (1), (3) and (4). As will be discussed further in Section 9, it is an important priority to continue work with integrating these tools with the HVM and the HVM Eclipse plugin.

Section 2 examines in more detail how embedded software engineers work with existing legacy C environments today and from this industrial case, Section 3 extracts a list of requirements that environments for embedded Java should seek to fulfill to support the incremental addition of Java software into a C based build and execution environment. Section 4 examines current execution environments for embedded Java and evaluate to which extent they support incrementality, integratability and efficiency. This overview of the current state of the art will show that current environments have come far in terms of efficiency and have even just made the first advances in terms of incrementality, but in terms of integratability there is a gap between the current state of the art and the requirements put up in Section 3.5. To close this gap Section 5, 6 and 7 introduces the HVM. The HVM builds on the ideas of existing embedded environments (mostly the FijiVM[50] and the KESO VM[21]) and adds a novel set of features mainly focused on integratability.

The HVM itself incarnates a body of contributions described in 5 conference papers, 1 journal article, and 1 extended abstract. These papers are included as appendices to this thesis and summarized in Section 8.

1.3 Delimitation

The challenges related to using Java in existing C based build and execution environments increase as the target platforms become smaller and smaller. Some of the reasons are,

- As the amount of computational and memory resources decrease on smaller and smaller devices, the requirement that Java environments are incremental and efficient becomes more and more important
- Because of the great diversity of low-end embedded devices compared to e.g. desktop environments, the nature of the build environments differ a great deal as well. The chance that the build environment used by a particular engineering company follows some commonly used standard is low. Build environments are often non-standard and have evolved over time and become very particular for the company in question. So integratability is even more important for low-end embedded systems.

Figure 3 illustrates an overview of computational platforms ranging from server platforms down to low-end embedded systems. The focus here is on low-

end embedded systems. In many cases the results could be applied on high-end embedded systems as well, but it is the low-end embedded platforms that can benefit the most from incremental, integratable and efficient Java environments.

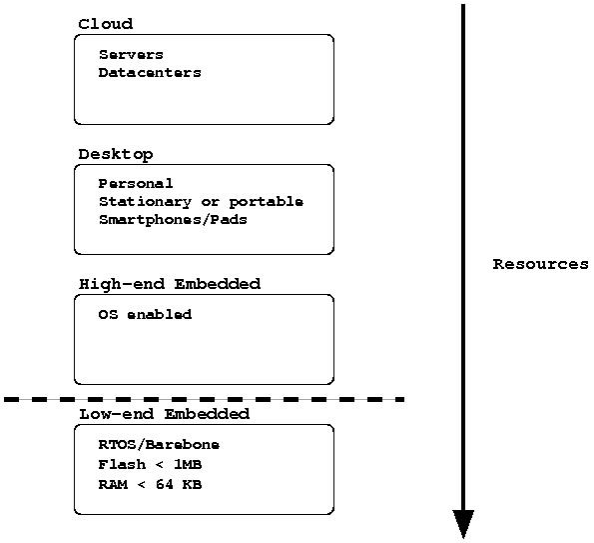


Figure 3: Platforms

The industrial case introduced in the following section is a prototypical example of a low-end embedded system: limited computational resources, non-standard build environment and a large amount of existing C software controlling the execution of the embedded software functionality.

2 An Industrial Case: KT4585

This section looks more closely at the *KIRK DECT Application Module* [54] from Polycom [53], also called the KT4585. This module can be used to wirelessly transmit voice and data using the DECT protocol. The device can be found in a range of DECT based mobile phones and in other communication solutions. It has the following features,

- 40 Mhz, 16 bit RISC architecture
- 8 kB RAM, 768 kB ROM
- External DSP (Digital Signal Processor) for voice encoding
- External dedicated instruction processor (DIP) for controlling a radio receiver/transmitter
- Microphone and speaker devices

- Low power battery driver.

The KT4585 can be programmed in C/Assembler and comes with a C based development platform and a small non-preemptive event driven RTOS. The RTOS is described in more detail as part of [37]. The bulk of the delivered software implements the DECT protocol stack, but an important part controls the DSP and DIP through low level device register access and interrupt handling.

The KT4585 is a rather complicated setup, making it a well suited case for finding the methods used by engineers when programming resource constrained, real-time, control and monitor software. An overview of the KT4585 architecture is illustrated in Figure 4.

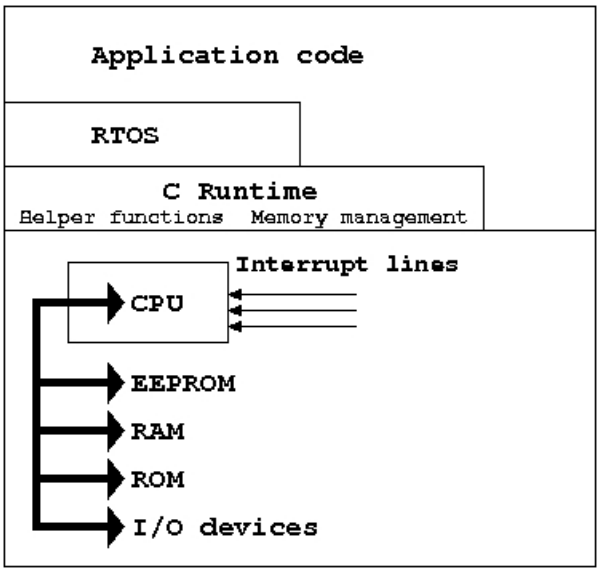


Figure 4: Simplified Architecture of the KT4585

2.1 RTOS

Polycom has developed a C based framework for programming the KT4585. This framework is based on an event-driven programming model. As observed in [26, 20], event-driven programming is a popular model for writing small embedded systems. Figure 5 illustrates the scheduling model applied on the KT4585.

The OS loop retrieves an event from the head of an event queue and dispatches the event to its handler. The handler is implemented as a C function and can be registered with the OS through a simple API. The events are dispatched in a first-come first-served order and cannot be given priorities. It is the responsibility of the software developer to handle events in a timely fashion, in order to allow other events to be handled. No tools or methods exist

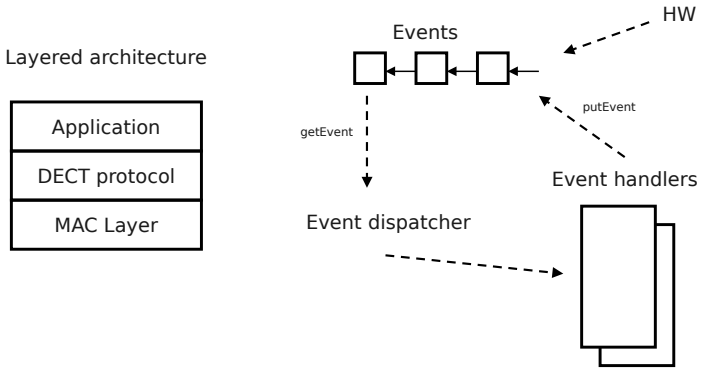


Figure 5: Event Driven Scheduling

to ensure that this rule is observed. A hardware watchdog timer will reset the device if an event handler gets stuck. Events can be inserted into the queue from other event handlers, but they can also be inserted into the queue from inside interrupt handlers. An example of this is shown below in Section 2.4.

2.2 CPU and Memory

The KT4585 main processor is the CR16c [59] from National [58]. It is a 16 bit RISC architecture with a 24 bit pointer size and on the KT4585 it is equipped with 8 kB of RAM and 768 kB of ROM which are accessed using the same instructions (Von Neumann architecture). Program code is placed in ROM. Static data can be read from ROM during runtime without any additional overhead. Writing to ROM at runtime is possible but difficult and usually avoided. This means that all dynamic data have to be kept in RAM.

It is programmed using a GCC cross compiler ported by Dialog Semiconductor [17] or the IAR C compiler from Nohau [44].

The DSP and DIP are external processors and they are programmed from the main C application by loading arrays of machine code into proper address spaces. The DIP controller runs a small hard real-time software program (512 words of instructions) that open and close the radio device at exactly the right time to adhere to the DECT frame structure. Programming the DIP software is an error prone task. As a consequence, it is seldomly changed. An assembler for the DIP instruction set do exist, but the DIP program can also be hand coded. The DIP program is stored in program memory as a C array of bytes. During

start up of the main C application, the DIP instructions are loaded from program memory and stored in a particular memory mapped address space at which the DIP program must be located. The DIP is continuously reprogrammed at runtime to open or close DECT connections. Apart from the DIP program, the DIP behavior is also controlled through a set of configuration parameters. These parameters are stored in EEPROM and retrieved and loaded into some DIP control registers at start up. The parameters are needed to fine tune the behavior of the radio electronics, a tuning made during production for each individual device.

The DIP issues interrupts to the main processor at various points in time to signal the reception or transmission of data over the radio. These data are read by the DIP and stored in a portion of RAM that is shared between the DIP and the main processor.

33.0 Memory map

33.1 MEMORY MAP OVERVIEW

Table 61: SC14480A5M, SC14480A2M6 CR16Cplus Memory Map

Address	Memory map
0x0000.0000 0x0000.7FFF	**Reserved** 32k reserved
0x0000.8000 0x0000.9FFF	**Non-shared RAM (A5M) (Not on A2M6)** 8 kbyte
0x0000.A000 0x0000.FFFF	**Reserved** 24k reserved
0x0001.0000 0x0001.FFFF	**Shared RAM for CR16Cplus, Gen2DSP and DIP** See Table 62 on page 132 for details
0x0002.0000 0x000E.FFFF	**Reserved** (832 kbyte)
0x000F.0000 0x002E.FFFF	**Program FLASH/ROM** 0x000F.0000-0x0012.FFFF for 2.0 Mbit ROM 0x000F.0000-0x0014.37FF for 2.6 Mbit ROM 0x000F.0000-0x0015.FFFF for 3.5 Mbit ROM 0x000F.0000-0x0018.FFFF for 5.0 Mbit ROM 0x000F.0000-0x0018.FFFF for 5.0 Mbit FLASH (FLASH including INFO block1, mirrored every 512 words)
0x002F.0000 0x00FE.EFFF	**Reserved**
0x00FE.F000 0x00FE.F7FF	**Boot ROM** (2 kbyte)
0x00FE.F800 0x00FE.FFFF	**Reserved** (2 kbyte)
0x00FF.0000 0x00FF.FBFF	On-chip Peripheral and Chip ID **Registers** (see page 135) (63 kbyte)
0x00FF.FC00 0x00FF.FFFF	Internally used for **ICU vectors** and traps (1 kbyte)
0x0100.0000 0x0100.01FF	DIP **Sequencer** RAM (512 bytes)
0x0100.0200 0x0100.03FF	DIP **Sequencer** RAM (512 bytes) addressed with <JMPF>,<BR_B1> commands

Confidential

Figure 6: KT4585 Memory Map

The relevant portion of the KT4585 memory map is listed in Figure 6: the area termed `Shared RAM for CR16Cplus, Gen2DSP and DIP` is the only area where both the DIP and the main processor have access. The main purpose of this area is to hold the data buffers for data being received or transmitted over the wireless link. The main program maps this area to some data structures and reads/writes the areas through pointers to these data structures. Here follows some simplified code illustrating this,

```
typedef struct {
    ... BYTE CRCresult; ...
} RxStatusType;

typedef struct {
    ... RxStatusType RxStatus; ...
} PPBearerRxDataType;

typedef struct {
    ...
    PPBearerTxDataType BearerTxData[NOOFBEARERS_PP/2];
    PPBearerRxDataType BearerRxData[NOOFBEARERS_PP];
    ...
} BmcDataPPBankType;

#pragma dataseg=BMCDATARAM
extern BmcDataPPBankType BmcDataRam;
#pragma dataseg=default
...
if ((BmcDataRam.BearerRxData[0].RxStatus.CRCresult & (1 << 6)) == 0) {
  restartDIP();
}
...
```

Now, if the `BmcDataRam` variable is located at the correct address in memory (`0x10000` according to Figure 6), and the DIP is programmed to place data into memory according to the definitions of the types above, then the data being received can be accessed from the main processor as is illustrated. The way in C to force the `BmcDataRam` variable to be located at a particular address is to annotate the source code at the declaration of the variable with compiler directives (the `#pragma dataseg=BMCDATARAM` above). The syntax of these directives vary from compiler to compiler. Then an option is given to the linker to make it place the segment in question at a particular address. For the IAR compiler from Nohau the option for the above will be `-Z(DATA)BMCDATARAM=10000-108ff`. Alternatively a linker script can be updated with this information.

The main program will also have to be able to program the DIP as illustrated below,

```
unsigned char* address = (unsigned char*) 0x10001da;
*address++ = 0x01;
*address++ = 0x63; /* U_VINT 01 */
```

This code stores the `U_VINT 01` instruction - which makes the DIP signal an interrupt to the CPU - at address `0x10001da` in the DIP sequencer RAM area. From the memory map in Figure 6 it is seen that this is `0x1da` bytes into the memory area used for the DIP program.

2.3 Device I/O

Memory mapped device registers are used to control the DIP and DSP, e.g. the DIP is controlled through the `DIP_CTRL_REG` register which is located at address `0xFF6006` (see Figure 7)[1].

Table 125: DIP_CTRL_REG (0xFF6006)

BIT	MODE	SYMBOL	DESCRIPTION	RESET
15-8	-			0
7	R/W	URST	If this bit is set to 0 the DIP starts executing the DIP sequencer program. The first DIP instruction executed is located at address DIP_RAM+1. Writing a 1 to this bit stops the DIP sequencer program execution.	1
6	R/W	PRESCALER	Global enable for P1 and P3 prescalers. If set to 1 and the DIP executes the <U_PSC> P3 divides by 16 and P1 divides by 16 if also XDIV =1. Both P1 and P3 are switched off/on if this bit is set to 0/1. The execution of <U_INTx>, <U_VINT> or <U_VNMI> DIP commands switches P1 and P3 always back to divide to 1. Note: If CLK_AMBA_REG[CLK_DIV_PRE] = 1, P3 always divides by 16, regardless of PRESCALER and <U_PSC>.	0
5	R/W	DIP_BRK_INT	If the DIP <BRK> command is executed the DIP stops executing the sequencer program and sets DIP_BRK_INT to 1. Also the DIP interrupt pending bit at the CR16Cplus is set to 1. The DIP_BRK_INT bit is cleared on reading. Writing a 1 to it starts DIP program execution at the location where the BRK command was located.	0
4	R/W	PD1_INT	Generates a DIP interrupt if S-field preamble detected. PD1 (SF) can be monitored on pin P2[5] /SF. See also chapter 7.3 This bit is ORed with DIP_CTRL2_REG[PD1_INT] Reading clears this bit and the DIP interrupt. Writing this bit: 0 = Disable PD1_INT interrupt 1 = Enabled PD1_INT interrupt is	0
3-0	R	DIP_INT_VEC	If the DIP <U_INTx> (x = 0..3) or <U_VINT> command is executed this bit is set to 1 and the DIP interrupt pending is set to 1. Reading this register sets these bits to 0.	0000

Figure 7: The DIP control register

Starting the DIP after the DIP code has been loaded is accomplished in C through code like this,

```
#define DIP_CTRL_REG       *((volatile unsigned short*)0xFF6006)

static void restartDIP(void) {
     DIP_CTRL_REG |= URST;
     DIP_CTRL_REG &= ~URST;
}
```

In general memory mapped device registers are used to control the behavior of all attached peripherals and electronic components that are controllable from the CPU program.

[1]The data sheet for the KT4585 describing the `DIP_CTRL_REG` and other device registers are not available for public download, so the reference to the document cannot be given here.

2.4 Interrupts

On the KT4585 interrupts are mostly used to facilitate the communication between the external DIP processor and the CPU. When the DIP is receiving a frame of wireless data, these data are placed in a buffer that is located in a memory space that is shared between the DIP and CPU. When the last bit of data has been received the DIP issues an interrupt to the CPU. When an interrupt occurs, the current event handler or the OS loop will get interrupted and control is automatically transferred to an interrupt handler. The synchronization between the interrupt handler and the event dispatcher is done by disabling interrupts during critical sections of code. Because of the frame structure of the DECT protocol, the CPU now has 5 ms to empty the buffer before it is overwritten by the next frame. The interrupt handler for the DIP interrupt empties the buffer and signals an event in the RTOS so the upper layers of the software can handle the interrupt in a soft real-time context. A simplified interrupt handler is shown below,

```
__interrupt static void dipInterruptHandler(void) {

    PutInterruptMail(DATAREADY);
    ... put data in mail ...
    DeliverInterruptMail(DIPCONTROLLERTASK);

    RESET_INT_PENDING_REG |= DIP_INT_PEND;
}
```

When an interrupt occurs, the hardware looks up the interrupt handler in the interrupt vector. The location of the interrupt vector in memory can be programmed through the use of special purpose instructions, and the content of the interrupt vector - which handlers to execute in the case of interrupts - can be set by writing a function pointer to appropriate locations in memory. Then the handler gets called. The declaration of the above handler is annotated with the `__interrupt` annotation. This signals to the IAR C compiler that the function is an interrupt handler. Such entry and exit code stubs will be automatically generated by the compiler to save and restore the interrupted context to the stack. On the KT4585 all interrupt handlers have to reset the interrupt explicitly (as is done above). Failure to do so will cause the interrupt to be reentered immediately.

2.5 C Runtime

The C runtime contains software features required during start up and execution of the main program. A subset of these features are,

- Start up. After the boot loader has loaded the program, an entry point defined by the user gets called. This is usually called `_start` or similar. This entry point does an absolute minimal set up of the software. On

the KT4585 it sets up the stack and initializes the interrupt vector table. Then it calls `main` - the C entry function

- Memory management. The C runtime environment may implement the `malloc` and `free` functions used to allocate and deallocate data
- Advanced arithmetic. If the micro controller does not natively support multiplication or division through arithmetic machine instructions, the C runtime may implement such functionality as ordinary functions.

The GCC and IAR C compilers come with a pre-built C runtime environment implementing all of the above, and more. It is possible to create applications that do not use the pre-built C runtime environment. Then the linker has to place the code in appropriate places to ensure that the correct entry point gets called at boot time.

2.6 Application Code

The actual software developed will consist of some abstract layers that do not communicate directly with the hardware. E.g. the upper layers of the DECT protocol stack are soft real-time software components that process events from the lower layers. But it also accesses features directly from the C runtime (arithmetic functionality and memory management) and it occasionally accesses the hardware directly through device registers. The soft real-time part of the software that does not access the C runtime, nor the hardware, makes up the by far largest portion of the framework in terms of lines-of-code.

2.7 Programming Environment

The hardware outlined above is programmed using the IAR Embedded Workbench [45]. The software configuration management is supported by the setup of 'projects' that groups and manages source code. The build procedure is automatically executed by the workbench based on the source code placement in the project structure. The IAR Embedded Workbench is a commercially available product that has been developed over many years and support a wide range of embedded targets. Apart from the software configuration management, the workbench also allows for the editing of source code, and finally it is also a configuration tool, that configures the compiler, linker, and debugger to generate code with certain capabilities. Figure 8 shows a screen shot from the IAR Embedded Workbench. For each category a large amount of options can be set, that may have a significant impact on how the program will eventually behave when it is executed.

The workbench is also used to download and start the executable and to run the debugger. All configurations set by the user are saved in a XML file. For the KT4585 an Eclipse plugin exists that can read and parse the XML configuration file. Based on this the Eclipse plugin is able to invoke the GCC cross compiler for the CR16c. The Eclipse plugin only supports a limited set of options.

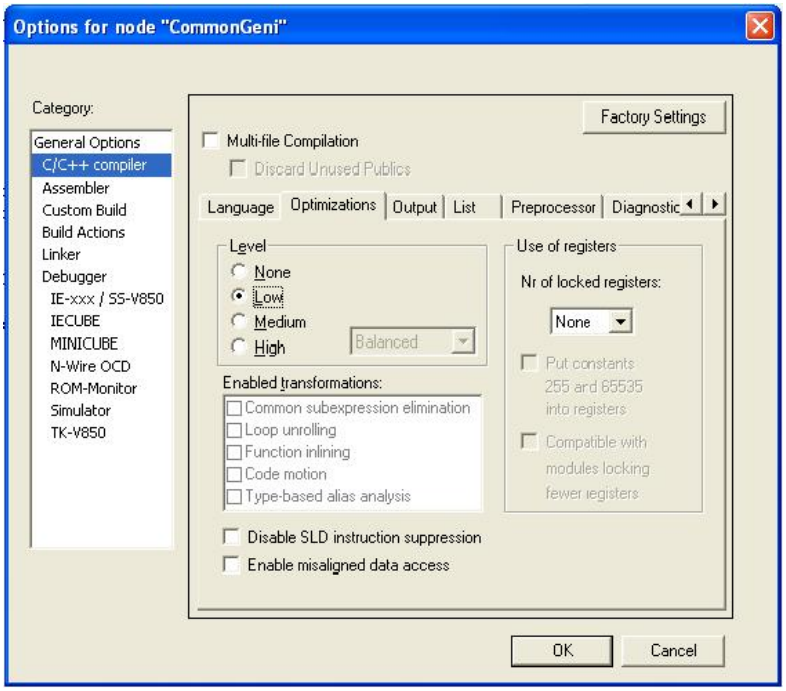


Figure 8: Options for the IAR Compiler, Linker and Debugger

3 Requirements Analysis - The Industrial Case

This section is an analysis of the industrial case described in Section 2. The outcome is a list of requirements that can reasonably be put on a Java execution environment for embedded systems such as the KT4585. Next Section 4 examines the current state of the art, comparing it with these requirements.

The industrial case described above will differ in its details from other cases on many points, because of the great diversity of embedded environments on the market, but it is assumed that the following statements hold for a significant number of low-end embedded development environments,

1. A C/Assembler cross-compiler tool-chain, either commercial (e.g. the IAR compiler from Nohau) or open-source (e.g. GCC), is used to produce executables for the embedded target

2. An IDE, similar to IAR Embedded Workbench or Eclipse with proper plugins, is used for software configuration management and to configure and call the compiler, linker and debugger

3. A standalone tool or the IDE from above is used to download the executable to the target

4. A simple RTOS exists for the target. No assumptions are made regarding what scheduling mechanism is used. Applications may also run bare bone, directly on the hardware
5. A significant amount of C code exists, possibly build on top of the RTOS above.
6. Hardware and I/O are controlled through direct access to device registers
7. Control over placement of data in memory is required
8. Interrupt handling (1st level) is required
9. An existing C runtime supports the initial start-up of the device and may support memory management and higher level arithmetic functionality.

Also, as stated earlier, focus here is on low-end embedded environments were memory and computational resources are limited. This means that size and efficiency of generated code are of interest. From these observations and from the industrial case, a list of features is extracted; features that the embedded developer will expect to be supported in his embedded Java development environment. The features are grouped under the following headlines,

1. Programming Environment
2. Software Integration
3. Hardware Integration
4. Performance.

3.1 Programming Environment Requirements

An existing embedded developer will be reluctant to abandon his currently used tool-chain. In many cases the compiler, linker, and debugger he uses, is adapted to the target in question and may even contain special purpose changes in functionality made by the tool-chain vendor or developer himself. The configuration of the tool-chain in terms of settings of options and optimizations will also be hard to change. The build procedure as supported by the IDE is also hard to change. Embedded programming is notoriously difficult, and switching to a different kind of software configuration management and build procedure will most likely be a task that developers will seek to avoid. The IDE itself, on the other hand, as used for editing of source code may not be of equal importance. The Eclipse environment with proper plugins (e.g. CDT, a C/C++ development plugin) is in many cases just as efficient, or even better, at manipulating C source code as any commercially available product. Assuming the validity of these observations a Java environment may benefit from satisfying the following programming environment requirements,

- It should be possible to compile the Java artifacts using existing, possibly non-standard, C compiler tool-chains
- The Java artifacts must be easily integratable into an existing build environment, the nature of which cannot be made any assumptions.

3.2 Software Integration Requirements

Only in the rare case, where a fresh software development project is started, and it is not based on any existing software, one can avoid integrating with legacy software. In, by far, the most common case an existing C runtime, RTOS, and software stack are present and those software components must be able to continue to function after introducing Java into the scenario. This leads to the formulation of the following software integration requirements,

- It should be possible to express Java functionality as an RTOS concept and schedule the execution of Java functionality by the RTOS scheduler
- Java functionality should not rely on any additional functionality than what is available in the existing C runtime environment
- Java functionality should be able to exchange data with existing legacy software components written in C.

3.3 Hardware Integration Requirements

Changing the hardware to accommodate the requirements of a new development environment will rarely be desirable in existing engineering scenarios. In many cases the existing hardware platform is chosen because of certain properties such as item price, power consumption, robustness to certain physical environments, other electronic attributes (e.g. resilience towards radiation and static discharges), physical size and integratability into an existing electronic component. So even though an alternative hardware execution platform for Java may exist, it is unlikely that engineers will change such an important hardware component. Hence it follows that it is desirable to support the following hardware integration requirements,

- The Java software components should be able to run on common off-the-shelf industrial hardware. This includes at least 8, 16 and 32 bit platforms
- It should be possible to access device registers and I/O from Java
- It should be possible to place Java software components in certain memory areas specified by the hardware
- It should be possible from inside Java software to handle interrupts generated by hardware
- Java software should be able to directly access memory types such as EEPROM, FLASH and RAM.

3.4 Performance Requirements

When an embedded developer e.g. adds a new task written in C to a set of existing tasks scheduled by an RTOS, he will expect to see the code size increase corresponding to how much C code he is adding. Similarly, if he is adding a task written in Java, he would expect to see the code-size increase almost linearly in relation to the amount of functionality added.

If the code manipulates byte entities, he would expect to see machine instructions being generated that are suited for byte manipulation; on the other hand, if the code being added manipulates long word entities (32 bit), he would expect to see code being generated that either utilizes long word instructions or combines byte instructions to handle the long word manipulation. On low-end embedded hardware the data width most efficiently supported by the machine instruction set is usually 8 or 16 bit. 32 bit (or larger) operations are supported by combining 8 or 16 bit operations. It has a major impact on code size end execution efficiency how successful the compiler are in choosing the right instructions for the right data type.

If the code being added allocates a certain amount of bytes in dynamic memory, it is expected that only this amount of bytes, perhaps plus some minimal amount of bookkeeping space, is required. In relation to execution efficiency he will expect that code written in Java will run almost as efficiently as C. Maybe he can accept that Java runs a little slower since he knows that Java performs some useful checks that he should have done in C (but forgot). These observations suggest the following performance requirements,

- Linear code size increase. When adding a Java software component, code size will grow corresponding to the functionality added
- Operation size awareness. If an operation performed by software can be inferred as or is declared as being a byte operation, byte operation machine instructions should be used to perform it. In general the most suited data width should be used for performing data manipulations
- Efficient dynamic data handling. The size of Java data objects should be close to the size of the actual data being stored. Just as close as the size of C structs are to the size of data saved in them
- RAM/ROM awareness. A C compiler will be careful to place static data (e.g. program code and constants) in ROM and only use RAM for truly dynamic data. The same should hold true for Java software artifacts - code and static data should be placed in ROM, whereas only truly dynamic Java objects are placed in RAM
- Execution efficiency. Performing a calculation or operation in Java should be on par with performing the same operation in C.

3.5 Requirements for Embedded Java

Java, as a high-level language, offer some interesting features that are not as easily supported in C: the Java language is a safe language and common mistakes made in C, such as pointer errors, endian confusion, dangling references, unexpected packing of structs, unclear semantics of arithmetic operations and operators and macro confusion, to mention some important ones, all these types of errors are impossible to make in Java. Additionally, on the host platform a wide range of open source and efficient set of tools exist to (1) analyze Java code and highlight potential problems, (2) use UML for modeling, or (3) do WCET and schedulability analysis. It will be acceptable to pay a certain price for these features, and a limited price in terms of slightly higher space requirements or slightly lower performance may be acceptable for non-crucial software components. But there are some areas where it will be difficult for the embedded developer to compromise,

- Programming Environment Requirements. Java must be integratable into the existing programming environment. Java artifacts (e.g. the VM) must be compilable by existing compilers and it must be possible to add these artifacts to an existing build procedure
- Software Integration Requirement. Java software components must be able to run in the context of an existing RTOS and legacy C software platform
- Hardware Integration Requirement. Java software components must be able to access and control hardware, and must be able to live on the current hardware platform
- Performance Requirements. Performance of Java software components must be on par with C in terms of space requirements and execution efficiency.

Another way of illustrating the requirements put up for Java environments, is that it should be possible to integrate Java into the existing build and execution environment used by Polycom on the KT4585. Section 2.1 described how software is scheduled on the KT4585. A natural approach to adding Java functionality into such a scenario would be to implement a new handler in Java. Figure 9 illustrates this.

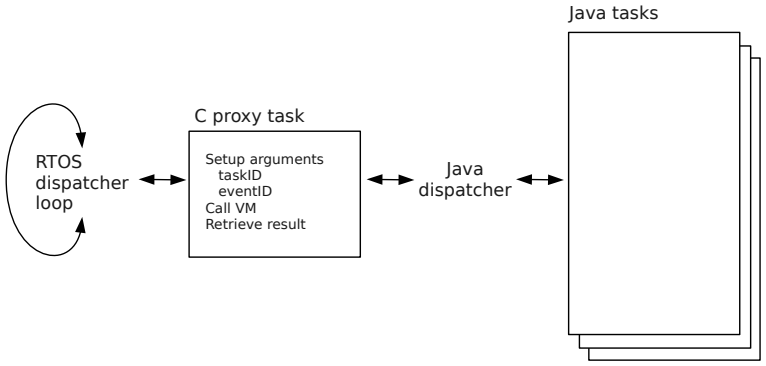


Figure 9: Example Integration

A new handler written in C is registered with the RTOS. The purpose of this handler is solely to delegate any events sent to it to the Java dispatcher. The Java dispatcher is a static method written in Java receiving a handler ID and an event ID. Its purpose is to look up the receiving handler (also written in Java) and call its handle method supplying it the event ID. This process proceeds through the following steps,

- Call setup. Let us assume for simplicity that the event value is a single byte. In that case the single byte is placed on top of the Java stack. Additionally, the ID of the handler is placed on the Java stack as well
- Call VM. Now the C proxy calls the Java dispatcher. It is assumed that the dispatcher is located in a static class. Thus it is possible to call the VM without any context on top of the Java stack apart from the handler ID and event ID. When returning from this call, the Java software components have handled the event in the Java realm
- Retrieve result. It may be possible for Java functionality to send back an indication whether the event was handled successfully or not. If this is supported, the result will be on the stack and can be popped from there.

Let us further more assume that the Java dispatcher and Java handlers are written in pure Java code and do not call any native methods.

- In an incremental Java environment the size of the added code would be some reasonable proportion of the functionality implemented in Java.

Actually, most engineers would expect the size to be of almost equal size to what would be added, had the Java handlers been written in C

- In an integratable Java environment, if the Java dispatcher and Java handlers are AOT (**A**head-**o**f-**T**ime) compiled into C (see Section 4.1.2), it should be straightforward to include the generated C files in the existing build and build them together with other handlers written in C. It should not require a particular build environment using a particular compiler, nor require the linking against any further libraries or dependencies, or the inclusion of various up until now unused header files.

- In an efficient Java environment the number of clock cycles required by Java to handle the event should be of nearly the same number of clock cycles had the Java handlers been written in C.

To be attractive to the part of industry that utilizes C for programming low-end embedded environments, an embedded Java environment should support the writing of a simple handler like above, compiling and integrating it into the existing build environment, without adding any dependencies.

If this is not possible out-of-the-box, the mentioned portion of the engineering industry will be reluctant to adopt embedded Java technologies.

The following section will describe the current state-of-the-art for embedded Java and look at to which extent the requirements laid out here are satisfied.

4 Requirements Analysis - State-of-the-art

This section gives an overview of the state-of-the-art of embedded Java environments. The main purpose of the section is to describe the ways that a language like Java can be executed, in sufficient depth to make an informed decision about which ways are the most promising for low-end embedded systems. The secondary purpose of the section is to describe a representative selection of existing embedded environments for Java, to show that embedded Java environments have come very far in terms of efficiency, but there is an opportunity for improving the current state-of-the-art when it comes to incrementality and integratability. Once these opportunities have been identified the HVM is introduced in the next section to demonstrate how this gap can be closed.

4.1 Java execution styles

Executing any programming language, e.g. Pascal, SML, C, C++, Java, or C#, can be done in multiple ways. Important execution styles are Ahead-Of-Time (AOT) compilation (or simply compilation), Just-In-Time (JIT) compilation or interpretation [18]. Hybrids exist as well, such as Dynamic-Adaptive-Compilation (DAC), which employs all three styles in the same execution environment. Some languages are most often executed using one particular style, e.g. C is usually compiled using an AOT compiler, and Java and C# are usually compiled using a JIT or DAC compilation strategy. The various execution strategies applies to all languages, and choosing the right one depends on the scenario in which the language is used. The following describes in more detail those properties of each execution style that are important to take into account when deciding on how to execute Java for low-end embedded devices.

4.1.1 Interpretation

In the beginning Java was interpreted, as stated in this quote [18]:

> The Java virtual machine (JVM) can be implemented in software as an instruction set simulator. The JVM dynamically interprets each byte code into platform-specific machine instructions in an interpreter loop.

But interpretation has been around long before the advent of Java. Interpretation can be traced back to 1966 and later the Pascal-P compiler from 1973 [69]. E.g. the Amsterdam Compiler KIT (ACK) [64] translates Pascal and other supported languages into an intermediate byte code format for a simple stack based virtual machine called EM (Encoding Machine). EM byte codes can be interpreted by the ACK interpreter, or further compiled into object code for a particular target. The Amsterdam Compiler KIT was used in a commercial setting by Danish company DSE [19] in 1983-2000.

When utilizing interpretation for execution of Java on low-end embedded platforms, the code size of the Java byte codes, as compared to the code size of a similar program translated into native code, becomes important.

There seems to be some debate if stack based byte codes such as Java byte codes requires less space than a native CISC instruction set. In [41] the authors claim a code size reduction of 16%-38% when using byte codes similar to Java byte codes as compared to native codes. On the other hand in [14] the authors conduct a similar measurement for .NET and conclude that no significant code size reduction can be measured.

Byte code compression has been the focus of a large amount of scientific works for many years (e.g. [13]), and it seems to be an underlying assumption that byte codes require a significantly smaller code size than native codes, but the final proof of this claim remains to be seen. A natural way to prove this claim would be to implement a convincing benchmark in hand coded C and in Java and compare the code size of each using two similarly mature execution environments: an AOT based execution environment for the C implementation and a interpreter based execution environment for the Java implementation. Recently the CDj and CDc benchmarks has appeared [36] and conducting this experiment using those benchmarks is an obvious choice for further research.

Section 7 include measurements for a simple implementation of the quicksort function in both Java and C that shows that the byte code version require approx 40% less space than the version compiled into native code for a low-end embedded RISC architecture.

In terms of execution speed it is an established fact that interpretation is significantly slower than AOT. Work in [22] estimates that interpreted VMs are a factor of 2-10 times slower than native code compilers. This factor can become even larger for non-optimized interpreters. The JamVM [34], which is a highly optimized Java interpreter for high-end embedded and desktop environments, claim to achieve average execution speeds of approximately 3 times slower than native code compilers, but measurements presented later in Section 7 indicates that this number is closer to 6 times slower than native code compilers.

Stated in general terms the following observations are made

1. Interpreted Java is significantly slower than hand coded AOT compiled C
2. Interpreted Java requires less space than hand coded AOT compiled C

4.1.2 AOT Compilation

A well known example of an AOT compilation based execution environment is the GCC compiler tool chain for the C programing language, first released by its author Richard M. Stallman in march 1987 [68]. GCC translates C source code into native code for a large range of platforms of many sizes and flavors [27]. AOT compilation techniques are probably one of the best explored fields within computer science, and AOT compilers apply the following and many more types of optimizations [3]

- Dead code elimination. Only those parts of the code base that may be reached in an execution of the program are included in the executable
- Function inlining. To speed up function calling the compiler may inline a function body at one or several call sites
- Data flow analysis. To use the most efficient machine instructions, AOT compilers will make a conservative estimate on the smallest data size required for a data operation
- Register allocation of actual parameters. For suitable target platforms parameters to function calls may be placed in registers to limit memory access at function calling
- Register allocation of data values. To avoid memory access, values are allocated in registers.

Today C compilers make an excellent job of producing efficient code for low-end embedded systems, and a wide range of configuration switches can be applied to optimize code for e.g. size or efficiency. GCC is open source but several commercially available C compilers (e.g. the IAR C compiler from Nohau [44]) exist as well, improving over the excellent performance of GCC on certain specific targets.

In 1996 Per Bothner started the GCJ project [9] which is an AOT compiler for the Java language and GCJ has been fully integrated and supported as a GCC language since GCC version 3.0. GCJ builds on GCC and compiles Java source code into native machine code. Compiling an object oriented language using AOT compilation techniques goes back to Simula'67 and was further perfected in the Beta programming language [40]. Even though object oriented languages contain language constructs such as virtuallity of both methods and types, the field of AOT compiling object oriented languages is well understood.

Traditional AOT compilers compile the source language into assembler code for the target platform. An alternative and interesting flavor of AOT compilation of Java is to compile Java byte codes into C - in effect using the C language as an intermediate language. This technique has been utilized by environments such as the JamaicaVM from aicas [4], IBM WebSphere Real-time VM [24], PERC [43], FijiVM [50] and KESO [21]. The generated C code can then be compiled into native code using GCC or a similar cross compiler. Using this strategy, the FijiVM achieves execution speeds of approx. 30% slower than that of C for the CDj benchmark. This result does not imply that Java-to-C compiled code can in general be executed with an efficiency loss of only 30%. Still the CDj benchmark is a non-trivial benchmark - Section 5.3.2 shows that it requires the compilation of approx. 600 methods - and the results reported for the FijiVM indicates that AOT compilation of byte codes into C may be a feasible technique for many scenarios. Comparable results for other Java-to-C capable VMs measuring efficiency for the CDj benchmark has not been found, so no indication of FijiVM performance can be given on this basis. Section 7 will compare a subset of the above VMs using other benchmarks.

Work comparing the code size of AOT compiled Java with AOT compiled C is lacking. Because of this lack of empirical data it is assumed that there is a correlation between code size and performance and that the code size of AOT compiled Java is close to the code size of AOT compiled C. This assumption is supported by measurements presented in Section 7. These observations lead to,

1. AOT compiled Java can be almost as fast as AOT compiled C
2. The code size of AOT compiled Java is almost the same as AOT compiled C.

4.1.3 JIT Compilation

Just-in-time compilation is a technique of spreading the compilation of a program out over time, interleaving code compilation with code execution as stated in the fllowing quote [12]:

> Unlike traditional batch compilers, our compiler is invoked on a per-method basis when a method is first invoked, this technique of dynamic translation.... Our compiler generates machine code from the byte code object, and caches the compiled code for use the next time this method is invoked.

It follows that just as the interpreter has to be executing on the target alongside the program being interpreted, in a similar manner the JIT compiler has to be executing on the target interleaved with the program itself. The idea of JIT compilation has been explored long before the advent of Java. Smalltalk and Self environments are based on JIT compilation, and many important advances in JIT compilation techniques were made in those environments [12, 16].

When running alongside the program, a JIT compiler can take into account how the program is actually being executed and optimize the most used execution path. An example from the realm of object oriented languages is generating code for virtual method dispatch. At a truly virtual call site an AOT compiler cannot accurately infer which method is going to be called, since it can, and will be, different target methods from one call to another. A JIT compiler on the other hand can gather statistics and discover which method is usually called, and optimize the call to handle this scenario efficiently. This idea is called a *Polymorphic Inline Cache* and was put forward by [33] and is one example of where JIT compilers can do better than AOT compilers.

The HP project Dynamo [7] takes a stream of AOT generated machine instructions and optimizes them at runtime by taking into account optimization opportunities revealing themselves when the program is executed. Dynamo achieves considerable speedups on most benchmarks, in some cases more than 20%.

The Dynamo project shows that even after a capable AOT compiler has generated fully optimized code, a JIT compilation strategy will be able to improve further on performance.

In their paper [35] the authors conduct very detailed measurements comparing a Java-to-C AOT compiler against a selection of other Java environments (not necessarily embedded), and they find that for their AOT compiler implementation, Java code executes approximately 40% slower than when executed using the best JIT compiler (HotSpot).

For Java environments supporting dynamic class loading, a JIT compilation strategy is especially useful, since a JIT compiler is immune to dynamic code changes in the sense that previously generated code can just be discarded and new code generated on the fly.

JIT compilers exist for high-end embedded systems as well as desktop and server platforms. The CACAO JIT [39] is a well know example for embedded systems, achieving impressive execution speeds for embedded Java programs (from 1 – 1.6 times slower than C). JIT compilers tend to require a significant amount of dynamic memory, and even though the CACAO JIT can run on systems with as little as 1MB of dynamic memory [10], on low-end embedded systems with e.g. 4 kB of dynamic memory JIT compilation becomes impractical. This is mainly due to the fact that generated code will quickly fill up the limited amount of available RAM on low-end embedded devices. Thus generated code has to be stored in flash, which is difficult, but not impossible, to do at runtime. To conclude,

- JIT compilation can be at least as fast as AOT compilation, in some cases faster
- JIT compilation requires extra dynamic memory as compared to e.g. interpretation or AOT.

4.2 Execution Styles for Embedded Platforms

For high-end embedded systems JIT compilation is a very attractive execution strategy. Firstly, it is efficient. Section 7 presents detailed measurements that substantiate the claim by the CACAO authors that Java can be executed approx 1-2 times slower than C. Secondly, it supports dynamic class loading since the invalidation of existing code as a consequence of loading new code is simply a matter of recompiling the code. For low-end embedded systems though, a JIT compiler has yet to emerge that runs with as little as the few kB of RAM that is customary on low-end embedded devices. Because of the proliferation of low-end embedded systems, portability becomes an issue as well. The code generator of the JIT compiler needs to be ported to every new target device that is to be supported.

The AOT compilation strategy is very attractive for both low and high-end embedded platforms. It too is very efficient. Section 7 will show that some AOT environments are even faster than claimed above and execute faster than C on some benchmarks. AOT compilation does not require additional RAM since code generation is done ahead of execution time on a host platform. It may require more ROM memory compared to C. On low-end embedded systems

the amount of ROM is usually a lot larger than RAM, so for many scenarios AOT compilation may be useful. Especially byte code-to-C AOT compilation is interesting for low-end embedded devices. This way of compiling Java is very portable. It borrows its portability from C as this language is supported on most low-end embedded systems. So Java-to-C compilers are very portable if they do not rely on unportable external libraries. Contrary to JIT environments, environments supporting only AOT compilation will have a hard time supporting dynamic class loading at runtime, which may be a significant drawback in some scenarios. But for low-end embedded devices dynamic class loading may not be desirable, and will be hard to support, since the classes loaded will have to be placed in ROM, and it is very difficult and usually avoided writing to ROM at runtime.

Interpretation uses the smallest amount of RAM and ROM of all execution styles, but it is an order of magnitude slower than native C. In some scenarios this may be acceptable, but in others it will not. Interpreters are just as portable as Java-to-C AOT compilers if the interpreter is written in portable C code and does not rely on unportable external libraries. Interpreters will be able to handle dynamic class loading just as easy as JIT compilers, still facing the additional challenge of how to store the loaded classes into ROM at runtime.

Until a JIT compiler appears that can run with just a few kBs of RAM, interpretation and AOT compilation are the only options for low-end embedded systems. Because of these reasons an environment supporting both AOT compilation (for efficiency) and interpretation (for its low memory requirements and dynamic nature) will be an attractive architecture. The HVM, later described in Section 5, supports such a hybrid execution style where parts (or all) of the code can be AOT compiled for efficiency and the rest can be interpreted in order to save on ROM storage.

4.3 State-of-the-art Environments

A large number of environments for embedded Java exist and they utilize both JIT, AOT and interpretation. Representative examples of embedded Java environments spanning all three execution styles are described below. With the exception of KESO and HVM, none of these environments are able to run on low-end embedded systems without changes. Detailed measurements of the execution efficiency of the example environments are presented in Section 7.

4.3.1 JamVM

The JamVM [34] is famed for being the most efficient Java interpreter. The size of the interpreter is approximately 200 kB ROM. It supports the full Java specification (including JNI) and has been ported to quite a number of platforms. JamVM is written in C and applies a large range of optimizations. One of these are so called labeled gotos supported by the GCC compiler. This feature allows the use of labels as values [28] and can improve the execution time of the VM interpreter loop significantly. JamVM is built using the `configure`,

`make, make install` GNU build style known from Linux and UNIX build environments. Most other compilers (e.g. the IAR compiler from Nohau used in many industrial settings) do not support labeled gotos. Neither is the JamVM build procedure supported in many low-end embedded build environments. Finally, because of the size of the JamVM executable, the JamVM is not suitable for low-end embedded systems as is. It may be possible to port it to a particular low-end embedded target by disabling unwanted features and making an adapted build environment for the specific target. It would be interesting to attempt a port of the JamVM to the KT4585 environment described in Section 2. If such a port would be successful, the JamVM would be an attractive execution engine for this environment, and it would pave the way for porting it on other low-end embedded environments as well. The JamVM uses the GNU Classpath Java class library [29]. The size of this is approx 8 MB and in its default configuration the JamVM loads classes as required from this archive during runtime. To make this work on a low-end embedded system, a tool for generating a library archive only containing the classes used, and a facility for loading this from ROM instead of a file system would have to be developed. In any case the JamVM remains the benchmark for interpreters because of its unsurpassed efficiency on high-end embedded and desktop systems.

4.3.2 CACAO

The CACAO JIT [39] is a free and open source implementation fully compliant with the Java Virtual Machine Specification. It supports i386, x86_64, Alpha, ARM, MIPS 32/64, PowerPC 32/64 and S390 target platforms. It runs with as little as approx. 1 MB of RAM memory. It uses GNU Classpath [29] or OpenJDK [46] as Java runtime library. It runs naturally in a Linux-like OS environment with sufficient packages installed to build the JIT itself and the class library of choice. While a selection of features of the CACAO JIT can be disabled to reduce memory requirements, it is not designed for low-end embedded systems such as the KT4585 described in Section 2, and it is not obvious that it would be possible to build the CACAO JIT and required libraries for that target. Additionally a port of the code generation engine for the CR16c RISC architecture would be required. The main issue though, with JIT compilers in general for low-end embedded systems, is the runtime dynamic memory requirements.

4.3.3 GCJ

GCJ is an AOT compiler that translates Java byte codes, as they appear inside the class file, into native code for a particular target. GCJ is built on GCC and building Java programs is done in a very similar manner as when building C programs. Figure 10 illustrates the build architecture. First the Java source code is compiled into class file format. Then the GCJ compiler is used to generate native code. Since GCC supports cross compilation for a very large range of embedded targets and since GCJ builds on GCC, Java programs can

be cross compiled into native code for many different targets. In short, GCJ reuses or builds on the portability already present in GCC.

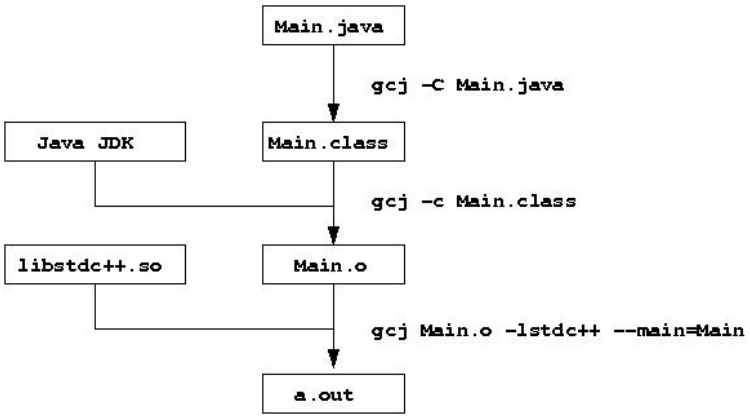


Figure 10: GCJ architecture

Still GCJ programs cannot directly run in a low-end embedded environment such as the KT4585 described in Section 2. The reason is that GCJ requires the library `libgcj` and this library is intended to be a complete J2SE implementation based on GNU Classpath making it too big for low-end embedded devices (several MBs). To solve this issue the `micro-libgcj` [30] project started, but has since been discontinued. The GCJ compiler itself (excluding the runtime environment) builds readily for low-end embedded targets. To make GCJ available - including the runtime environment - on low-end embedded devices an incremental version of libgcj with the same footprint as libgcc would be really attractive. Such a version does not exist and it is not currently possible to produce sufficiently small executables using GCJ to allow them to run on low-end embedded systems such as the KT4585. Additionally to compiling Java directly into native code, the GCJ runtime environment features an embedded interpreter. Thus GCJ supports a hybrid execution environment featuring both interpretation and static compilation. GCJ is based on some very interesting design principles (1) GCJs extreme portability (inherited from GCC) allows it to run all targets where GCC is supported and (2) GCJ supports a hybrid execution style of both AOT compilation and interpretation. The last challenge remaining before GCJ can really be applied to low-end embedded devices is to get rid of its dependency to the monolithic libgcj runtime library.

4.3.4 FijiVM

The FijiVM [50] is a AOT compiler that translates Java byte codes into C. This is a different strategy than GCJ which translates straight into native code. The generated C code then has to undergo an extra step of compilation from C into native code for the target in question. In practice this strategy gives a very high

degree of portability since C compilers most likely exist for any given embedded target. It offers a higher degree of flexibility, as opposed to the strategy chosen by GCJ, since the choice of which C compiler to use can be made by the user. In the case of FijiVM however, GCC is the preferred compiler, but it should be possible, perhaps with some changes, to use other compilers than GCC. The architecture is depicted in Figure 11.

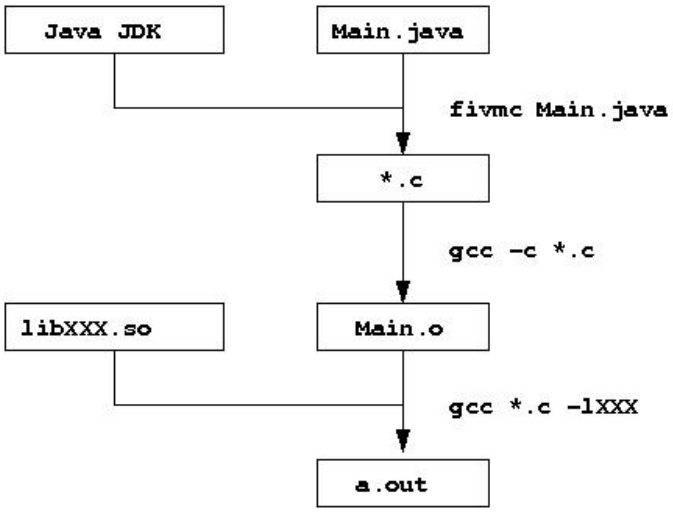


Figure 11: FijiVM architecture

It is tempting to think that using C as an intermediate stage before generating native code might incur some overhead or might make it difficult to perform certain types of optimizations applicable to the actual target. In practice though, FijiVM proves by example that the outcome of the entire compilation process can be very efficient native code. Section 7 shows that FijiVM outperforms all other execution strategies (including JIT) and on average produces the best code for all examined execution platforms. The declared contribution by the FijiVM is its real-time capabilities and the FijiVM includes a real-time garbage collector [49] and efficient support for threading and locking. The efficiency of the generated C code is another important contribution and measurements show that, on benchmarks presented in Section 7, the FijiVM is the most efficient Java execution environment for embedded systems.

The target platforms for the FijiVM are high-end embedded systems, and a 32 or 64 bit POSIX compliant OS is required. In their default configuration FijiVM executables are linked against `libgcc`, `librt` and `libpthread` and they require approx. 600 kB RAM and 1 MB of ROM. In their work [50] the authors state that the Fiji runtime is approx. 200 kB which must be the lower bound for ROM requirements of FijiVM executables. The FijiVM was not designed for low-end embedded systems, but there is no reason why the code-generation part of Fiji cannot be used to generate very efficient code for low-end embedded

targets. An interesting and very useful option for research would be to separate the FijiVM code generation module from the rest of the infrastructure and use this for generating standalone, non-monolithic, efficient executables for low-end embedded targets.

4.3.5 KESO

The KESO JVM [21] is an AOT Java-to-C compiler with the same overall architecture as FijiVM described in Section 4.3.4. While KESO does include a garbage collector, neither the GC nor the environment as such have a claimed real-time behavior as is the case with the FijiVM. The main contribution of KESO is its ability to run on low-end embedded environments such as the KT4585, an ability no other Java environments offer (apart from the HVM introduced later in Section 5). The KESO AOT compiler applies a number of optimization techniques to achieve very efficient execution speeds - almost as efficient as hand coded C. KESO is the first JVM for low-end embedded systems that supports incrementality for Java applications: you only "pay" for what you use. The price to pay for this incrementality is the lack of dynamic class loading and the lack of full Java support, as only a limited Java JDK - specific to KESO (the KESO core libraries) - is available to the programmer. Additional contributions of KESO are the concepts of isolated execution domains and its use of the OSEK/VDX operating system [31] - a well known RTOS used mostly in the automotive industry.

KESO comes very far in meeting the goals as it is both incremental and efficient. By adding just a few more features to support integratability, the KESO JVM would be a very interesting candidate for use on environments such as the KT4585.

4.3.6 JamaicaVM

The JamaicaVM from aicas [4] is a commercially available Java VM for real-time Java systems development[2]. The focus of the JamaicaVM is to ensure hard real-time execution guarantees, including real-time GC [61], while maintaining completeness and efficiency. The JamaicaVM supports interpretation as well as selective compilation of classes directly into C. Development is supported in Eclipse through the Jamaica Eclipse Plug-in. It also supports 'Smart Linking'. This is a technique based on first completing an initial run of the application during which profile information is extracted. This profile information can be used to select which methods should be compiled and which methods should be interpreted. Also it can help to remove unused code and data from the executable. Since compiled code takes up more program memory space than interpreted code, the JamaicaVM tools help to configure the application in such a way that a good performance is achieved but at a reasonable cost in terms of

[2]More information about the JamaicaVM can be found in the user manual which is available for download from their website after registration

executable size. The programmer can explicitly force the JamaicaVM to always compile selected methods.

Apart from the builder and execution engine itself, a range of analysis tools exists as well: The JamaicaVM ThreadMonitor to monitor the real-time behavior of applications, and VeriFlux: a static analysis tool to prove the absence of potential faults such as null pointer exceptions or deadlocks. The target platforms supported by the JamaicaVM are in the realm of high-end platforms. The VM itself occupies approximately 1 MB of ROM. Applications are linked against the OpenJDK Java class libraries. RAM requirements are in the range of a few MBs and upwards depending on the application.

4.4 Opportunities for Embedded Java

Section 3.5 described a simple scenario in which a single task is written in Java and integrated with an existing build and execution environment (the KT4585 described in Section 2). This section examines to which extent the environments listed above support this scenario. This identifies the gap between what the current state-of-the-art offers and what embedded software engineers accustomed to working in a C based programing environment expect.

- JamVM. Requires the addition of the source of the JamVM itself which is built using a build environment different than the one used by the KT4585 developers. Also the size of the VM (200 kB) is impractical
- CACAO. Requires the addition of the JIT itself which has further dependencies to POSIX and other libraries. The size of the JIT and other requirements makes it non incremental and impractical to include in the KT4585 build end execution environment
- GCJ. Requires the linking against libgcj which is too large for low-end embedded systems. Even if it was not, most embedded engineers would not like to add additional libraries they did not implement themselves or know in detail
- FijiVM. Requires a POSIX-like OS (uses librt, libpthread and others)
- KESO. Includes additional OSEK header files. However, the core part of the C code produced by KESO from the Java source does not rely on OSEK, so this dependency is mostly artificial. KESO does a very good job at only translating code that is actually run. KESO is incremental and does not even require libc in most cases. Currently KESO generates C code into a variable set of files with variable names. This makes it hard to include these files into an existing build environment, but the core part of the C files generated is standard ANSI C and could be compiled by any compiler. In short it should be possible without too much work to make a version of KESO that could support the above integration scenario

- JamaicaVM. The target platforms of the JamaicaVM are in the range of high-end systems, but the completeness and tool support of the JamaicaVM is far better than the non-commercial versions. It would be very interesting to explore further if a completely compiled Jamaica application, built using 'smart linking', can be linked to an executable without including the interpreter, and if this will bring the size of the application down into the range of low-end embedded systems. 'Smart linking' is based on an initial profiling run of the application and how to do this on low-end embedded systems is a challange. An interesting idea to explore is to execute the profile run of the application on a host system in a simulated environment.

- Others. All other JVMs have requirements that make them non integratable and non incremental. The most common cause is the requirement of a POSIX like execution environment and the insistence on being fully Java compliant which as a result requires the linking against a large number of external libraries.

Section 7 will show that embedded Java can be almost as efficient as C, both in terms of speed and code size. But an important opportunity for embedded Java is to acknowledge the nature of existing C based environments and enable the easy integration of Java into such environments. In other words, to be able to support the scenario described in Section 3.5 out-of-the-box while maintaining efficiency. To show that this is possible, the HVM has been implemented. The HVM is similar to KESO but makes it possible to support easy integration with existing C based execution environments. The remaining part of the thesis will describe the use, design and implementation of the HVM and measure some key performance indicators of the environment and compare these to C and other existing embedded Java environments.

5 The HVM - Design

The HVM is a Java execution environment for low-end embedded devices, such as the KT4585. It is the platform for and outcome of the experimental work done during this thesis. Work was begun in the autumn of 2010. The main goal of the HVM project is to be able to support the integration scenario from Section 3.5 on the KT4585.

Before describing the design and methods used in the implementation of the HVM, the outcome of the requirements analysis of Section 3.5 is translated into these measurable design goals for the HVM,

1. It must be able to execute with a few kBs of RAM and less than 256 kB ROM
2. It must be integratable with the KT4585 build and execution environment, i.e. it must be compilable together with existing C source for the KT4585 and it must be possible to schedule Java code alongside existing C code using the KT4585 RTOS
3. Java code should execute nearly as efficiently as C
4. It should be possible to control hardware directly from Java
5. It should be possible to handle 1st level interrupts in Java
6. It should be possible to use any Java API, and not tie Java into some particular API
7. Java code should be able to easily read and write data in the C layer
8. It should be easy and straightforward to program Java for the KT4585 and translate it into a format suitable for compilation and download.

The following Section 5.1 will give a brief demonstration of how the HVM is used seen from a programmers perspective. After this use scenario, Section 5.2 describes the methods used in the implementation of the HVM to achieve the design goals.

5.1 HVM Demonstration

The example code in Figure 12 is used below to give a demonstration of how the HVM works from the programmers perspective.
This small program inserts 5 words into an ArrayList, sorts them and then checks that they are sorted. If the `test` method returns true, an error occurred. The HVM is able to compile this test program into C code and eventually run it successfully on the KT4585 platform or even smaller (8 bit) platforms. The HVM build environment is implemented in Java as an Eclipse plugin, and the user experience is best if Eclipse is used. Eclipse is a very common IDE for Java and other kinds of software development. The HVM build environment

```
package test.icecapvm.minitests;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Iterator;

import devices.Console;

public class TestArrayList {

    public static void main(String[] args) {
        boolean failed = test(args);
        if (failed) .....
    }

    public static boolean test(String[] args) {
        ArrayList<String> list = new ArrayList<String>();

        list.add("hello");
        list.add("to");
        list.add("you");
        list.add("Stephan");
        list.add("Korsholm");

        Object[] array = list.toArray();
        Arrays.sort(array);

        list = new ArrayList<String>();
        for (int i = 0; i < array.length; i++) {
            list.add((String) array[i]);
        }

        if (array.length == 5) {
            Iterator<String> sortedNames = list.iterator();
            String previous = null;
            String next;
            while (sortedNames.hasNext()) {
                next = sortedNames.next();
                if (previous != null) {
                    if (previous.compareTo(next) > 0) {
                        return true;
                    }
                }
                Console.println(next);
                previous = next;

            }
            return false;
        }
        return true;
    }
}
```

Figure 12: HVM Example

can also be executed from the command line without the use of Eclipse. To compile and run it using Eclipse, the user must make a new standard Java project. It is not required to make a special kind of 'embedded' Java project - a standard Java project is sufficient. This will most likely add a dependency to the JDK currently applied by the user, this could be the standard JDK from SUN, OpenJDK, GNU Classpath or some other vendor specific JDK. All that the HVM requires for the above to work is that the `java.util.ArrayList` and other classes used in the above program are available. Figure 13 illustrates how the user can activate the HVM build environment to compile the above program.

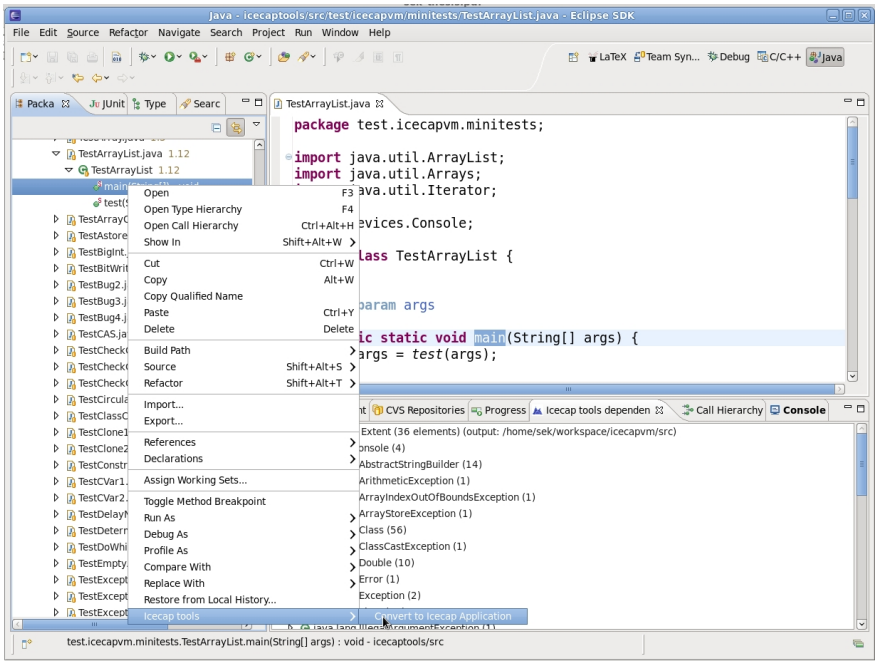


Figure 13: Use Scenario

By right clicking on the main method of the program the user can activate the HVM builder from a popup menu. The result will be that the Java program is translated in to a C program which is placed in an output folder chosen by the user. This C code is written in strict ANSI C and is completely self contained, making it possible for the user to include in any existing build environment he may be using for his particular embedded platform. How exactly the code will eventually get called and executed is the topic of a later Section.

After the Java-to-C code generation has been completed the next step for the user is to activate his usual C build environment for his embedded target, and eventually download the resulting application to the target.

This cycle of (1) Editing Java code, (2) Activating the HVM Java-to-C translation, (3) Activate native build environment and finally (4) Download and run application on target, will now be the everyday software development cycle of the user.

In many cases Eclipse will also be the preferred development environment for developing C code for the target - this is in fact the case for some Polycom developers working with the KT4585 - in which case all source code, both Java and C, will be available to the user in the same IDE.

An important part of the Eclipse integration is the view, entitled *Icecap tools dependency extent*. Figure 14 zooms in on this view and shows parts of all dependencies of the example application listed in Figure 12.

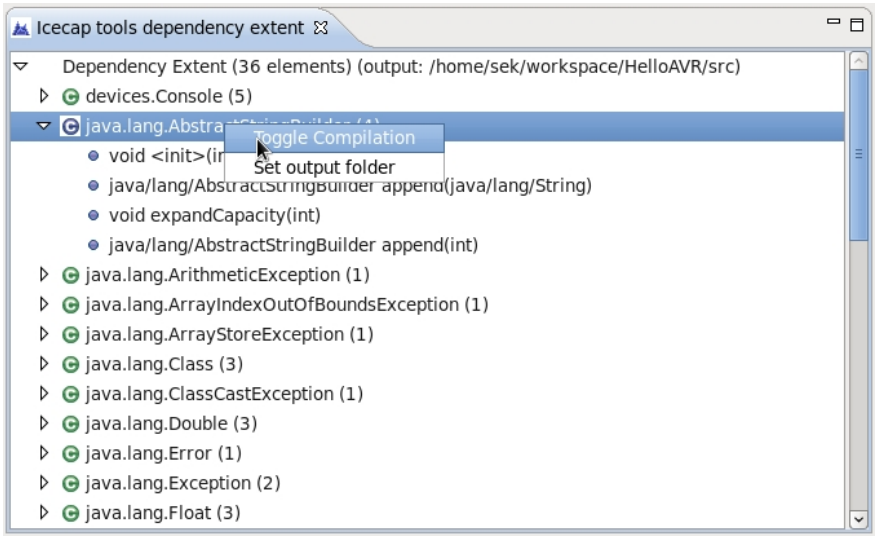


Figure 14: Browsing dependencies

The view lists that a total of 36 classes have been translated into C. Not all methods available in the source of these classes are translated, but only those that may be used by the application. In the above example the class `java.lang.AbstractStringBuilder` has been expanded to show the 4 methods that are included in the set of dependencies. Also visible in the view is the output folder where the generated C source is placed. All methods under `AbstractStringBuilder` are marked with the blue color. This indicates that the programmer has marked these methods for compilation ("Toggle Compilation"). Unmarked methods will be interpreted.

5.2 Methods

This section describes the most important methods used to meet the design goals for the HVM. The main part of the HVM is the Java-to-C compiler and this is also the part containing the most complexity and functionality. In summation most methods presented below are standard methods from within the realm of compiler construction and program analysis.

5.2.1 Intelligent Class Linking

One of the most important features of the HVM is its ability to find the dependencies of a given application and thus perform the act of program specialization and optimize used libraries and profiles for a given application.

To perform this analysis the HVM makes one major simplification which is to preclude the use of dynamic class loading. In other words the HVM is only able to handle statically configured applications that does not rely on dynamic

class loading. This restriction of normal Java functionality is also made by the KESO and FijiVM environments.

Starting from the main entry point of the program, the HVM build environment performs a static whole program analysis and calculates a conservative estimate of all classes and methods that may be entered at runtime. This set of classes and methods called the *dependency extent*. Calculating a safe but good approximation to the dependency extent allows the HVM to keep the size of the resulting application down so that the 1st primary requirement is fulfilled.

The dependency extent of the example program in Figure 12 is comprised of 36 classes and 90 methods. Apart from the `test` method itself, programmed by the user, the rest of the methods in the dependency extent originates from the Java JDK, e.g. the used methods in `java.util.ArrayList` and from the `Arrays.sort` method and its dependencies. To calculate the dependency extent, the HVM build tools scans the byte code starting from the main entry point and follows all possible paths from there. Two interesting cases highlight how this works (1) a standard control flow branch case, illustrated in Figure 15 and (2) a method call control flow case, illustrated in Figure 16.

```
if (condition)
{
     if-part....
} else {
     else-part
}
```

Figure 15: If branch

When an if-branch is encountered, the analysis proceeds through both the condition (which might be an elaborate expression), through the if-part and through the else-part. All code encountered by following these 3 possible flows through the program is added to the dependency extent. A careful static value analysis to determine if both the if-part and the else-part can indeed be entered at run time is not performed currently by the HVM tools. This method is straight forward and clearly a safe conservative estimate of the dependency extent stemming from if-branches.

```
A a = getA();
a.foo();
```

Figure 16: Invoke branch

Opposed to the flow of control inherent in if-statements as above, predicting the flow of control for a method invocation is not straight forward. The reason is because of the virtuallity of method invocations in Java. In Figure 16 it is not statically possible to determine which foo-method gets called, and thus where the flow of control might proceed. The reason is that the method `foo` is a virtual

method and may be implemented in several subclasses of `A` and it is unknown from just looking at the call site what the runtime type of the object `a` may be. The method the HVM applies to solve this problem is to keep track of which possible subclasses of `A` may have been instantiated up until the call of `foo`. If it is possible to know which subclasses of `A` may have been instantiated earlier, the possible set of methods that may get called at this call site is known and analysis can proceed by adding all of them to the dependency extent and visit them recursively. It is crucial now to observe that following these new paths might lead us to discover that new subclasses of `A` could be instantiated. In that case, and if the flow of control ever returned to the above call site, the analysis for that call site must be done again. This may in turn lead to even more subclasses of `A` being added to the dependency extent. The HVM dependency analysis marks for each virtual call site the list of classes that may have been instantiated prior to execution of the call site. If analysis reencounters the same call site, the current list of instantiated classes is compared to the list in effect at the last visit to this call site. If the lists are equal the analysis terminates, if new classes have been added the analysis is redone.

Following this method the analysis arrives at a conservative estimate of the possible targets of the virtual call. This method will indeed terminate because of the following arguments: The set of classes in the dependency extent is not infinite. Each time the analysis arrives at the call site it will either have added at least one class to the list of instantiated classes or no new classes has been added. If no new classes have been added, the analysis is done and the analysis of this call site terminates. If one or more new classes have been added the analysis is repeated, but new classes cannot be added indefinitely since the set of possible classes to instantiate is not infinite.

The method described here is an incarnation of the *Abstract Interpretation* method described in e.g. [42] chapter 1.5. For each virtual call site the set of *traces* is collected. This is intuitively the ways that program flow may reach the call site. For each of the collected traces there will be a finite number of classes that may have been instantiated. The total number of all classes instantiated along all possible traces is the set of possible types for `a` in the example from Figure 16. Static type inference of method calls for object-oriented languages is a well known field and described in e.g. [51]. The method can also be viewed as an instance of the k-CFA algorithm as described in [60]. The way the control flow at virtual method call sites is handled, is actually what is called a *variable-type analysis* introduced by [62].

Even though the above method is correct in producing a safe conservative estimate and also terminates it may not be a practical method. The time complexity for k-CFA is EXPTIME [67], but in practice it is possible to calculate the dependency extent of large non-trivial programs, which the above example in Figure 12 also indicates. Even so, when utilizing library code like e.g. `java.util.ArrayList` the developer commonly encounters *dependency leaks*. As an example consider the Java HelloWorld program from Figure 17.
Analyzing this program using the method applied by the HVM will result in a dependency leak. The reason is that use of the `System` class involves the

```
class HelloWorld {
  public static void main(String[] args)
  {
    System.out.println("HelloWorld!");
  }
}
```

Figure 17: HelloWorld Dependency Leak

analysis of the class initializers in class `System`, and using the standard JDK from Sun or any of the other vendors this in turn requires the initialization of 'the rest of the world' to put it in popular terms. The HVM tools will actually run and attempt to collect the dependency extent, but will eventually give up. For the Java JDK 1.6 the `java.util.*` does not leak and can be successfully analyzed by the HVM tools.

In a low-end embedded scenario, such as the KT4585, dependency leaks are not an issue. If it were possible to calculate the dependency extent of the `System` class it would be of such a magnitude that compiling it for a low-end embedded target would be impractical.

5.2.2 Self-contained Source Code Generation

Another important feature of the HVM is its independence of external libraries. This is crucial to keeping the size of the application down to a level were it can be deployed on low-end embedded devices, and it is crucial for supporting integration into an existing build environment.

An example of were dependencies may seep into the HVM is in the handling of the `long` arithmetic. The `long` data type may not be implemented in the C runtime for e.g. 8 or 16 bit architectures - e.g. the IAR compiler runtime for the KT4585 does not support `long` arithmetic. To get around this problem the HVM implements `long` arithmetic internally to support the `long` byte codes. The `float` and `double` data types exhibit the same problem, but internal support for these data types have not been added yet. So for `float` and `double` a compromise has been made and these data types are only supported if they are supported by the underlying C runtime.

In most cases the code produced by the HVM can be compiled and linked without the presence of a C runtime library (-nostdlib). If the C programmer is using `float` and `double` already these will be supported by his existing C runtime and he will also be able to use them in Java space. The embedded interpreter, also part of the HVM, has this property as well: no linking against other libraries are required.

Java exceptions are supported and implemented internally in the HVM. It would be tempting to use the `setjmp` and `longjmp` functionality available in some C runtimes but this has not been done as it would go against the principle of not adding dependencies that may not be supported by a given build environment. Exceptions can be thrown and caught in both AOT compiled and

interpreted code and flow across the boundaries of these execution types. The cost for full support of Java exceptions is approx 25% in terms of execution time. This is a significant cost, and for that reason it would be desireable to allow the developer to selectively exclude exception support from the HVM. On some low-end embedded systems, e.g. where all I/O is handled through hardware objects and 1st level interrupt handlers, the occurence of an exception may be a fatal error from which there is no graceful recovery. Excluding exceptions from released code, in such scenarios, while maintaining them during development for debugging purposes might be a desirable feature. Still, the HVM supports exceptions, and they can be used in scenarios where they are desired.

The HVM does not use `malloc` or `free`. Instead all data are allocated statically, including the Java heap. Dynamic data, both Java data originating from using the `new` keyword, and runtime allocations made by the HVM internals - required in some special cases - are allocated in the heap.

These are some examples of how the design principle of not including any external dependencies has been followed in the HVM. As a result the C code produced by the HVM can be further compiled by any cross compiler and included in all build environments. Not adding any dependencies is perhaps the most important attribute of the HVM when it comes to supporting integratability.

In short, if the HVM is able to calculate the dependency extent of a given application, there is a very good chance that the source code can be compiled into an application sufficiently small to deploy to a target such as the KT4585.

5.2.3 ROM/RAM aware

To reduce the RAM requirement to a minimum, the HVM is careful to place static data in ROM. As a result the only data placed in RAM is static class variables and the heap. The size of the heap can be configured at compile time. If no data is allocated during a program run, then no heap is required. Still a small amount of RAM is needed for the Java stack and some internal HVM bookkeeping.

Java objects are packed according to their field variables. Additionally to actual object content, the HVM adds a 16 bit hidden field to allocated objects. 10 bits of this field is a reference to the type (class) of the object, the remaining 6 bits are reserved for future use. As a result the HVM does not support the use of more that 1024 classes in any given application.

5.2.4 Hybrid Execution Style

The HVM includes an embedded interpreter, and the software developer can inside Eclipse mark which methods should be interpreted and which methods should be compiled into C. As mentioned in Section 4.1.1 there is some debate if the space required to store byte codes is less than space required to store natively generated code, in the general case. In the case of the HVM, byte codes take up approximately 50% less space than AOT generated code. For this reason the option to execute selected methods using interpretation has been added. The

developer can select a limited amount of methods for AOT compilation and keep the rest as interpreted. This way ROM space requirements can be reduced.

The interpreter is dynamically adapted to which byte codes are actually being marked for interpretation. This means that interpreter functionality for byte codes that are not used, is not included in the application. In many cases only a limited subset of all byte codes are used, so this feature can potentially save a significant amount of ROM space. E.g. many programs for low-end embedded devices does not utilize the `long` data type - for such programs those parts of the HVM that implements `long` arithmetic are not included in the executable.

The way to achieve this selective exclusion of unused functionality is for the static analyzer to record which byte codes are part of the dependency extent and which of those have been marked for interpretation. This knowledge is expressed as a set of compiler flags that are either defined or not defined, e.g. the following excerpt from the HVM interpreter illustrates this,

```
#if defined(LMUL_OPCODE_USED)
case LMUL_OPCODE: {
    unsigned char topInc = handleLMUL(sp, code);
    sp -= 4;
    if (topInc == 0) {
        initializeException(sp,
                            JAVA_LANG_ARITHMETICEXCEPTION,
                            JAVA_LANG_ARITHMETICEXCEPTION_INIT_);
        sp++;
        code = ATHROW_OPCODE;
        break;
    }
    sp += topInc;
    method_code++;
}
continue;
#endif
```

Whether or not the flag `LMUL_OPCODE_USED` is defined, is stored in an auto generated file produced by the HVM dependency analyzer. Additionally the same selective exclusion of the `handleLMUL` function is in effect.

The methods described up until now all contribute to the HVM adhering to the 1st primary requirement. Now follows a describtion of which methods are applied to support integratability.

5.2.5 Easy Activation

The HVM main entry point can be called directly from existing C code:

```
initVM();
excep =
 enterMethodInterpreter(&methods[mainMethodIndex], mainMethodJavaStack);
```

The function `initVM` is part of the HVM infrastructure and must be called just once before control flow enters Java space. The function `enterMethodInterpreter` activates Java code. In the above scenario the main method of the Java code will be called. The `enterMethodInterpreter` function will determine if the method being activated is AOT compiled or interpreted and start execution accordingly. The `methods` array and the `mainMethodIndex` variable is auto generated by the HVM. The `mainMethodJavaStack` has to be allocated by the user and supplied as the stack to use for Java code. This allows existing C code to call the HVM in an easy way at any time and it is e.g. utilized to support the addition of Java event handlers to a legacy RTOS and to handle 1st level interrupts, both of which are described in Section 7 below. The absence of new dependencies and the ability to enter Java space easily from existing C code makes it straightforward to compile a simple piece of Java code and include it in an existing C based application.

5.2.6 Code efficiency

The efficiency by which the AOT generated C code can be executed hinges on a range of optimizations performed by the HVM AOT compiler. The following section highlights three of the most important optimizations made in the HVM: stack to variable allocation, producer-consumer analysis and devirtualization.

The JVM is stack based, which means that all values, both method parameters, method local variables and intermediate values are kept on the stack. A straight forward implementation of the individual byte codes will give rise to frequent stack access to push and pop values from the stack. This memory access to simulate the stack as it evolves is resource consuming, thus most JVM implementations seek to simulate stack behavior in some other manner than actually accessing the stack. The HVM attacks this challenge by mapping each stack cell to a C variable. Rather than reading from/writing to the actual stack cell in memory, the HVM AOT compiler generates C code that caches each stack cell in a C variable. For short stack sizes the resulting code is of a nature where the C compiler can store the variables in registers, thus avoiding costly memory accesses. Still, for stacks sufficiently large where most stack cells are accessed within a short span of code the C compiler will have to use the C stack for variables as it runs out registers. To illustrate this technique the following is an example of a simple sequence of byte codes performing an addition,

```
ICONST_1
ISTORE_0
ICONST_2
ISTORE_1
ILOAD_0
ILOAD_1
IADD
ISTORE_0
```

These byte codes access two local variables and two stack cells. In the HVM both local variables and stack cells used for computation are kept on the same

stack. Just before the execution of the `IADD` opcode the stack will look like depicted in Figure 18.

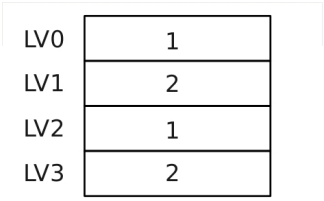


Figure 18: Stack example

By mapping each stack cell to a local variable, the HVM will generate the following code for the above byte code sequence:

```
LV2 = 1;
LV0 = LV2;
LV2 = 2;
LV1 = LV2;
LV2 = LV0;
LV3 = LV1;
LV3 = LV2 + LV3;
LV0 = LV3;
```

First, even though the JVM is a stack based architecture no stack access takes place in the above. This is very important for efficiency, since memory access is very expensive and requires a long sequence of native instructions, especially on 8 and 16 bit architectures. Secondly it seems inefficient to use so many variables for such a simple calculation, but fortunately the C compiler that eventually translates this into machine code will be able to optimize this into a very efficient format.

An alternative way to avoid the frequent stack access inherent in a stack based VM like the JVM is to translate the Java byte code into SSA format (Static Single Assignment) format [15]. This is done by the SOOT framework [66]. The code generated by the HVM is not in pure SSA format, but it is a sequence of assignments and simple calculations that do not access the Java stack directly. The HVM then leaves the optimization of the resulting statements to the C compiler - an optimization which all C compilers do well, and which e.g. GCC has evolved to perfection over many years of optimization.

How the stack cells should be allocated to C variables is decided by a method local static analysis of the byte code. At joining paths the stack-to-variable mapping is merged in cases where this is possible, or in some cases flushed to the actual stack if no simple merge can be done. If a joining path has a very different mapping than the joined path, it may be required to flush some or all cached variables to the actual stack and reload them according to the path into

which execution is joined. The way this is handled in the HVM is ad hoc and can most likely be improved.

At entry into an exception handler the stack-to-variable mapping is completely flushed. This may make jumps to exception handlers ineffective.

5.2.7 Code efficiency - Producer-Consumer Analysis

Mapping stack cells to C variables as above lowers the number of stack access during method execution. Still the question remains which data types to use for the C variables. The Java stack is 32 bit wide, but in many cases it is possible to infer a more narrow data type, e.g. `short` or `byte` for a given stack cell. Performing this optimization is crucial for 8 or 16 bit architectures, since 32 bit data manipulation requires a large amount of instructions on a 16 bit architecture, and even larger amount on a 8 bit architecture. The better one can choose the proper data type for a data manipulation the smaller the resulting code will be (saving on ROM) and the more efficient the resulting application will be able to execute.

When accessing local variables and formal parameters it is straight forward to narrow the type according to the declared type of the variables. When loading actual parameters for method calls the same method can be applied: when loading a value onto the stack that is going to be passed to a method, the proper data type can be found by looking at the signature of the method.

These observations have given rise to an analysis performed by the HVM that termed *Producer-Consumer Analysis*. The purpose of the analysis is for each byte code to infer the most appropriate data types for stack cells manipulated by the byte code. Consider the final `ISTORE_0` byte code from the example in Section 5.2.6. This byte code is storing the result into local variable number 0. If the size of the local variable 0 is known to be 1 byte, then it is not necessarry to store all 4 bytes into LV0. Also there is no need to perform a full 32 bit addition when implementing the `IADD` functionality.

The following description of the details of producer-consumer analysis is based on these observations,

1. Java byte codes are either producing stack cells (e.g. `ICONST_0`), consuming stack cells (e.g. `ISTORE_0`), both consuming and producing stack cells (e.g. `IADD`), or not producing nor consuming stack cells (e.g. `GOTO`)

2. No matter along which program trace execution arrives at a byte code, the height of the Java stack is always the same.

3. A stack cell may have been produced by many byte codes, but it is only going to be consumed by one byte code.

Observation (1) above is obvious by looking at the Java spec for all byte codes. Observation (2) is a structural constraint on JVM code as defined by the JVM specification. Observation (3) is not obvious, but the HVM has so far not come across programs where this is not the case. Should it turn out that (3)

does not hold, it is possible with limited effort to support it in the producer-consumer analysis. The reason for it not being supported is purely practical: no benchmark or program has exhibited this behavior so far. Producer-consumer analysis as implemented in the HVM is a method local analysis and is not carried over across method invocations. For each byte code it calculates all possible traces that lead up to the execution of the byte code. Along each trace it collects all byte codes that may have produced the stack cell(s) on the stack before executing the byte code. The set of traces are calculated using a form of abstract interpretation in the same manner as calculating the dependency extent described in Section 5.2.1, but in this incarnation the handling of method calls is simple, since traces are not calculated across method call boundaries.

The result of the producer-consumer analysis is that each byte code is annotated with a data structure describing all producers and consumers of all stack cells present on the stack before execution of this byte code. This information is used by the Java-to-C compiler to use the most narrow data type for all data manipulation.

If the Java software developer uses appropriate data types like `byte` and `short` whereever possible it will have a significant impact on code size and efficiency on 8 and 16 bit architectures. On 32 bit architectures the effect will not be as significant.

Producer-consumer analysis is not a constant propagation analysis. In the simple example from Section 5.2.6, producer-consumer analysis will be able to allow us to use the `byte` data type for storing the result of the addition into local variable 0 only if it can see that the data type of LV0 is `byte`. A constant propagation analysis would furthermore be able to gain knowledge of the data range of values from looking at actual values placed on the stack. The HVM does not use constant propagation analysis.

5.2.8 Code efficiency - Devirtualization

A useful side effect from calculating the dependency extent as described in Section 5.2.1 is that for each method call site a list of possible classes implementing the callee is calculated. A special case is, if this list contains one member only. In that case the method call is not really virtual but can be substituted for a direct call. This will allow the Java-to-C compiler to generate a call directly to the C function generated from the callee. This type of call is significantly more efficient than going through a virtual dispatch table, and additionally it will allow the C compiler to perform inlining of the call. In the HVM the `invokevirtual` Java byte code is substituted for the `invokespecial` Java byte code at call sites that are deemed non-virtual. This will in turn make the Java-to-C compiler generate a direct method call when translating the `invokespecial` byte code.

The effects of devirtualization have been thoroughly examined in [21] and [35].

5.2.9 Code efficiency - Other Optimizations

The HVM applies a number of other standard optimizations,

- Null pointer checks. Checks for pointers being null can be avoided if it can be statically determined that the pointer cannot be null. E.g. the `this` pointer in non-static methods cannot be null, as this would have caused a null pointer exception when the method was called
- Parameter passing. Non-virtual methods are translated into C functions and formal parameters for the method are translated into corresponding formal parameters for the C function. When calling an interpreted method from a compiled method, this cannot be done. Instead the actual parameters have to be placed in the stack
- Return values and exceptions. The HVM supports Java exceptions. A consequence of this is that methods must, apart from returning their return value (if not void), also return an indication of if an exception occurred. In the HVM the exception state is packed together with the return value. If the resulting data type becomes wider that 32 bit the return value is placed on the stack.

5.2.10 Hardware Objects

Hardware Objects according to [57] are supported in the HVM, both for interpreted and compiled methods. Hardware Objects is a facility for accessing raw memory from Java space, but it's often used to control IO devices through access to device registers of the underlying micro controller. On many embedded targets (e.g. the ATMega2560 from Atmel) the access to device registers has to take place using special purpose machine instructions, a read or write through a load/store instruction will not have the desired effect. For this reason the HVM cannot make an implementation of hardware objects that simply accesses memory using standard C idioms. The HVM delegates the actual access to native C functions implemented in a thin hardware specific interface layer. The function for writing a bit to IO has the following signature,

```
void writeBitToIO(int32 address,
                  unsigned short offset,
                  unsigned char bit);
```

An implementation of this function now has to be given for each target. When access to fields of Hardware Objects is done from Java space, the interpreter or Java-to-C compiler will make sure that appropriate functions are called. If enabling inlining in the C compiler, the function will be inlined and executed very efficiently.

5.2.11 1st Level Interrupt Handling

1st level interrupt handling as described in [38] is a facility for handling hardware interrupts in Java space when they occur and not at a later point in time. The

HVM has been designed to be interruptable almost anywhere, and in the rare cases where it is not, interrupts are disabled around critical regions. Combined with the facility for entering Java space easily, as described in Section 5.2.5, the ISR written in C or assembler can be very thin and simply call Java space directly to handle the interrupt.

5.2.12 API Independence

The HVM does not put any requirements on the JDK used in Java programs. When creating a new embedded Java project using e.g. Eclipse, the software developer can build upon a JDK of his own choice. When the HVM computes the dependency extent of an application, byte codes are loaded using the BCEL API [5], and it makes no difference if one or the other JDK is used. The HVM will simply load any byte code it encounters and translate it into C.

5.2.13 Native Variables

The main purpose of the HVM is to support the stepwise addition of Java software components into an existing C based environment. It follows that the ability to read C data from Java space and vice versa becomes very important. Traditionally this is supported in Java through native methods, alternatively through Hardware Objects. The HVM adds a third option termed *native variables*. The basic idea is to map static Java class variables to C variables and have the Java-to-C compiler produce code that reads and writes to the corresponding C variable instead of the class variable as would be the usual case. This mapping can be controlled by the developer through the `IcecapCVar` annotation, the definition included here:

```
public @interface IcecapCVar {
        String expression() default "";
        String requiredIncludes() default "";
}
```

As an example of its use, consider the global C variable `uint32 systemTick;` part of the KT4585 programmers framework. This variable is continuously updated by the KT4585 RTOS every 10 ms and used to measure system time. Using native variables this counter can be accessed directly from Java space in the following manner:

```
public class XXX {
    @IcecapCVar(expression ="systemTick",
                requiredIncludes = "extern uint32 systemTick;")
    static int tick;

    public void test()
    {
        if (tick % 100 == 0)
        {
            ...
        }
    }
}
```

The optional `expression` attribute defines to which C expression the static variable is mapped and the optional `requiredIncludes` attribute allows the developer to add includes or external declarations required to compile the `expression`. Native variables can only be accessed from compiled code.

5.3 Results

This section contains two example programs that illustrate the results of applying the methods described above. The first example is a very simple example showing how the integration scenario described in Section 3.5 can be achieved, the other example is a more elaborate example that illustrates how the HVM scales to large applications.

5.3.1 Integrating With an RTOS

In this example a new task written in Java is added to an existing schedule of the KT4585. In its default configuration the KT4585 runs 17 tasks written in C, each implementing various parts of the DECT protocol stack. The purpose of the new Java task is to control the DIP (see Section 2.2) by stopping or starting it. The DIP is stopped on the KT4585 by clearing a particular bit in a device register at address 0xFF6006, and it is started by setting the same bit. Figure 19 shows the full implementation of our Java RTOS task.
Actual control of the DIP is achieved through the use of the `DIP_CTRL_REG` hardware object. To be able to send events to this task from the other tasks implemented in C, a proxy task is registered with the RTOS. In the KT4585 this is accomplished using the following piece of C code:

```
DIPCONTROLLERTASK = OSRegisterTask(dipControllerProxy);
```

```
public class DIPController {
   public static final byte STARTDIP = 0x10;
   public static final byte STOPDIP = 0x11;
   public static final byte INITTASK = (byte) 0x99;
   public static final short URST = 0x0080;

   private static class Port extends HWObject {
      public short data;
      public Port(int address) {
         super(address);
      }
   }

   private static Port DIP_CTRL_REG;

   @IcecapCompileMe
   static boolean handleEvent(byte primitive) {
      boolean handled = true;
      switch (primitive) {
         case INITTASK:
            DIP_CTRL_REG = new Port(0xFF6006); break;
         case STARTDIP:
            DIP_CTRL_REG.data |= URST; break;
         case STOPDIP:
            DIP_CTRL_REG.data &= ~URST; break;
         default:
            handled = false;
      }
      return handled;
   }
}
```

Figure 19: RTOS task written in Java

And the content of the C proxy task is:

```
static uint32 stack[10];

void dipControllerProxy(MailType *mail) {
   stack[0] = mail->Primitive;
   main_DIPController_handleEvent(stack);
}
```

The function `main_DIPController_handleEvent` has been automatically generated by the HVM, and it contains C code that implements the functionality of the `handleEvent` method from Figure 19. The HVM generates a handful of C files - always with the same names - and if these files are added to the existing build environment, adding a new task written in Java to the existing schedule

of the KT4585 has been accomplished.

- After this setup has been done, all future work with extending the behavior of the Java task can take place entirely in Java
- The C source generated from the Java task is completely self contained and does not add any further dependencies to the application
- The Java task needs a heap. This is included in the auto generated C files. The size of the heap can be set by the developer.

This example illustrates how the HVM supports the design goal of integratability. How well it behaves in terms of efficiency is the topic of Section 7.

5.3.2 Full SCJ application

This section illustrates the maturity level of the HVM by showing how it can run the miniCDj benchmark from [52]. This benchmark is built on top of the SCJ profile. The HVM supports the SCJ profile Level 0 and level 1 [63]. The SCJ profile offers a scoped memory model and preemptive scheduling mechanism. The HVM implements these features almost entirely in Java using Hardware Objects, native variables and 1st level interrupt handling. The implementation is described in detail in a paper accepted for the JTRES'12 conference. The paper has been included in the Appendix.

The HVM can fully analyze, compile and run the miniCDj benchmark on 32 and 64 bit Intel platforms, but the benchmark requires a backing store of at least 300 kB, so it will not be able to run on a low-end embedded system. Still it will compile for a low-end embedded system and show how well the HVM program specialization can keep the ROM foot print down.

To demonstrate RAM requirements, a simple SCJ Level 1 application consisting of 1 mission and 3 periodic handlers scheduled by a priority scheduler, is run. This application can run with a backing store of approx 8 kB thus allowing us to deploy it on the KT4585.

After some minor adjustments the miniCDj benchmark compiles against the `javax.safetycritical` package from the HVM SCJ implementation. As JDK the OpenJDK 1.6.0 class libraries has been used in this evaluation. After the HVM program specialization has optimized the application a total of 151 classes and 614 methods are included in the final binary. These classes are divided between the packages as described in Figure 20.

Since the KT4585 C-runtime does not support `float` and `double` - two data types used heavily by the miniCDj benchmark - the generated C code was compiled for a similar platform with `float` support: the AVR ATMega2560 platform from Atmel. This is a 8 bit architecture with 8 kB of RAM and 256 kB of flash. The code was compiled using the avr-gcc compiler tool chain [6] .

The resulting ROM requirements are listed in Figure 21. Results are listed for a *mostly interpreted* and for a *compilation only* configuration.

	Classes	Methods
`java.lang.*`	46	171
`java.util.*`	10	42
`javax.safetycritical.*`	46	185
`minicdj.*`	49	216
Total	151	614

Figure 20: Program specialization results

	ROM
Mostly interpreted	94682
Compilation only	282166

Compiling the miniCDj benchmark for an 8 bit low-end device (ATMega2560). Using the HVM and the avr-gcc compiler tool-chain. Numbers in bytes.

Figure 21: HVM-SCJ ROM requirements

Using the *mostly interpreted* configuration the ROM meets the goal with a large margin and is well below the 256 kB available on the ATMega2560. Using the *compilation only* configuration the resulting application is approximately 276 kB and no longer fits onto the ATMega2560.

The reason for the difference in ROM size between the compilation and interpretation configuration is, that C code generated by the HVM Java-to-C compiler requires more code space than the original Java byte codes. Whether this is a general rule cannot be inferred from the above, and if the HVM Java-to-C compiler was able to produce tighter code the difference would diminish. But this experiment has an interesting side-effect and shows, that in the particular case of the HVM, the hybrid execution style supports the running of programs on low-end embedded devices, that would otherwise not fit on the device.

The work reported in [52] shows results from running the miniCDj benchmark on the OVM, but it does not report a resulting ROM size. It does state however that the benchmark is run on a target with 8MB flash PROM and 64MB of PC133 SDRAM - a much larger platform than the ATMega2560.

In the simple SCJ application with 1 mission and 3 handlers the RAM usage can be divided into the parts shown in Figure 22. The stack sizes and the required sizes for the SCJ memory areas were found by carefully recording allocations and stack heights in an experimental setup on a PC host platform. The results from compiling the application for the KT4585 using the gcc cross compiler for the CR16c micro-controller (this benchmark does not utilize `float` or `double`) is shown below,

The results show that a total of approx 10 kB RAM are required. The ROM size of the application is approx 35 kB. These numbers allows us to run SCJ applications on low-end embedded systems such as the KT4585.

SCJ related	bytes
'Main' stack	1024
Mission sequencer stack	1024
Scheduler stack	1024
Idle task stack	256
3xHandler stack	1024
Immortal memory	757
Mission memory	1042
3xHandler memory	3x64 = 192
HVM infrastructure	
Various	959
Class fields	557
Total	9715

Figure 22: HVM-SCJ RAM requirements

6 The HVM - Implementation

The HVM Java-to-C compiler is implemented in Java. It can be deployed as an Eclipse plugin or it can run as a standalone Java application. When run as an Eclipse plugin the developer can select from inside the Eclipse workbench, which method is going to be the main entry point of the compilation. The calculated dependency extent is displayed in a tree view inside Eclipse, allowing the developer to browse the dependency extent and click on various elements to see them in the Eclipse Java code viewer. When run from the command line, input to the process is given manually to the compiler.

An overview of the compilation process is depicted in Figure 23.

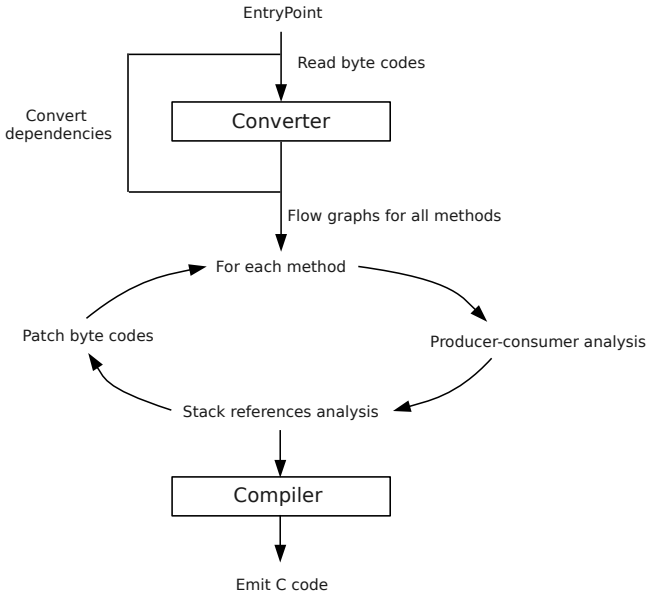


Figure 23: Compilation sequence overview

The entry point - e.g. the `main method` or the `handleEvent` method of a task - is the input to the process. The HVM converter will read the byte codes from the entry point and convert each byte code into a node in the control flow graph of the method. These nodes will in following visits of this graph be annotated with information used by the compilation process. While constructing the control flow graph, required dependencies are put on a stack of work items and will give rise to further methods being loaded, until the full dependency extent of the main entry point has been loaded. The details of identifying the dependency extent is explained in Section 5.2.1.

After all methods have been converted into flow graphs, each flow graph is visited several times performing various analysis on the graph. Each analy-

sis will annotate the byte codes with information pertaining to the particular analysis. E.g. the producer-consumer analysis annotates each byte code with a representation of which other byte codes have produced cells on the stack as the stack looks upon entry into this byte code.

Various information from the constant pool of the class file containing the method code is inlined into the byte code, extending and rearranging the byte code. Other changes to the byte code are done as well to make interpretation and compilation easier in the following phases.

After all methods have been annotated, the Java-to-C compiler visits the flow graph one final time to produce the final C code that is the outcome of the compilation.

If a method is marked for interpretation, the byte codes of the method are translated into a C array of `unsigned char` values.

The produced code, together with the interpreter and other utilities implemented in C, are copied by the HVM plugin into a previously specified location and can now be included in the existing build environment for a particular platform.

7 HVM Evaluation

Section 5.3.1 demonstrated how the HVM can be used to add Java software components into an existing C based execution and development platform. Additionally Section 5.3.2 demonstrated the scalability of the HVM to large SCJ applications.

This Section shows measurements comparing the execution efficiency of the HVM to other similar environments. Even though the HVM can be used to program Java for embedded systems it is also very important to engineers that the efficiency by which Java can run is close to the efficiency they are accustomed to for their current C environments.
For high-end embedded platforms results already exists regarding execution speeds of Java programs compared to the same program written in C. In their paper [49] the authors show that their Java-to-C AOT compiler achieves a throughput to within 40% of C code on a high-end embedded platform. This claim is thoroughly substantiated with detailed and elaborate measurements using the CDj and CDc benchmarks[36].

Since the memory requirements of the CDj and CDc benchmarks (see Section 5.3.2) prevents us from running them on low-end embedded systems, this thesis introduces a small range of additional benchmarks. The idea behind these benchmarks are the same as from CDj/CDc: To compare a program written in Java with the same program written in C.

7.1 Method

The 4 benchmark programs are written in both Java and C. The guiding principles of the programs are,

- *Small.* The benchmarks are small. They don't require much ROM nor RAM memory to run. The reason why this principle has been followed is that it increases the probability that they will run on a particular low-end embedded platform

- *Self-contained.* The benchmarks are self-contained, in that they do not require external Java nor C libraries to run. They don't even require the `java.util.*` packages. The reason is that most embedded JVMs offer their own JDKs of varying completeness, and not relying on any particular Java API will increase the chance of the benchmark running out-of-the-box on any given execution environment

- *Non-configurable.* The benchmarks are finished and ready to run as is. There is no need to configure the benchmarks or prepare them for execution on a particular platform. They are ready to run as is. This will make it easier to accurately compare the outcome from running the benchmarks on other platforms, and allow other JVM vendors to compare their results

- *Simple*. The behavior of each benchmark is simple to understand by a quick scrutinizing of the source code. This makes it easier to understand the outcome of running the benchmark and to asses the result.

The benchmark suite of only 4 benchmarks is not complete and the quality and relevance of the suite will grow as new benchmarks are added. The guiding principles of the benchmarks are very important, especially the principle of being self-contained, since this is a principle most important for being successful at running a benchmark on a new embedded platform.

The current benchmarks are:

1. *Quicksort*. The `TestQuicksort` benchmark creates an array of 20 integers initialized with values from 0 to 20 in reverse order. Then a simple implementation of the quicksort method sorts the numbers in place. This benchmark applies recursion and frequent access to arrays

2. *TestTrie*. The `TestTrie` benchmark implements a tree like structure of characters - similar to a hash table - and inserts a small number of words into the structure. This benchmark is focusing on traversing tree like structures by following references

3. *TestDeterminant*. The `TestDeterminant` benchmark models the concept of vectors and matrices using the Java concepts of classes and arrays. Then the Cramer formula for calculating the determinant of a given 3x3 matrix is applied

4. *TestWordReader*. The `TestWordReader` benchmark randomly generates 17 words and inserts them into a sorted list of words, checking the list before each insert to see if it is not there already. Only non duplicates are inserted.

The nature of these benchmarks are not exhausting all aspects of the Java language, but they still reveal interesting information about the efficiency of any given JVM for embedded systems. The purpose of the benchmarks are to reveal how efficiently Java can be executed in terms of clock cycles as compared to C and how much code space and RAM are required. The benchmarks are not intended to test garbage collection, and non of the benchmarks require a functioning GC to run. Nor do they give any information about the real-time behavior of the system under test. To test GC efficiency and/or real-time behavior of a given JVM the CDj/CDc benchmarks are available.

In Section 7.2 compares the results from running these benchmarks on GCC, FijiVM, KESO, HVM, GCJ, JamVM, CACAO and HotSpot. This will give us valuable information about the efficiency with which these environments can execute Java code as compared to each other and as compared to C based execution environments.

7.1.1 Benchmark execution - High-end Platforms

Since only three of the tested execution environments (GCC, KESO and HVM) are capable of running these benchmarks on low-end embedded systems, they were first run on a 32 bit Linux PC. On this platform all JVMs under test could execute the benchmarks. The number of instructions required to run the benchmarks was measured using the **P**erformance **A**pplication **P**rogramming **I**nterface (PAPI) [11, 47]. The reason for measuring the instruction count and not the number of clock cycles is that the instruction count is a deterministic value for the benchmarks, but the clock cycle count is not on advanced processors. This first run of all the benchmarks on a 32 bit Linux PC will not by it self give us the desired results for low-end embedded platforms, but it will allow us to compare the JVMs under test against each other and against C on high-end platforms. To achieve the desired results for low-end embedded platforms the benchmarks will be run on a particular low-end embedded environment as well using GCC, HVM and KESO. This will give the desired results for these two JVMs, but compared with the results for high-end environments one can make statements about what could have been expected had it been possible to run all JVMs on a low-end embedded environment.

For all execution environments the native instruction count was measured by calling the PAPI API before and after each test run. The tests was run several times until the measured value stabilized - this was important for the JIT compilers especially, but also for the other environments. E.g. calling `malloc` for the first time takes more time that calling `malloc` on subsequent runs. All in all the measurements reported are for *hot* runs of the benchmarks.

7.1.2 Benchmark execution - Low-end Platforms

To obtain a result for low-end embedded platforms the benchmarks was run using GCC, HVM and KESO on a ATMega2560 AVR micro-controller. This is an 8 bit micro-controller with 8 kB of RAM and 256 kB ROM. On this simple platform there is a linear, deterministic correspondence between number of instructions executed and clock cycles. The AVR Studio 4 simulator was used to run the benchmarks and accurately measured the clock cycles required to execute each benchmark. Figure 24 shows an example of running the `TestQuicksort` benchmark using GCC. To produce the executable the avr-gcc cross compiler (configured to optimize for size) was used.

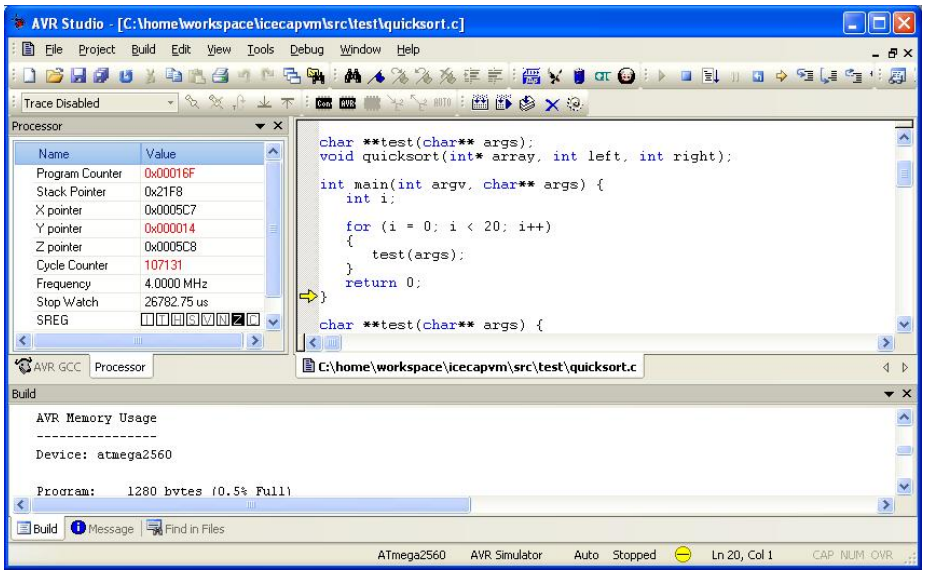


Figure 24: AVR Studio 4

In this test run the clock cycles spent to get to the for-loop was measured (in this case 125 clock cycles), and this number was subtracted from the time taken to perform the benchmark. Then the test was run 20 times, in this case yielding a clock cycle count of 107131. GCC takes (107131 - 125) / 20 = 5350 clock cycles to perform the benchmark.

To obtain similar results for KESO, the C source produced by the KESO Java-to-C compiler was compiled using the avr-gcc cross compiler. An AVR Studio 4 project was created to enable the measurement of clock cycles as above. Again the start up time was measured and each benchmark run a number of times to arrive at an average time taken for KESO to execute the benchmark. Similarly for HVM. All projects configured to optimize for size.

These measurements are directly relevant for low-end embedded platforms and allow us to validate how the HVM compares to GCC and KESO. Since these three environments also appear in the high-end platform measurements, where they can be related to results from the other environments, they offer a chance in Section 7.3 to predict how these other high-end environments would have performed had they been able to run on the ATMega2560.

7.2 Results

The measurements performed using the PAPI API on a 32 bit Linux PC platform are listed in Figure 25 and 26.

The instruction count taken for the C version to execute is defined as 100. The instruction count taken for the other environments is listed relatively to C above. E.g. the HVM uses 36% more instructions to execute the Trie benchmark

	C	KESO	FijiVM	HVM	GCJ
Quicksort	100	101	136	111	172
Trie	100	93	54	136	245
Determinant	100	59	37	96	171
WordReader	100	251	218	177	328
Total	100	126	111	130	229

Figure 25: Instruction Count Comparison - Part 1

	C	JamVM	HVMi	CACAO	HotSpot
Quicksort	100	697	4761	147	156
Trie	100	772	1982	294	234
Determinant	100	544	1664	294	48
WordReader	100	975	4979	263	142
Total	100	747	3346	250	145

Figure 26: Instruction Count Comparison - Part 2

than native C.

The results from comparing HVM and KESO on the ATMega2560 are listed in Figure 27.

	C	KESO	HVM
Quicksort	100	108	130
Trie	100	223	486
Determinant	100	190	408
WordReader	100	331	362
Total	100	213	347

Figure 27: Cycle count comparison

This is an accurate cycle count comparison for KESO and HVM.

7.3 Discussion

The most interesting results are contained in Figure 27. This whows that for the benchmarks tested,*KESO is approximately 2 times slower than C and the HVM is approximately 3 times slower than C.*

There are several observations that should be taken into account when considering the above experiment:

- KESO supports GC, the HVM does not but relies on SCJ memory management. Even though GC is not in effect above, the KESO VM probably

pays a price in terms of execution efficiency for the presence of GC

- The HVM supports Java exceptions, KESO does not. Very rudimentary experiments not shown here indicate that the cost of exception support is an approx 25% decrease in performance for the HVM
- Scrutinizing the C code produced by KESO shows that the Java type `short` is used in places where this is not correct. E.g. code had to be manually fixed for the WordReader benchmark to reintroduce the correct data type `int` in various places. Using `short` where `int` is required might be reasonable in several cases, and this will have a significant impact on performance, especially on 8 bit platforms as the ATMega2560.

The following substantiated observations for low-end embedded platforms can be made,

- Java-to-C compilers are a little slower than native C, but not by an order of magnitude. It is likely that they can be approximately half as fast as native C
- KESO is faster than HVM. The HVM achieves a throughput of approx 50% that of KESO.

8 Summary and Contributions

Apart from this thesis, and the HVM itself as a complete software product, the research performed during the thesis period has been documented in 5 conference papers, 1 journal article and 1 extended abstract. This Section gives a brief introduction to each paper and puts it in context of the HVM. All the papers are included in full in the appendix.

8.1 The Java Legacy Interface - JTRES 2007

In 2007 I used an existing VM (called SimpleRTJ) to support the execution of Java software components on the KT4585 embedded device. In the role of main author of this paper I describe how the SimpleRTJ VM was changed to allow for executing Java tasks alongside existing C tasks. An important outcome of this was a proof-of-concept that it was actually possible to execute Java on the KT4585 while keeping existing legacy software running. The experiment also identified some short comings - most importantly the chosen VM being an interpretation only VM, it was too slow for most practical purposes. Additionally intelligent class linking was not supported thus making it hard to scale to larger Java APIs.

8.2 Hardware Objects for Java - ISORC 2008

Since it was now possible to run Java in and by itself on the KT4585, I quickly recognized the need for being able to interact with hardware in a more elegant manner than through native methods. Based on an idea called *Hardware Objects* by Martin Schoeberl - the main author of this paper - I implemented this feature in the SimpleRTJ and through this experiment contributed to the evaluation of the Hardware Object concept. Hardware Objects later became an important feature of the HVM used in e.g. the SCJ implementation, and in general the preferred way for the HVM to interact with hardware from Java space.

8.3 Interrupt Handlers in Java - ISORC 2008

Another important feature used when programming embedded devices is the ability to handle interrupts as they occur. In the role of main author of this paper I describe how interrupts can be handled directly in Java space, in the context of the actual interrupt and not at some later point. I implemented the idea in the SimpleRTJ VM and demonstrated how Hardware Objects together with 1st level interrupt handling allows for writing device drives in pure Java.

8.4 A Hardware Abstraction Layer in Java - TECS Journal 2011

In the role of co-author a contribution was made to the evaluation section of this journal paper. The paper described how a modern, type-safe programming

language like Java can be used to write devices drivers for embedded systems. The examples of implementing the concept of Hardware Objects and 1st level interrupt handling for the SimpleRTJ, as described above, played a significant role in the evaluation section of this Journal paper. Additionally I implemented the concepts on a new VM (KaffeVM).

At this point in time it was clear that Java can indeed be made to run on low-end embedded devices, can be integrated with existing C code, and can be used to to program hardware. It was also clear that existing VMs - e.g. SimpleRTJ and KaffeVM, was not suitable for devices like the KT4585. The main reasons were lack of efficiency and the monolithic nature of the VMs. At this point in time I decided to start the work with implementing the HVM.

8.5 Flash Memory in Embedded Java Programs - JTRES 2011

The first paper coming out of this effort was about how to support host initialized static data in Java. This concept of static/read-only data placed exclusively in flash, was well known from C environments. In the role of sole author I describe in this paper how read-only data can be handled in a similar manner in Java environments. The HVM was the experimental workbench for the evaluation.

After this paper a burst of implementation work was done on the HVM to increase its performance.

8.6 Towards a Real-Time, WCET Analysable JVM Running in 256 kB of Flash Memory - Nordic Workshop on Programming Theory 2011

In the role of a contributing author this extended abstract describes the changes that had to be made to the HVM in order to make it analyzable for WCET.

8.7 Safety-Critical Java for Low-End Embedded Platforms - JTRES 2012

In the role of 1 of 3 equal contributors this paper presents an implementation of the Safety-Critical Java profile (SCJ), targeted for low-end embedded platforms with as little as 16 kB RAM and 256 kB flash. The implementation is built on top of the HVM. The work utilizes many of the capabilities of the HVM: Hardware objects, 1st level interrupt handling, native variables, and program specialization through intelligent class linking.

The resulting profile implementation and evaluation benchmarks are the until now most clear indication that the ideas behind the HVM scale to complex applications and to real industrial settings.

9 Future Work

The state of the HVM today is such that it can be used as it is in many concrete industrial and educational scenarios. Following this step forward, a selection of obvious research directions present themselves. This section presents those research topics and hint at a future research plan.

9.1 Tool Support

The first and foremost task to take up, is to follow up on the ideas in [8]. In their paper the authors lays out a vision for a complete environment comprised by a set of tools for supporting the development and execution of hard-real time embedded Java. Using the WALA [1] and UPPAAL frameworks the authors have developed tools for *Conformance checking*, *Memory analysis* and *Schedulability analysis*. Using UPPAAL the authors of [23] present the tool *TetaJ* that statically determine the WCET of Java programs executed on the HVM. These tools are today standalone tools, and it would be obvious to embed them into the HVM Eclipse plugin. The HVM plugin today supports the initiation of the Java-to-C translation, but it can be extended to also activate tools for the types of analysis mentioned above. This will bring about a complete Eclipse based toolbox for development of real-time Java programs for embedded systems.

9.1.1 Debugging

One obvious tool that is missing is the ability to interactively debug Java programs. Currently the HVM takes some care to produce fairly readable C code, and it uses names of C data variables that are inherited from Java space. Also the Java source lines are embedded as comments into the generated C code. This makes it possible today to debug the resulting program using existing C based debugging environments. Still, it resembles only being able to debug at the assembler level while programming at the C level. The Eclipse environment defines a standardized debugging protocol, and it would be very useful for programmers to support single-stepping and other standard debugging facilities, at the Java level, of applications executed on the HVM.

9.2 Large Scale Industrial Experiments

The Polycom industrial case presented in the analysis part of this thesis has formed the basis of HVM experiments up until now. Recently, through cooperation with the Danish company Grundfos [32], a new platform has been introduced. Grundfos is looking for input to their decision making about which direction their embedded development methods should take in the future. Grundfos has considered model based development, but are also interested in exploring the use of Java for low-end embedded software development. Currently work is going on with defining a concrete development project where the HVM will

be integrated with their existing C based environment and where Grundfos developers will produce new functionality written in Java and executed alongside existing C based functionality.

Such large scale industrial experiments will bring valuable feedback about howthe HVM and other tools may be improved, and it will allow Grundfos to work with state-of-the-art embedded Java environments to gain the benefits from high-level software development on their low-level devices.

9.3 HVM optimizations

Section 7 shows that the HVM today achieves a throughput of approximately a third of native C. For the work with the Polycom and Grundfos devices this is acceptable, and no present request to improve on this has been made. Still, there are some obvious optimizations still to make in the HVM to improve on efficiency:

- *Static initializers.* When static methods and variables are accessed the Java specification defines when the static initializers of the static class must be executed. Currently this check for if the class has been initialized is done more often than necessary. A static analysis of the Java code will be able to infer at which points the check for static initialization can be avoided, because it will be statically decidable that the check must have been performed previously. This is supported by KESO, but not yet by the HVM.

- *Exception handling.* Handling the throw and catch of exceptions, which is supported in full by the HVM, comes at a significant cost of approximately 25% (see Section 5.2.2). Given an application it could be useful to explore to which extent a static analysis can gain information about which exceptions could be thrown and where. If it can be inferred that a method cannot throw an exception the code generated for the method can be made significantly more efficient.

- *Object field access.* In the HVM access to object fields is done by casting the object reference to an `unsigned char` array and accessing certain parts of that array depending on the type of the field. In KESO, each Java class is translated into a C struct and the object fields are accessed through access to the struct members. In practice this allows GCC and other C compilers to generate more efficient code for accessing object fields. This trick could benefit the HVM as well.

- *Virtual dispatch.* The HVM is very good at recognizing if a virtual call site is not really virtual and turning it into a non virtual method call (which can be implemented more efficiently, see Section 5.2.8). Still some call sites are truly virtual and today they are not handled very efficiently in the HVM. At a truly virtual call site a special utility function gets called for the dispatching of the call to the proper receiver. Instead the compiler

could generate an in-line switch statement that switches on the type of the target object. This will give a larger code size but will most likely be significantly more efficient. This idea resembles a *Polymorphic Inline Cache* and was put forward by [33].

- *Template recognition.* The Java compiler translates Java constructs like `for` and `while` loops into common templates of byte code. If the HVM could recognize these templates, the corresponding `for` and `while` constructs of the C language could be used. Today the HVM generates `goto` statements. It is not obvious that using the corresponding `for` and `while` constructs of the C language instead of `goto` would increase performance, but it will most likely make the resulting C code more readable.

9.4 HVM Optimizations Correctness

The producer-consumer method described in Section 5.2.7 is based on some conjectures about the nature of byte codes generated by Java compilers. Based on these conjectures one can narrow the type of data manipulated by individual byte codes, and thus optimize the code generated to use the most narrow type possible. It seems important to substantiate that the conjectures indeed hold true for all programs, or if they do not, adapt the optimizations to handle failing scenarios. Work by [25] is relevant here.

9.5 Java Level GC

The HVM SCJ implementation, described in [63] and included in the appendix, uses the concepts of Hardware Objects, 1st level interrupt handling and native variables to implement e.g. process scheduling almost entirely in Java - a feature usually implemented in the VM infrastructure. Currently investigations are undergoing to establish if it would be possible to use the same features to implement a garbage collector for the HVM entirely (or almost entirely) in Java. Such a solution would have several advantages. First it would be more portable than a C based solution, and secondly the intelligent class linking facility would exclude the GC if it were not used, thus keeping down the size of executables not utilizing GC. Thirdly it would allow developers to easily implement or adapt GC strategies using the Java language as opposed to changing the VM source code itself.

9.6 Merging HVM with KESO, FijiVM and others

Each of the VMs - KESO, FijiVM, and HVM - have some advantages that the others don't have. E.g. the KESO VM supports the useful concept of isolated domains, and it produces tight and readable C code. The FijiVM produces rather obscure but strikingly efficient C code, and it is the most efficient of the three mentioned VMs. Additionally the FijiVM supports real-time GC, a very relevant feature. The HVM produces C code that is completely self contained

and does not rely on a POSIX like OS or the presence of any other OS or C runtime library, while still producing fairly efficient C code. Also the HVM is integrated with the Eclipse environment and supports program specialization through intelligent class linking.

Ideas from one environment could benefit the others. E.g. could KESO be integrated with Eclipse? Could it generate code not depending on external features? How easy would it be to refactor FijiVM to support intelligent class linking and thus allow it to produce self-contained code, in case the Java source does not use any OS functionality? In other words, does FijiVM have to be monolithic? Does KESO have to rely on OSEK?

Extending each of these VMs with features from each other or carefully selecting the best parts from each to collect them in a new and even better VM would be very interesting.

9.7 HVM in Educational Settings

The HVM has already been used in educational settings and some new ones are under the way. Recently two master thesis projects from AAU were completed, in which the HVM played a significant role (1) WCET analysis of Java programs on common hardware, also documented in [23] and (2) an implementation of STM (**S**oftware **T**ransactional **M**emory) based on the HVM. In the near future it is the intention to use the HVM at VIA University College, when teaching courses in real-time programming and concepts. Previously those courses was taught using the C programming language. Evaluations of the courses revealed that C, being in this setting a new language to the students, was an obstacle in the process of learning the curriculum. The curriculum is not C as such, but rather real-time programming and concepts. Since the students are intimately familiar with Java, the teaching of real-time programming and concepts will be better when using Java instead of C. At AAU, courses are available to industrial partners as well as normal students. One of these courses have the aim of upgrading current C programmers to master Java as well. In this course AAU plans to use the HVM to allow the industrial partners to program exercises and examples in Java on their own embedded hardware.

9.8 HVM for .NET

The C# language is usually translated into a byte code format called the **C**ommon **I**ntermediate **L**anguage (CIL). Could the ideas behind the HVM, especially the program specialization through the intelligent class linking feature, be applied to C# programs as well? The C# language is targeted at server, desktop and in some case high-end embedded systems as well, but it may be possible to run CIL programs on even smaller platforms.

10 Conclusion

This thesis has presented the analysis, design and implementation of the HVM. The HVM adds incrementality and integratability to the set of features supported by Java on low-end embedded platforms.

The work started out of a frustration: it was not possible to find an embedded Java environment that could run on the KT4585 while keeping the existing legacy software running as well. Existing environments were not incremental: adding limited amounts of Java code required a disproportionate amount of RAM and ROM. They were not integratable: they required the inclusion of POSIX-like functionality or the use of particular compilers and libraries, or they produced or contained code that was not directly compilable by the KT4585 build environment. The HVM shows that incrementality can be achieved: the smallest possible Java program executed using the HVM requires approx 8 kB of ROM and a few bytes of RAM. The HVM shows that integratability can be achieved: it produces self-contained, strict ANSI-C code, that can be embedded into existing build environments about which no assumptions are made. In operational terms one can program a new OS Task for the KT4585 in Java and include it in the existing build environment (see Section 5.3). C tasks and Java tasks can coexist, and the KT4585 software developer can choose which parts of an application to write in C and which parts to write in Java.

The HVM supports Hardware Objects and 1st level interrupt handling, and adds the novel feature of native variables. These three concepts make it possible to write device drivers or, in general, to access hardware from Java space, just as the developer will be accustomed to do in C.

It is now possible to start field research with companies such as Polycom and Grundfos and write Java code for industrial devices in operation today. This will in the future give valuable input to the underlying assumption that Java is a better language than C in terms of code quality and speed of development. This claim can not be verified or rejected yet, but now the technical tools to start the experimental work are available.

In the autumn of 2012 the HVM is going to be used in 6th semester courses at VIA University College in Horsens for programming the KT4585 in Java. The feedback from the students will be valuable feedback about the use of the HVM itself.

Apart from the HVM itself, the thesis work have appeared in 5 conference papers, 1 journal article and 1 extended abstract (See Section 8). The HVM has been used in 2 master thesis projects and been referred from several other papers.

The work with the HVM has not finished. Section 9 lays out a number of future tasks, and the HVM can be a significant part of fruitful research at CISS, VIA University College, and elsewhere in the near future.

The introduction stated a problem: that current environments for embedded Java lacked in incrementality and integratability. This thesis has presented techniques to solve these problems, and the HVM demonstrates their feasibility.

11 Acknowledgement

I am profoundly grateful to Hans Søndergaard. He introduced me to embedded real-time Java in the first place, and later encouraged me to start this thesis work. He also introduced me to a very fruitful and inspiring research community at CISS, Aalborg University. I am forever grateful to A.P. Ravn from CISS for teaching me how to write scientific materiel and for always helping me to keep the vision clear and the overall goal in site. I am very grateful to my counsellor Bent Thomsen, who encouraged me to take up my thesis work again after a longer break, and with whom I have had many inspiring conversations. I have used his deep insight into the field of current research in real-time embedded software to position and clarify my work in relation to other efforts. In the first half of my thesis period I was very lucky to be introduced to Martin Schoeberl. Through his ideas and cooperations with him I appeared as coauthor in my first publications. I hope in the future I will be able to work with him again. I also thank Kasper Søe Luckow, Casper Svenning Jensen and Christian Frost for their master thesis work, in which they made the HVM interpreter time predictable and used UPPAAL to calculate WCET boundaries for Java code executed on the HVM [23]. I'm very grateful to Polycom and my former boss, Dion Nielsen, who allowed me to start this thesis work on a part time basis while working at Polycom. Finally, I am very grateful to VIA University College and my current boss Jens Cramer Alkjærsig, who have allowed me to finish the work and given me time and resources to do it.

Appendix

The Java Legacy Interface

Stephan Korsholm
Centre for Embedded Software Systems (CISS)
Aalborg University
DK-9220 Aalborg Ø
stk@cs.aau.dk

Philippe Jean
KIRK telecom
Langmarksvej 34
DK-8700 Horsens
phj@kirktelecom.com

ABSTRACT

The *Java Legacy Interface* is designed to use Java for encapsulating native legacy code on small embedded platforms. We discuss why existing technologies for encapsulating legacy code (JNI) is not sufficient for an important range of small embedded platforms, and we show how the Java Legacy Interface offers this previously missing functionality.

We describe an implementation of the Java Legacy Interface for a particular virtual machine, and how we have used this virtual machine to integrate Java with an existing, commercial, soft real-time, C/C++ legacy platform.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*Scheduling, Threads*; D.4.4 [**Operating Systems**]: Communications Management—*Message sending*; D.4.4 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; D.3.4 [**Processors**]: Run-time environments, Interpreters.

General Terms

Languages, Design.

Keywords

Java/C integration, legacy code, encapsulation, multilanguage interoperability, embedded systems, scheduling, real-time.

1. INTRODUCTION

During recent years a range of real-time capable embedded execution environments for Java have appeared that are both efficient, predictable and complete. The Jamaica Java virtual machine is built around a hard real-time garbage collector, and has become a commercial product supported by aicas [2]. Another commercial product is the PERC implementation of Java supported by Aonix [3]. Both these environments are hard real-time and high-performance. A number of open-source virtual machines for Java (JVM) exist as well, e.g. SableVM [17], JamVM [10], and Jelatine [1].

These implementations are based on accomplishments such as hard real-time garbage collection [19], ahead-of-time compilation, and the emergence of real-time standards and profiles for embedded Java. All together this no longer leaves any doubt that the benefits of the high level, OO language Java can be utilized on the board of an embedded real-time system [4].

For many existing embedded platforms it is not the only requirement, before starting to use Java, that good Java environments are available - in many cases it is also mandatory that *Java can be integrated with the existing legacy platform in order to keep important legacy software running* while allowing future development to take place in Java.

Two ways of integrating Java with legacy code are particularly interesting,

1. Integrate via a Java architecture and call non Java functionality implemented in legacy code.
2. Integrate from the legacy architecture and call Java functionality.

Using the first method the scheduling of the legacy code will be managed by the Java architecture, whereas using the other method the scheduling of the legacy code will not be changed, rather the Java code will be scheduled according to the existing scheduling mechanisms of the legacy platform.

In this paper we are interested in 2), as this method of integration tends to have less impact on the legacy platform. On a certain range of legacy platforms the existing technology JNI (Java Native Interface) [11] can be used to integrate from the legacy architecture and call Java functionality; but on an important range of (typically smaller) embedded platforms, without room for a full JNI implementation, or with scheduling mechanisms other than threading, JNI cannot be used for integrating Java with legacy code.

To meet this need, we have specified the *Java Legacy Interface* (JLI). JLI supports the integration of Java with legacy platforms using method 2) above, and JLI makes it possible to integrate types of platforms where JNI cannot be used for this integration. We suggest a design for JLI and describe an implementation for a particular JVM, and how we have

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.
JTRES '07 September 26-28, 2007 Vienna, Austria
Copyright 2007 ACM 978-59593-813-8/07/9 ...$5.00.

used this JVM to integrate Java with an existing, commercial, soft real-time, C/C++ legacy platform.

The paper is organized as follows: in Section 2 we look at the types of legacy platforms that could benefit from using JLI, and we describe in more detail why JLI is needed. In Section 3 we present our initial design for JLI. In Section 4 we show how we have implemented JLI for a particular legacy system. Finally in sections 5 and 6 we look at what other people have done in this area and look ahead at what are the next steps to take.

2. JAVA ON LEGACY PLATFORMS

The effort required to integrate a JVM with a given legacy platform depends to a large extent on the scheduling mechanism used by the platform. In this section we will look at some important types of scheduling mechanisms: thread based scheduling, event-driven scheduling and task based scheduling.

On thread based scheduling platforms existing technologies (JNI) can most likely be used to integrate Java with legacy code, whereas on the other types of platforms this may not be possible.

2.1 Thread Based Scheduling

Many legacy platforms are build on an OS supporting threading. The legacy code is executed as a number of threads scheduled by the OS kernel. When introducing Java to such a platform, JNI can be used to both start the JVM and to execute Java code. Using the JNI function `JNI_CreateJavaVM`, the JVM can be loaded and initialized. It is now possible for some of the threads to execute legacy code while other threads execute Java code. Using the JNI function `JNI_AttachCurrentThread` native threads can be attached to the JVM and execute functionality implemented in Java. In this scenario the Java code and legacy code are now both scheduled using the existing scheduler of the legacy platform as implemented by the OS. Thus, on legacy platforms built on top of e.g. Linux, Windows CE or VxWorks Java code can be integrated with legacy code in this manner.

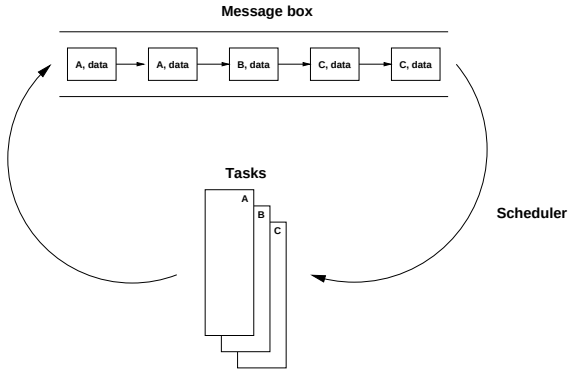
2.2 Event-Driven Scheduling

As observed in [8, 7], event-driven programming is a popular model for writing small embedded systems. In an event-based execution scheme, application code may register interest in certain events by installing event handlers. The main executive, also termed the event loop, waits for events and invokes appropriate handlers. Event handlers cannot be preempted, but run to completion before allowing the main executive to schedule the next handler. For most embedded systems to remain responsive it is important that each event handler is short lived. In many cases it may be considered a fatal error if a handler fails to return before a given maximum timeout. Events may be put on the event queue by e.g. hardware interrupts, but event handlers may also invoke other handlers asynchronously by sending events to each other. Thus on an event-driven legacy platform, code is executed as a continued sequence of handlers being invoked to handle events.

Unfortunately JNI cannot be used for integrating Java with an event-driven execution environment, which we will show in the next section, thus excluding Java from being easily integrated with this important range of embedded legacy platforms.

2.3 Task Based Scheduling

Task based scheduling is a particular instance of event-driven scheduling. A task is a sequence of code that does some computation based on some input. Tasks communicate by sending messages to each other. A message is a piece of data destined for a particular task (the receiver) and the message data will act as input to the receiving task once it gets scheduled. One or more message boxes will hold the message queue(s), and when tasks send messages to other tasks these messages are inserted into the proper message box. This principle is also described in [20]. Figure 1 shows such a system with a single system wide message box.



In this example, C will be scheduled twice, then task B once, and finally task A twice. What happens thereafter depends on the messages sent by tasks A, B, and C.

Figure 1: Task scheduling

When a task gets scheduled it runs to completion. In other words tasks are non-preemptive and cannot be interrupted by other tasks, so in effect all tasks get the same priority. For a task based system to remain responsive it is important that each task invocation only runs for a short while.

On task based scheduling platforms problems occur immediately if trying to use JNI:

- **JVM initialization**.
  Using `JNI_CreateJavaVM` to load and initialize the JVM is not possible because this action does not map to a single task of limited duration. Depending upon the Java code the number of class initializers that should be run may vary, and the entire initialization of the JVM cannot be guaranteed to finish within the maximum duration allowed for each task.
- **Java thread execution**.
  Using `JNI_AttachCurrentThread` to attach Java threads to the legacy scheduler, or indeed executing the JVM itself as a task of limited duration, is not directly possible.

The initialization problem might be solved with an "initialization phase" before entering an "active" phase, but the

scheduling requires much more intensive changes. Yet, if too many changes to the legacy platform are required to solve the above problems, project managers may be reluctant to use Java. To avoid these problems it is tempting to let the JVM take over the responsibilities of the legacy OS. Using this idea, the JVM becomes the OS and is responsible for scheduling both Java code and legacy tasks. Legacy tasks can be activated by means of static native methods, in effect moving the main executive loop from legacy code into Java.

This may be possible in some cases, and it is worth exploring further how many changes are actually required to use the JVM as the main executive responsible for scheduling the execution of legacy code and Java code respectively.

In the following we do not investigate this option further but look at how the problems can be solved on a task based scheduling platform without changing the legacy scheduling mechanism.

3. THE JAVA LEGACY INTERFACE

JLI is designed to support easy integration of Java on embedded platforms where lack of resources or design choices prevent the use of JNI. This may be the case for smaller embedded platforms without room for an OS, or with an OS using other scheduling mechanisms than threading.

Just as JNI works well with larger thread based scheduling platforms, we must ensure that JLI works well with the most common alternatives to thread based scheduling platforms. We have in the initial suggestion for JLI decided to support task based scheduling platforms, but the methods and techniques described are directly applicable to standard event-driven legacy platforms, and we expect, to other types of legacy platforms as well.

For a Java enabled legacy platform to support JLI, changes need to be made in three places:

1. **The JVM**. The JVM needs to support a small set of interface functions. Changes to the JVM should be kept to an absolute minimum, but we have identified some changes that cannot be avoided.
2. **The legacy code**. We do not need to change the scheduling of legacy code, or the legacy code itself, but we add a hook to the legacy scheduler.
3. **Two additional tasks**. The bulk of the JLI functionality is implemented in two new tasks that we have added to the set of legacy tasks.

Points 1. and 2. above need to be done for every JVM and every legacy platform, but the majority of the JLI implementation is in the two new tasks and these are to a large extent generic across task based scheduling platforms.

Figure 2 gives an overview of JLI and affected components. The depicted components provide execution of the JVM in a task based manner, and provide communication between Java and legacy software through message passing. In Figure 2 we see that the JLI functions are implemented in three places: the JVM, the `MessageRouterTask` and the legacy scheduler. E.g. the JLI function `JLI_runVM` is implemented by the JVM and used by the `JVMTask`. In the following we describe in detail these concepts and how they interact.

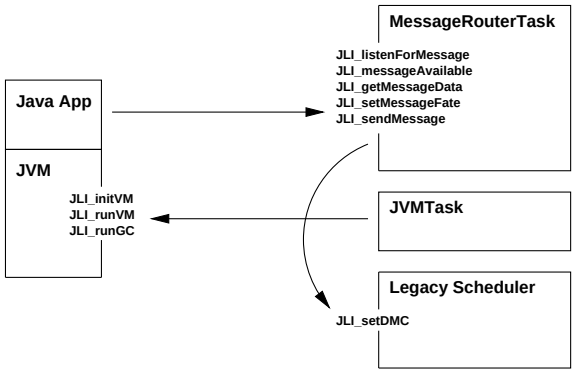


Figure 2: JLI overview

3.1 Java Code Execution

JLI provides an interface for executing the JVM on a task based scheduling platform alongside other executing legacy tasks. In other words JLI must provide a way to ensure that the JVM does not execute for longer than the maximum duration allowed for each task. This means that it must be possible to call the JVM and ask it to execute for a while, but return before the maximum timeout.

3.1.1 Execution principles

The above observations gives rise to the three JLI interface functions relating to JVM execution, see Figure 2. Each of these functions takes a timeout value and returns a status. The timeout specifies for how long the function is allowed to run and the returned status indicates to the caller for how long the function actually ran. The first interface function specified by JLI is

```
int JLI_runVM(int timeout)
```

It will execute byte codes, but suspend and return before the timeout has been reached. Before returning, the JVM must save the entire state of execution and continue using that state the next time `JLI_runVM` is called. If it returns 0 the execution of the Java application has finished.

Before application code starts executing, the application must be initialized. The initialization of the JVM must be done in a task based manner as well. Thus the second interface method relating to code execution is

```
int JLI_initVM(int timeout)
```

This function behaves as above but executes byte codes in the class initializers of the Java application. When it returns 0 the initialization has completed.

Apart from executing byte codes, the JVM is also responsible for doing garbage collection. The exact same requirements as above hold for doing GC - it must be possible to do a bit of GC and return before the maximum task timeout.

The third JLI interface function is

```
int JLI_runGC(int timeout)
```

Note that this does not require the GC implementation to be incremental or real-time. Only that the GC operation can be suspended and resumed later.

3.1.2 Task scheduling

When the JVM supports the JLI interface functions given above, we can create a new task (`JVMTask`) written as a legacy task in C/C++ that handles the proper invocation of `JLI_initVM`, `JLI_runVM` and `JLI_runGC`.

Consider as an example how the the `JVMTask` will execute the JVM:

1. The `JVMTask` will call `JLI_runVM` with a proper timeout that ensures that `JLI_runVM` will return before the maximum task timeout.

2. If `JLI_runVM` returns a value larger than 0, the `JVMTask` will send a message to itself.

3. This message is put in the message box, and eventually the `JVMTask` will get it's own message and may now call `JLI_runVM` to continue running the JVM.

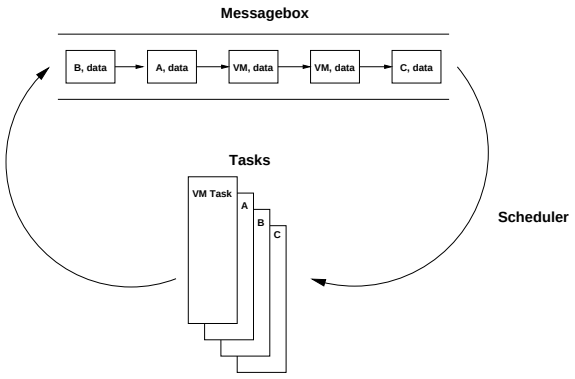


Figure 3: Task based JVM scheduling

Figure 3 illustrates a scenario in which the task C is scheduled first, then the `JVMTask` will run twice, followed by task A once and task B once. The proper invocation of `JLI_initVM` and `JLI_runGC` can be handled in the same manner.

In the resulting integration, the JVM runs as a task in the exact same manner as other existing tasks in the legacy code. This also means that the `JVMTask` will automatically get more or less resources as the system executes. If the legacy tasks are busy sending a lot of messages to each other, the `JVMTask` will automatically get scheduled less.

The legacy code is not affected by this solution. Even though it is required to add a new task (the `JVMTask`) additional to the existing set of legacy tasks, all tasks are still scheduled in the exact same manner as before, and no changes are required in the scheduler.

3.1.3 JVM slicing

The functions `JLI_initVM` and `JLI_runVM` *slice* the execution of the JVM, and we have implemented JVM slicing for an existing JVM not initially supporting this feature.

We also sliced the mark, sweep and compaction phases of the garbage collector. This means that doing a full garbage collection now corresponds to calling `JLI_runGC` several times until it returns 0, indicating that the full GC has completed.

Slicing the JVM and the garbage collector does not in any way give incremental or real-time properties to the Java code being executed by the JVM. Even if the JVM supports a real-time profile, its local scheduling has to be adapted to the slicing in order to satisfy the real-time properties of the profile. We have not considered the implications of this in this work. The slicing simply facilitates the execution of the JVM on a task based scheduling platform.

In Section 4, we look at how and to what extent these functions can be implemented in a non real-time JVM with a simple stop-the-world garbage collector.

3.2 Communication

Looking again at Figure 2, we have in Section 3.1 covered the JVM specific part of JLI. We now move on to look at that part of JLI that is implemented by the two new tasks and the legacy scheduler. This part of JLI provides support for communication between Java code and legacy code.

Legacy tasks communicate by sending messages to each other, so it should be possible to send messages from Java code to legacy tasks and vice versa. To support this we have designed the concept of a *message router*. A message router looks at each message and decides if the message should be routed along the usual channels or routed to Java. If the message is routed to Java, the Java application gets a chance to handle the message instead of the legacy task to which the message was originally intended. Section 4.1 gives a Java code example of how this looks, but here we move on to explain the details of message routing.

3.2.1 Message routing

The message router has a hook into the scheduler that allows the message router to look at each message before it is delivered by the scheduler. To support this the legacy scheduler implements the JLI function,

```
void JLI_setDMC(int (*dmc)(receiver, msg))
```
[1]

and the legacy scheduler must call the supplied callback function (`dmc`) prior to actually delivering the message to the receiver task. If the `dmc` function returns 0 the legacy scheduler must not deliver the message as usual; it must discard it.

The message router implements the `dmc` function to decide if the message should be routed to Java or if it should be routed along the usual channels to the receiving task.

[1]Sets the **D**eliver **M**essage **C**allback.

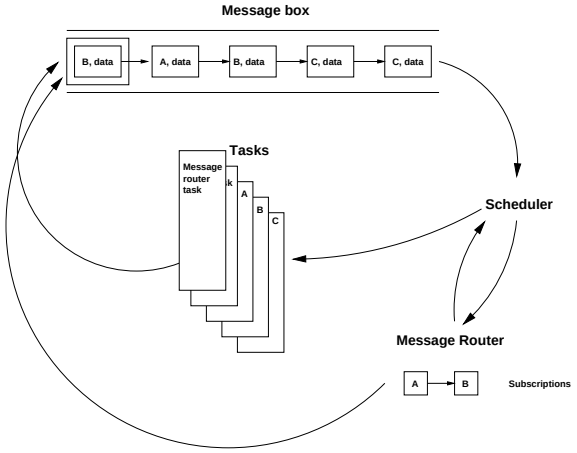
3.2.2 Rerouting messages

The Java program can subscribe to messages for a particular task. To do this Java programs call the native JLI function

```
void JLI_listenForMessage(int receiver)
```

This will insert a task identification into a list of task identifications managed by the message router. The message router compares the receiver of the message to the list of task identifications. If the receiver is not in the list, the message router returns the message to the scheduler and the scheduler will deliver the message to the legacy task as usual. All messages not subscribed to by Java are delivered nearly as fast as usual as only a minimal overhead is required to check each message against the list. If the message should be routed to Java it will take a different path. In this case it will be noticeably delayed compared to the normal case.

When the message is routed to Java it will be packaged into a new message (a message within a message) and sent to a new task we have introduced, called the `MessageRouterTask`, see Figure 2. This is illustrated in Figure 4.



In this illustration the Java application has subscribed to messages for task A and B. Every time a message is sent through the scheduler, the message router will look at the message and compare it to the list of subscriptions. The first two messages to C is not in the list and will be delivered to task C as is. The third message to task B is in the list of subscriptions and will be packaged in an enclosing message and sent to the MessageRouterTask.

Figure 4: Routing messages to the Java application

3.2.3 Handling rerouted messages

When the `MessageRouterTask` receives a rerouted message, it unpacks the message and saves a pointer to the enclosed message in a local variable. To complete the routing, the `MessageRouterTask` will now send a message to itself and return to the scheduler to allow other tasks to run, among those the `JVMTask`. Java code calls the native JLI function

```
int JLI_messageAvailable()
```

to check if a new message has been saved in the local variable. This function returns the length of the message in bytes. A return value of 0 specifies that no message is available to be handled by Java code.

If a message is indeed available, Java code calls another native JLI function

```
int JLI_getMessageData(byte[] data)
```

to get the message data. The message data is returned to Java in a Java byte array. Now the Java application will do two things in sequence,

1. **Message fate**. The Java application looks at the message data and calls a native JLI function

   ```
   void JLI_setMessageFate(boolean fate)
   ```

   If `fate` is 0 the message should be sent along the usual channels to the intended receiver. Otherwise it should be handled exclusively within Java, and the intended receiver of the message should not get the message.

2. **Message handling**. Then a method for handling the message within Java is called.

When the `MessageRouterTask` gets scheduled again, it will check if the Java code has finished the initial analysis of the message and set the message fate. In case Java signals that it wants to handle the message itself, the `MessageRouterTask` will clear the saved message pointer and thus be ready to accept a new message to Java. If Java signals that it does not want the message, the `MessageRouterTask` sends the message on to the originally intended receiver. Note that it cannot send this message through the message box as that would cause the message to pass through the entire history described above once again. Instead the message is delivered immediately by calling the receiving task from within the `MessageRouterTask`.

If the `MessageRouterTask` receives a new message for Java before the previous message for Java has been analyzed and picked up by the Java code, the `MessageRouterTask` will simply resend the message through the message box, thus delaying it until Java has handled the first message.
Figure 5 shows a rerouted message arriving at the `MessageRouterTask`. This is a message that has been subscribed to by the Java application and the message router has previously told the scheduler not to send this to the intended receiver, but have instead packaged the message within a new message and sent it to itself as explained in Section 3.2.2.

A pointer to the message is now saved to hold it until Java picks up the message. The `JVMTask` will eventually be scheduled and the Java code will get the message data through `JLI_getMessageData`. Through the `JLI_setMessageFate` function the Java application will signal if the message to B should be resent to B or just discarded by the message router.

The above implementation has minimal impact on the scheduler, with respect to the amount of code which needs to be

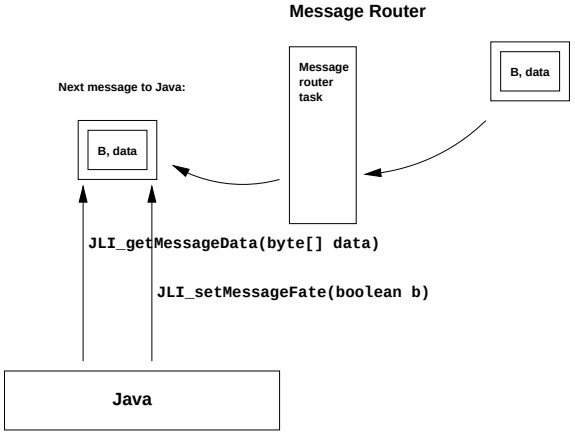


Figure 5: Reading messages from Java

added. Even so we require a hook in the execution loop of the scheduler, and we have had to accept a small delay for each message, even messages not to be handled by Java. Also, we have had to accept a rather large delay for messages that are going to be handled by Java. This is acceptable for some types of messages, e.g. messages updating the user interface, but unacceptable for other types of messages, e.g. messages emptying critical buffers. In general if the application programmer thinks that a message can be handled by Java he must also know that the handling of the message will take longer (since interpreted Java runs slower than compiled C) and it is likely that the delay in delivery of the message is acceptable.

The above facilities for subscribing to messages allow Java code to monitor messages for profiling or debugging purposes without actually handling messages. They also allow Java code to handle all messages to a particular task or only some. The next section describes how JLI has been used to Java-enable an existing C/C++ legacy platform and discusses the possibilities for using Java in the future on the platform.

4. THE KIRK EXAMPLE JLI

KIRK telecom (recently acquired by Polycom) has produced DECT based wireless telephony systems for nearly 15 years. Recently new DECT based, wireless, general purpose control modules have been added to the product portfolio. The generic DECT modules can for example be attached to machinery in a production line or cooling storage in supermarkets, and through a wireless link report collected data about the state of the machinery to a central server (e.g. a PC) controlling the production line or cooling storage.

To allow end users to configure and adapt the behavior of these modules we have embedded a JVM in the modules. The system is fully functional, but a commercial introduction is pending copyright issues. The intention is to allow end users to develop control software for the modules in Java and download their applications to the modules using the wireless DECT link.

The firmware shipped with the modules is written entirely in C as a number of legacy tasks controlling the module and giving it a reasonable default behavior. Through Java the end users can add to this behavior or completely change it by writing new tasks in Java or change the behavior of existing legacy tasks, by subscribing to messages for selected tasks.

In the following we will describe a Java code example illustrating the result of the current integration. Then we will discuss the effort required to implement JLI on the KIRK devices.

4.1 Adapting Task Behavior - A Java Example

The default legacy firmware consists of a number of legacy tasks, among those the `EEPROMTask`. This task handles reading and writing to/from persistent storage on the module. In the following we will look at a Java example program subscribing to messages for the `EEPROMTask`. The main method of the program looks like this:

```
package main;

import kirknative.JTask;
import kirknative.KIRK;

public class Main {

    public static void main(String[] args) {
        JTask eepromTask = new EEPROMTask();
        eepromTask.start();
        try {
            eepromTask.join();
        } catch (InterruptedException e) {
        }
    }
}
```

The `EEPROMTask` class is a subclass of the `JTask` class. The `JTask` class is part of the KIRK API which is a small set of classes interfacing to the JLI implementation on the module. The above program instantiates and starts an `EEPROMTask` object that acts as a Java peer task to the legacy `EEPROMTask`. Through this object, Java code can subscribe to and handle messages to the `EEPROMTask`. The `EEPROMTask` is implemented as follows,

```
public class EEPROMTask extends JTask {
    public EEPROMTask() {
        super("EEPROMTASK");
    }

    public boolean analyzeMessage(Message message) {
        return true;
    }

    public void handleMessage(Message message) {
        KIRK.printString(taskName
                         + ": "
                + message.toString());
    }
}
```

Because of the implementation of the `JTask` class (not shown here) whenever a message is sent to the legacy `EEPROMTask` the method `analyzeMessage` will be called in order to determine the fate value. Since it returns `true`, the Java code signals to the JLI that the message should be sent to the legacy `EEPROMTask` as well. Then the method `handleMessage` is called. For the sake of simplicity, all this method is doing here is to print a trace message.

Inside the implementation of `JTask` and related API classes we find the code actually using the JLI functions. This is basically a Java thread that calls `JLI_messageAvailable` continually. If such a busy wait technique is too time consuming, Java synchronization mechanisms may be used with internal Java threading. When a message is available to the subscribed task the `analyzeMessage` and `handleMessage` methods are called in sequence as illustrated by the following excerpt from the API implementation,

```
public void run() {
 while (!stop) {
  ListenFor current = list;
  while (current != null) {
   int messageSize =
    JLI_messageAvailable(current.getTaskID());
   if (messageSize > 0) {
    byte[] messageData = new byte[messageSize];
    JLI_getMessageData(messageData);
    Message message = Message.getMessage(messageData);

    if (current.
         getListener().
          analyzeMessage(message)) {
     JLI_setMessageFate(true);
    } else {
     JLI_setMessageFate(false);
    }
    current.getListener().handleMessage(message);
   }
   current = current.getNext();
  }
 }
}
```

4.2 JLI Implementation Effort

To add Java support to the KIRK devices we have used the SimpleRTJ JVM from [16]. The reason for choosing this particular implementation is partly because of other people having good experiences with the JVM [15], and partly because early experiments showed that it was very well written, very simple, and very portable.

A desirable side effect of using an existent JVM not initially supporting JLI is that we have gained detailed knowledge about what is required to support JLI on a JVM not specifically programmed with legacy integration in mind.

In the following we describe in more detail how we have prepared the SimpleRTJ for JLI.

4.3 Slicing the JVM

Slicing the JVM consists of dividing the execution of the JVM into smaller units. To do this all recursions and long loops are avoided and replaced either by bounded loops or several limited steps. We analyzed the program dependency graph closely in order to find candidates for slicing points in the source code of the JVM.

After the slicing the previously unbounded loop in the JVM source that handled the continuous execution of the byte codes was changed to allow it to only execute a certain number of byte codes. The maximum timeout of the KIRK legacy tasks is 250 ms, and we made an estimate on how many byte codes we in average could execute during each invocation of `JLI_runVM` in order to return to the scheduler before the maximum timeout. In our future work we will improve on this method by using WCET analysis (see Section 6).

Estimating how many byte codes can be executed to meet the task deadline becomes difficult if the Java application calls legacy code through JNI. In that case an estimate on the WCET of the legacy function being called must be taken into account. In our case we did not need to call through JNI, since we only use the JLI methods for communicating with legacy code as described in 3.2, and we have postponed this problem for our future work.

The technique used was not the best and rather ad-hoc. We believe that the slicing effort can be improved by proper use of existing tools for program slicing (e.g. CodeSurfer [9]).

If building a JVM from scratch it would be possible to incorporate requirements of JLI into the JVM design, thus supporting JLI from the outset. Also the presence of an incremental garbage collector would have made the slicing of the collector more straightforward.

4.4 Scheduling GC

The garbage collector provided by SimpleRTJ is a "stop-the-world" collector which uses simple mark-sweep collection. We sliced the GC using the same techniques as when slicing the byte code execution itself. In the current KIRK implementation of JLI, we check before the execution of each VM slice if a garbage collection is needed. We can check this quickly and decide to call `JLI_runGC` instead if available heap memory becomes too low. In this manner we do not need to run garbage collection in the context of `JLI_runVM`, but rather only in the context of the scheduler.

4.5 Legacy Code Changes

The changes required to the KIRK legacy platform scheduler were very minimal. The effect of adding a hook to the scheduler as required by JLI was not noticeable on the KIRK devices, but in our future work we will make measurements revealing the actual impact of adding this hook.

The implementation of the `JVMTask` and `MessageRouterTask` consists of approx. 350 and 200 lines of code respectively. They are in no way hardware specific, and generally very portable and readily applicable to other legacy platforms of the type described in Section 2.3.

Finally we have developed the KIRK Java API (mentioned in Section 4.1) interfacing to the JLI. This is under constant development and we await user feedback to continue its improvement.

4.6 End User Benefits

Using the KIRK API and the JLI implementation for the KIRK devices we have allowed the Java applications programmer to do things like,

- Listen in on messages to various tasks without changing or stealing the messages. This can be used for monitoring or profiling.
- Grabbing messages to various tasks and handling them exclusively in Java, or handling them both in Java and legacy C.
- Implement entire tasks in Java by grabbing all messages to that task.
- Implement a subset of a task in Java, by grabbing only a subset of messages to that task and routing the remaining messages along the usual channels to the C legacy task.

The JLI implementation with the KIRK API is not only useful for the end user of the KIRK products, but can also be used by KIRK developers to incrementally port selected parts of functionality from C to Java.

5. RELATED WORK

In this section we will look at some alternative approaches to integrate Java with legacy code.

The Java Native Interface (JNI [11]) specifies hooks into the implementation of the JVM that are designed to integrate Java and native C code. If a JVM implements the JNI specification, it allows native C code to load the JVM and execute Java code, and the other way around, it allows Java code to call native C code. JNI also specifies how to attach native threads to the JVM and execute Java code in the context of a native OS thread. If the full specification of JNI is available in the JVM and if the legacy platform supports native threads and semaphores, proper use of JNI should be enough to result in a simple integration of the JVM. But if the platform OS does not support native threads and semaphores, something else is required.

Using an architectural middelware such as CORBA [5] is a standard way of integrating multiple devices in a distributed heterogeneous environment. These techniques have been enabled for embedded environments as well [12]. They are capable of supporting advanced integration (especially distribution). However, it is well known that they come with a huge footprint; thus using such middlewares does not seem like the right tool for the job.

Instead of using a JVM to execute Java we could use ahead-of-time compilation. Using this approach, Java byte code is compiled into native code for the platform and linked with legacy code to obtain the final executable. In this manner no JVM is needed and finding a way to schedule the JVM becomes irrelevant. Using ahead-of-time compilation to execute Java on embedded systems has been done by many, e.g. [18]. A slight variation on this strategy is to compile Java source code (not byte code) into C and then use the C compiler for the platform to compile and link the Java functionality together with legacy C code. This is done by e.g [14] and [21]. The former demonstrates how generated C source can by integrated with a hard real-time garbage collector, thus maintaining one of the most important benefits of Java, namely garbage collection, while still gaining the efficiency of a compiled language. Yet, these approaches do not directly address the scheduling of the Java code, but this is the focus of the Phantom compiler for Java [13]. The Phantom compiler is a serializing compiler that in a manner of speaking compiles the scheduling of the Java tasks into the resulting native code.

In general, interpretation using a JVM and ahead-of-time compilation do not preclude each other, but rather support each other. Each of these methods of execution has its strengths and weaknesses compared to the other. Most importantly, ahead-of-time compilation precludes dynamic class loading, and in our case this feature was particularly important. But if an ahead-of-time compiler is available for the platform, and dynamic class loading is not needed, ahead-of-time compilation might be the best way to get Java onto the platform.

6. CONCLUSION AND FUTURE WORK

In this paper we have advocated that integration with legacy code is important and we have demonstrated that existing technologies (JNI) for integrating Java with legacy code are not sufficient for relevant types of legacy platforms.

We have suggested a design for the *Java Legacy Interface* and shown how JLI can be used to integrate Java with legacy code build on task based scheduling platforms.

We have implemented JLI for the KIRK legacy platform and implemented a Java API for integrating Java with KIRK legacy code.

Using JLI, it is now possible on the KIRK platform to incrementally port selected parts of legacy functionality into Java and to allow end users to configure device behavior through simple Java applications that the users can develop and deploy themselves.

We have identified the following important topics for our future work:

- WCET analysis is required to implement the slicing functions accurately. The JVM must know when the timeout is about to be reached. This can be done if a WCET exists for each byte code being executed. Interpretation of all byte codes are implemented in C and we are currently looking at tools like e.g. OTAWA [6] to do WCET analysis for our particular JVM running on our particular hardware.
- If the JVM supports a real-time profile, slicing the JVM will clearly have an impact on the properties of this profile. We would like to look into the implications of the slicing, and examine to which extent the side-effects of the slicing can be alleviated.
- JLI can be used to integrate a JVM with task based or event-driven platforms, but we would like to make a survey of other types of relevant embedded legacy

platform to examine if JNI can be used and if not, if JLI can be augmented to support the platform.

7. REFERENCES

[1] G. Agosta, S. C. Reghizzi, and G. Svelto. Jelatine: a virtual machine for small embedded systems. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 170–177, New York, NY, USA, 2006. ACM Press.

[2] aicas. http://www.aicas.com/jamaica.html. Visited June 2007.

[3] Aonix. http://www.aonix.com/perc.html. Visited June 2007.

[4] D. F. Bacon, P. Cheng, D. Grove, M. Hind, V. T. Rajan, E. Yahav, M. Hauswirth, C. M. Kirsch, D. Spoonhower, and M. T. Vechev. High-level real-time programming in Java. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 68–78, New York, NY, USA, 2005. ACM Press.

[5] J. Boldt. The common object request broker: Architecture and specification. Specification formal/97-02-25, Object Management Group, July 1995.

[6] H. Casse; and C. Rochange. Otawa, open tool for adaptive wcet analysis. http://www.irit.fr/recherches/ARCHI/MARCH/.

[7] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42, New York, NY, USA, 2006. ACM Press.

[8] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 2003. ACM Press.

[9] GRAMMATECH. http://www.grammatech.com/products/codesurfer/. Visited June 2007.

[10] jamvm. http://jamvm.sourceforge.net/. Visited June 2007.

[11] S. Liang. *The Java Native Interface - Programmers Guide and Specification*. Addison-Wesley, 1999.

[12] S. Malek, C. Seo, and N. Medvidovic. Tailoring an architectural middleware platform to a heterogeneous embedded environment. In *SEM '06: Proceedings of the 6th international workshop on Software engineering and middleware*, pages 63–70, New York, NY, USA, 2006. ACM Press.

[13] A. N. and T. Givargis. Synthesis of time-constrained multitasking embedded software. *ACM Trans. Des. Autom. Electron. Syst.*, 11(4):822–847, 2006.

[14] A. Nilsson, T. Ekman, and K. Nilsson. Real Java for real time - gain and pain. In *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 304–311, New York, NY, USA, 2002. ACM Press.

[15] E. Potratz. A practical comparison between Java and ada in implementing a real-time embedded system. In *SigAda '03: Proceedings of the 2003 annual ACM SIGAda international conference on Ada*, pages 71–83, New York, NY, USA, 2003. ACM Press.

[16] RTJComputing. http://www.rtjcom.com. Visited June 2007.

[17] sablevm. http://www.sablevm.org/. Visited June 2007.

[18] U. P. Schultz, K. Burgaard, F. G. Christensen, and J. L. Knudsen. Compiling Java for low-end embedded systems. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 42–50, New York, NY, USA, 2003. ACM Press.

[19] F. Siebert. *Hard Realtime Garbage Collection - in Modern Object Oriented Programming Languages*. PhD thesis, 2002.

[20] A. S. Tanenbaum. *Operating Systems - Design and Implementation*. Prentice-Hall, 1987.

[21] A. Varma and S. S. Bhattacharyya. Java-through-c compilation: An enabling technology for Java in embedded systems. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 30161, Washington, DC, USA, 2004. IEEE Computer Society.

Hardware Objects for Java

Martin Schoeberl
Institute of Computer Engineering
Vienna University of Technology, Austria
mschoebe@mail.tuwien.ac.at

Christian Thalinger
Institute of Computer Languages
Vienna University of Technology, Austria
twisti@complang.tuwien.ac.at

Stephan Korsholm
Department of Computer Science
Aalborg University DK-9220 Aalborg
stk@cs.aau.dk

Anders P. Ravn
Department of Computer Science
Aalborg University DK-9220 Aalborg
apr@cs.aau.dk

Abstract

Java, as a safe and platform independent language, avoids access to low-level I/O devices or direct memory access. In standard Java, low-level I/O it not a concern; it is handled by the operating system.

However, in the embedded domain resources are scarce and a Java virtual machine (JVM) without an underlying middleware is an attractive architecture. When running the JVM on bare metal*, we need access to I/O devices from Java; therefore we investigate a safe and efficient mechanism to represent I/O devices as first class Java objects, where device registers are represented by object fields. Access to those registers is safe as Java's type system regulates it. The access is also fast as it is directly performed by the bytecodes* getfield *and* putfield*.*

Hardware objects thus provide an object-oriented abstraction of low-level hardware devices. As a proof of concept, we have implemented hardware objects in three quite different JVMs: in the Java processor JOP, the JIT compiler CACAO, and in the interpreting embedded JVM SimpleRTJ.

1 Introduction

In embedded systems Java is now considered an alternative to C/C++. Java improves the safety of programs due to compile time type checking, additional runtime checks, and reference integrity. Those properties result in an increase of programmer productivity. Furthermore, Java is much more portable and thus facilitates reuse.

However, portability and safety comes at a cost: access to low-level devices (common in embedded systems) is not possible from pure Java. One has to use native functions that are implemented in C/C++. Invocation of those native functions incurs runtime overheads. Often they are developed in ad-hoc fashion, thus making them error prone as well; for instance if they interfere with the Java VM or garbage collection when addressing Java objects. Before we present our proposed solution, *Hardware Objects*, we describe the problem as seen from a Java virtual machine (JVM).

1.1 Embedded JVMs

The architecture of JVMs for embedded systems are more diverse than on desktop or server systems. Figure 1 shows variations of Java implementations in embedded systems and an example of the control flow for a web server application. The standard approach of a JVM running on top of an operating system (OS) is shown in sub-figure (a). A network connection bypasses the JVM via native functions and uses the TCP/IP stack and device drivers of the OS.

A JVM without an OS is shown in sub-figure (b). This solution is often called *running on the bare metal*. The JVM acts as the OS and provides thread scheduling and low-level access to the hardware. Thus the stack can be written entirely in Java. An example of the approach is JNode[1] which implements the OS fully in Java.

Sub-figure (c) shows a solution where the JVM is a Java processor. With this solution the native layer can be completely avoided and all code is Java.

However, for both (b) and (c) we need access to device registers and in some applications also interrupts. Here we focus on an object-oriented approach to access device registers which is compatible with Java. The issue of interrupts is treated in a companion paper [7], because it is more related to synchronization and thread scheduling.

1.2 Related Work

An excellent overview of historical solutions to access hardware devices from high-level languages, including C, is presented in Chapter 15.2 of [2]. The solution in Modula-1 (Ch. 15.3.1) is very much like C; however the constructs are safer, because they are encapsulated in modules. In Ada (Ch 15.4.1) the representation of individual fields in registers can be described precisely using *representation* classes, while the

[1] http://www.jnode.org/

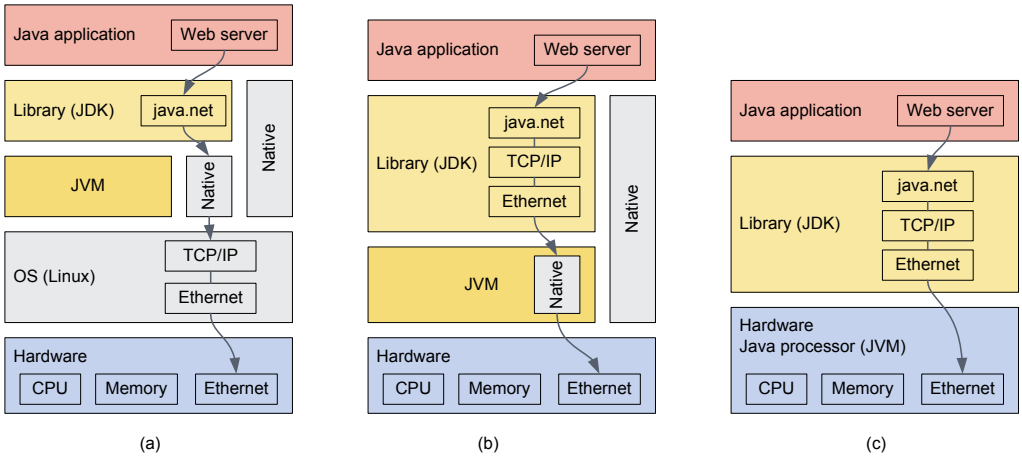


Figure 1. (a) Standard layers for embedded Java with an operating system, (b) a JVM on the bare metal, and (c) a JVM as a Java processor

corresponding structure is bound to a location using the Address attribute.

More recently, the RTSJ [1] does not give much support. Essentially, one has to use RawMemoryAccess at the level of primitive data types. A similar approach is used in the Ravenscar Java profile [8]. Although the solution is efficient, this representation of physical memory is not object oriented and there are some safety issues: When one raw memory area represents an address range where several devices are mapped to there is no protection between them.

The aJile processor [6] uses native functions to access devices through IO pins. The Squawk VM [15], which is a JVM mostly written Java that runs without an operating system, uses device drivers written in Java. These device drivers use a form of peek and poke interface to access the device's memory. The JX Operating System [3] uses *memory objects* to provide read-only memory and device access, which are both required by an OS. Memory objects represent a region of the main address space and accesses to the regions are handled via normal method invocations on the memory objects representing the different regions.

The distinctive feature of our proposal is that it maps a hardware object onto the OO address space and provide, if desired, access methods for individual fields, such that it lifts the facilities of Ada into the object oriented world of Java.

The remainder of the paper is structured as follows: in Section 2 we motivate hardware objects and present the idea. Section 3 provides details on the integration of hardware objects into three different JVMs: a Java processor, a Just-in-time (JIT) compiling JVM, and an interpreting JVM. We conclude and evaluate the proposal in Section 4.

2 Hardware Objects

Let us consider a simple parallel input/output (PIO) device. The PIO provides an interface between I/O registers and I/O pins. The example PIO contains two registers: the *data register* and the *control register*. Writing to the data register stores the value into a register that drives the output pins. Reading from the data register returns the value that is present at the input pins.

```
typedef struct {
    int data;
    int control;
} parallel_port;
#define PORT_ADDRESS 0x10000;

int inval, outval;
parallel_port *mypp;
mypp = (parallel_port *) PORT_ADDRESS;
...
inval = mypp->data;
mypp->data = outval;
```

Figure 2. Definition and usage of a parallel port in C

The control register configures the direction for each PIO pin. When bit n in the control register is set to 1, pin n drives out the value of bit n of the data register. A 0 at bit n in the control register configures pin n as input pin. At reset the port is usually configured as input port[2] – a safe default configuration.

When the I/O address space is memory mapped, such a parallel port is represented in C as a structure and a constant for the address. This definition is part of the board level configuration. Figure 2 shows the parallel port example. The parallel port is directly accessed via a pointer in C. For a system with a distinct I/O address space access to the device registers is performed via distinct machine instructions. Those instructions are represented by C functions that take the address as argument, which is not a type-safe solution.

[2]Output can result in a short circuit between the I/O pin and the external device when the logic levels are different.

```
public final class ParallelPort {
    public volatile int data;
    public volatile int control;
}

int inval, outval;
myport = JVMMagic.getParallelPort();
...
inval = myport.data;
myport.data = outval;
```

Figure 3. The parallel port device as a simple Java class

```
package com.board-vendor.io;

public class IOSystem {

    // do some JVM magic to create the PP object
    private static ParallelPort pp = JVMPPMagic();
    private static SerialPort sp = JVMSPMagic();

    public static ParallelPort getParallelPort() {
        return pp;
    }
    public static SerialPort getSerialPort() {..}
}
```

Figure 4. A Factory with static methods for Singleton hardware objects

This simple representation of memory mapped I/O devices in C is efficient but unsafe. On a standard JVM, native functions, written in C or C++, allow low-level access to devices from Java. This approach is neither safe nor object-oriented (OO) and incurs a lot of overheads; parameters and return values have to be converted between Java and C.

In an OO language the most natural way to represent an I/O device is as an object. Figure 3 shows a class definition and object instantiation for our simple parallel port. The class ParallelPort is equivalent to the structure definition for C in Figure 2. Reference myport points to the hardware object. The device register access is similar to the C version.

The main difference to the C structure is that the access requires no pointers. To provide this convenient representation of I/O devices as objects we just need some *magic* in the JVM and a mechanism to *create* the device object and receive a reference to it. Representing I/O devices as first class objects has following benefits:

Safe: The safety of Java is not compromised. We can access only those device registers that are represented by the class definition.

Efficient: For the most common case of memory mapped I/O device access is through the bytecodes getfield and putfield; for a separate I/O address space the IO-instructions can be included in the JVM as variants of these bytecodes for hardware objects. Both solutions avoid expensive native calls.

```
public class IOFactory {

    private final static int SYS_ADDRESS = ...;
    private final static int SERIAL_ADDRESS = ...;
    private SysDevice sys;
    private SerialPort sp;
    IOFactory() {
        sys = (SysDevice) JVMIOMagic(SYS_ADDRESS);
        sp = (SerialPort) JVMIOMagic(SERIAL_ADDRESS);
    };
    private static IOFactory single = new IOFactory();
    public static IOFactory getFactory() {
        return single;
    }
    public SerialPort getSerialPort() { return sp; }
    public SysDevice getSysDevice() { return sys; }
    // here comes the magic!
    Object JVMIOMagic(int address) {...}
}

public class DspioFactory extends IOFactory {

    private final static int USB_ADDRESS = ...;
    private SerialPort usb;
    DspioFactory() {
        usb = (SerialPort) JVMIOMagic(USB_ADDRESS);
    };
    static DspioFactory single = new DspioFactory();
    public static DspioFactory getDspioFactory() {
        return single;
    }
    public SerialPort getUsbPort() { return usb; }
}
```

Figure 5. A base class of a hardware object Factory and a Factory subclass

2.1 Hardware Object Creation

Representing the registers of each I/O device by an object or an array is clearly a good idea; but how are those objects created? An object that represents an I/O device is a typical Singleton [4]. Only one object should map to a single device. Therefore, hardware objects cannot be instantiated by a simple new: (1) they have to be mapped by some JVM magic to the device registers; (2) each device is represented by a single object.

One may assume that the board manufacturer provides the classes for the hardware objects and the configuration class for the board. This configuration class provides the Factory [4] methods (a common design pattern to create Singletons) to instantiate hardware objects.

Each I/O device object is created by its own Factory method. The collection of those methods is the board configuration which itself is also a Singleton (we have only one board). The configuration Singleton property is enforced by a class that contains only static methods. Figure 4 shows an example for such a class. The class IOSystem represents a minimal system with two devices: a parallel port as discussed before to interact with the environment and a serial port for program download and debugging.

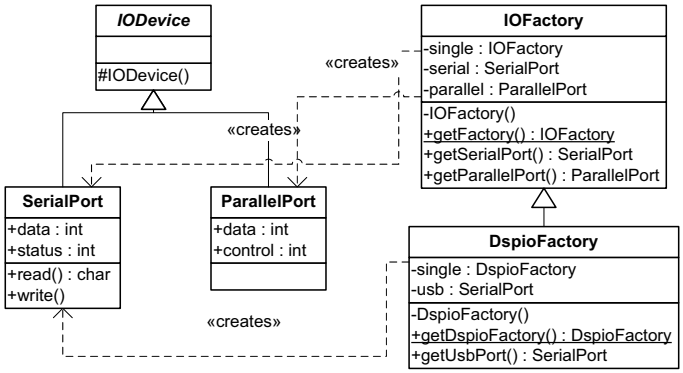


Figure 6. Hardware object classes and board Factory classes

This approach is simple, but not very flexible. Consider a vendor who provides boards in slightly different configurations (e.g., with different number of serial ports). With the approach described above each board requires a different IOSystem class that lists all devices.

2.2 Board Configurations

We can avoid the duplication of code by replacing the static Factory methods by instance methods and use inheritance for different configurations. With a Factory object we represent the common subset of I/O devices by a base class and the variants as subclasses.

However, the Factory object itself shall still be a Singleton. Therefore the board specific Factory object is created at class initialization and can be retrieved by a static method. Figure 5 shows an example of a base Factory and a derived Factory. Note how getFactory() is used to get a single instance of the hardware object Factory. We have applied the idea of a Factory two times: the first Factory generates an object that represents the board configuration. That object is itself a Factory that generates the objects that represent the actual devices – the hardware objects.

The shown example Factory is a simplified version of the minimum configuration of the JOP [11] FPGA module *Cycore* and an extension with an I/O board that contains an USB interface.

Furthermore, we show in Figure 5 a different way to incorporate the JVM *magic* into the Factory: we define well known constants (the memory addresses of the devices) in the Factory and let the native function JVMIOMagic() return the correct I/O device type.

Figure 6 gives a summary example (a slight variation of the former example) of hardware object classes and the corresponding Factory classes as an UML class diagram. The serial port hardware object contains additional access methods to the device register fields. The figure shows that all I/O classes subclass the abstract class IODevice, a detail we have omitted in our discussion so far.

```
public class Example {

    public static void main(String[] args) {

        IOFactory fact = IOFactory.getFactory();
        SerialPort sp = fact.getSerialPort();

        String hello = "Hello World!";

        for (int i=0; i<hello.length(); ++i) {
            // busy wait on transmit data register empty
            while ((sp.status & SerialPort.MASK_TDRE)==0)
                ;
            // write a character
            sp.data = hello.charAt(i);
        }
    }
}
```

Figure 7. Hello World using hardware objects

2.3 Using Hardware Objects

Creation of hardware objects is a bit complex, but usage is very simple. After obtaining a reference to the object all what has to be done (or can be done) is to read from and write to the object fields. Figure 7 shows an example of the client code. The example is the *Hello World* program using low-level access to the terminal via a hardware object.

3 Implementations

In order to show that our proposed approach is workable we have chosen three completely different JVMs for the evaluation: a Java processor (JOP [11, 13]), a JIT JVM (CACAO [5]) and a small interpreting JVM (the SimpleRTJ VM [10]). All three projects are open-source and make it possible for us to show that hardware objects can be implemented in very different JVMs.

We provide implementation details to help other JVM developers to add hardware objects to their JVM. The techniques used for JOP, CACAO, or SimpleRTJ cannot be used one-to-one. However, the solutions (or sometimes a workaround) presented here should guide other JVM developers.

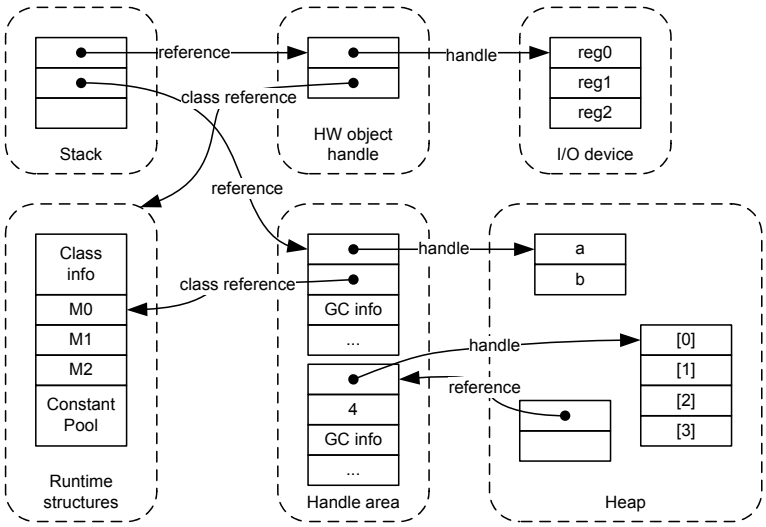


Figure 8. Memory layout of the JOP JVM

3.1 HW Objects on JOP

We have implemented the proposed hardware objects in the JVM for the Java processor JOP [11, 13]. No changes inside the JVM (the microcode in JOP) were necessary. The tricky part is the creation of hardware objects (the Factory classes).

3.1.1 Object Layout

In JOP objects and arrays are referenced through an indirection, called the *handle*. This indirection is a lightweight read barrier for the compacting real-time garbage collector (GC) [12, 14]. All handles for objects in the heap are located in a distinct memory region, the handle area. Besides the indirection to the *real* object the handle contains auxiliary data, such as a reference to the class information, the array length, and GC related data. Figure 8 shows an example with a small object that contains two fields and an integer array of length 4. We can see that the object and the array on the heap just contain the data and no additional hidden fields. This object layout greatly simplifies our object to I/O device mapping. We just need a handle where the indirection points to the memory mapped device registers. This configuration is shown in the upper part of Figure 8. Note that we do not need the GC information for the HW object handles.

3.1.2 The Hardware Object Factory

As described in Section 2.1 we do not allow applications to create hardware objects; the constructor is private. Two static fields are used to store the handle to the hardware object. The first field is initialized with the base address of the I/O device; the second field contains a pointer to the class information. The address of the first static field is returned as the reference to the serial port object. We have to solve two issues: (1) obtain the class reference for the HW object; (2) return the address of a static field as a reference to the hardware object.

We have two options to get a pointer to the class information of a hardware object, such as SerialPort, in a method of IOFactory:

1. Create a normal instance of SerialPort with new on the heap and copy the pointer to the class information.

2. Invoke a static method of SerialPort. The method executes in the context of the class SerialPort and has access to the constant pool of that class and the rest of the class information.

Option 1 is simple and results in following code for the object factory:

```
SerialPort s = new SerialPort();
int ref = Native.toInt(s);
SP_MTAB = Native.rdMem(ref+1);
```

All methods in class Native, a JOP system class, are *native*[3] methods for low-level functions – the code we want to avoid in application code. Method toInt(Object o) defeats Java's type safety and returns a reference as an int. Method rdMem(int addr) performs a memory read. In our case the second word from the handle, the pointer to the class information. The main drawback of option 1 is the creation of normal instances of the hardware class. With option 1 the visibility of the constructor has to be relaxed to package.

For option 2 we have to extend each hardware object by a class method to retrieve the address of the class information. Figure 9 shows the version of SerialPort with this method. We use again native functions to access JVM internal information. In this case rdIntMem(1) loads one word from the

[3]There are no native functions in JOP – bytecode is the native instruction set. The very few native functions in class Native are replaced by a special, unused bytecode during class linking.

```
public final class SerialPort {

    public volatile int status;
    public volatile int data;

    static int getClassRef() {
        // we can access the constant pool pointer
        // and therefore get the class reference
        int cp = Native.rdIntMem(1);
        ...
        return ref;
    }
}
```

Figure 9. A static method to retrieve the address of the class information

on-chip memory onto the top-of-stack. The on-chip memory contains the stack cache and some JVM internal registers. At address 1 the pointer to the constant pool of the actual class is located. From that address we can calculate the address of the class information. The main drawback of option 2 is the repetitive copy of getClassRef() in each hardware class. As this method has to be static (we need it before we have an actual instance of the class) we cannot move it to a common superclass.

We decided to use option 1 to avoid the code duplication. The resulting package visibility of the hardware object constructor is a minor issue.

All I/O device classes and the Factory classes are grouped into a single package, in our case in com.jopdesign.io. To avoid exposing the native functions (class Native) that reside in a system package we use delegation. The Factory constructor delegates all low-level work to a helper method from the system package.

3.2 HW Objects in CACAO

As a second experiment we have implemented the hardware objects in the CACAO VM [5]. The CACAO VM is a research JVM developed at the Vienna University of Technology and has a Just-In-Time (JIT) compiler for various architectures.

3.2.1 Object layout

As most other JVMs, CACAO's Java object layout includes an object header which is part of the object itself and resides on the garbage collected heap (GC heap). This fact makes the idea of having a *real* hardware-object impossible without changing the CACAO VM radically. Thus we have to use an indirection for accessing hardware-fields and hardware-arrays. Having an indirection adds obviously an overhead for accesses to hardware-fields or hardware-arrays. On the other hand, CACAO does widening of primitive fields of the type boolean, byte, char, and short to int which would make it impossible to access hardware-fields smaller than int directly in a Java object. With indirection we can solve this issue. We store the address of the hardware-field in a Java object field and access the correct data size in JIT code.

When it comes to storing the hardware address in a Java object field, we hit another problem. CACAO supports 32 and 64-bit architectures and obviously a hardware address of a byte-field on a 64-bit architecture won't fit into a widened 32-bit object field. To get around this problem we widen all object fields of sub-classes of org.cacaovm.io.IODevice to the pointer size on 64-bit machines. To be able to widen these fields and to generate the correct code later on in the JIT compiler, we add a VM internal flag ACC_CLASS_HARDWARE_FIELDS and set it for the class org.cacaovm.io.IODevice and all its subclasses, so the JIT compiler can generate the correct code without the need to do super-class tests during the JIT compiler run. For hardware-arrays we have to implement a similar approach. The object layout of an array in CACAO looks like this:

```
typedef struct java_array_t {
    java_object_t objheader;
    int32_t       size;
} java_array_t;

typedef struct java_intarray_t {
    java_array_t header;
    int32_t      data[1];
} java_intarray_t;
```

The data field of the array structure is expanded to the actual size when the array object is allocated on the Java heap. This is a common practice in C.

When we want to access a hardware array we have the same problem as for fields – the array header. We cannot put the array directly on the hardware addresses. Therefore we add a union to the java_xxxarray_t-structures:

```
typedef struct java_intarray_t {
    java_array_t header;
    union {
        int32_t  array[1];
        intptr_t address;
    } data;
} java_intarray_t;
```

Now we can allocate the required memory for Java arrays or store the hardware address for hardware arrays into the array object.

3.2.2 Implementation

CACAO's JIT compiler generates widened loads and stores for getfield and putfield instructions. But when we want to load byte or short fields from a hardware object we need to generate 8-bit or 16-bit loads and stores, respectively. To get these instructions generated we implement additional cases in the JIT compiler for the various primitive types.

Whether the JIT compiler needs to generate 8-bit or 16-bit loads and stores for boolean, byte, char, or short fields is decided on the flags of the declared class.

Contrary to hardware fields, when accessing hardware arrays we have to generate some dedicated code for array accesses to distinguish between Java arrays and hardware arrays at runtime and generate two different code paths, one to access Java arrays and the other to access hardware arrays.

3.3 HW Objects in SimpleRTJ

In a third experiment we have implemented hardware objects for the SimpleRTJ interpreter [10]. The SimpleRTJ VM is described in more detail in [9]. To support the direct reading and writing from/to raw memory we introduced an additional version of the put/get-field bytecodes. We changed the VM locally to use these versions at bytecode addresses where access to hardware objects is performed. The original versions of put/get-field are not changed and are still used to access normal Java object fields.

The new versions of put/get-field to handle hardware objects are different. An object is identified as a hardware object if it inherits from the base class IODevice. This base class defines one 32 bit integer field called address. During initialization of the hardware object the address field variable is set to the absolute address of the device register range that this hardware object accesses.

The hardware object specific versions of put/get-field calculates the offset of the field being accessed as a sum of the width of all fields preceding it. In the following example control has an offset of 0, data an offset of 1, status an offset of 3 and finally reset an offset of 7.

```
class DummyDevice extends IODevice {
   public byte control;
   public short data;
   public int status;
   public int reset;
}
```

The field offset is added to the base address as stored in the super class instance variable address to get the absolute address of the device register or raw memory to access. The width (or number of bytes) of the data to access is derived from the type of the field.

To ensure that the speed by which normal objects are accessed do not suffer from the presence of hardware objects we use the following strategy: The first time a put/get-field bytecode is executed a check is made if the objects accessed is a hardware object. If so, the bytecode is substituted with the hardware object specific versions of put/get-field. If not the bytecode is substituted with the normal versions of put/get-field.

For this to be sound, a specific put/get-field instruction is never allowed to access both normal and hardware objects. In a polymorphic language like Java this is in general not a sound assumption. However, with the inheritance hierarchy of hardware object types this is a safe assumption.

3.4 Summary

We have described the implementation of hardware objects on JOP in great detail and outlined the implementation in CACAO and in SimpleRTJ. Other JVMs use different structures for their class and object representations and the presented solutions cannot be applied directly. However, the provided details give guidelines for changing other JVMs to implement hardware objects.

On JOP all the code could be written in Java,[4] it was not necessary to change the microcode (the low-level implementation of the JVM bytecodes in JOP). Only a single change in the runtime representation of classes proved necessary. The implementation in CACAO was straightforward. Adding a new internal flag to flag classes which contain hardware-fields and generating slightly more code for array accesses, was enough to get hardware objects working in CACAO.

4 Conclusion

We have introduced the notation of hardware objects. They provide an object-oriented abstraction of low-level devices. They are first class objects providing safe and efficient access to device registers from Java.

To show that the concept is practical we have implemented it in three different JVMs: in the Java processor JOP, in the research VM CACAO, and in the embedded JVM SimpleRTJ. The implementation on JOP was surprisingly simple – the coding took about a single day. The changes in the JIT JVM and in the interpreter JVM have been slightly more complex.

The proposed hardware objects are an important step for embedded Java systems without a middleware layer. Device drivers can be efficiently programmed in Java and benefit from the same safety aspects as Java application code.

4.1 Performance

Our main objective for hardware objects is a clean OO interface to I/O devices. Performance of the access of device registers is an important secondary goal, because short access time is important on relatively slow embedded processors while it matters less on general purpose processors, where the slow I/O bus essentially limits the access time. In Table 1 we compare the access time to a device register with native functions to the access via hardware objects.

On JOP the native access is faster than using hardware objects. This is due to the fact that a native access is a special bytecode and not a function call. The special bytecode accesses memory directly, where the bytecodes putfield and getfield perform a null pointer check and indirection through the handle.

The performance evaluation with the CACAO JVM has been performed on a 2 GHz x86_64 machine under Linux

[4] except the already available primitive native functions

	JOP		CACAO		SimpleRTJ	
	read	write	read	write	read	write
native	8	9	24004	23683	2588	1123
HWO	21	24	22630	21668	3956	3418

Table 1. Access time to a device register in clock cycles

with reads and writes to the serial port. The access via hardware objects is slightly faster (6% for read and 9% for write, respectively). The kernel trap and the access time on the I/O bus dominate the cost of the access in both versions. On an experiment with shared memory instead of a real I/O device the cost of the native function call was considerable.

On the SimpleRTJ VM the native access is slightly faster than access to hardware objects. The reason is that the SimpleRTJ VM does not implement JNI, but has it's own proprietary, more efficient, way to invoke native methods. It does this very efficiently using a pre-linking phase where the invokestatic bytecode is instrumented with information to allow an immediate invocation of the target native function. On the other hand, hardware object field access needs a field lookup that is more time consuming than invoking a static method.

4.2 Safety and Portability Aspects

Hardware objects map object fields to the device registers. When the class that represents an I/O device is correct, access to the low-level device is safe – it is not possible to read from or write to an arbitrary memory address. A memory area represented by an array is protected by Java's array bounds check.

It is obvious that hardware objects are platform dependent, after all the idea is to have an interface to the bare metal. Nevertheless, hardware objects give device manufacturers an opportunity to supply supporting software that fits into Java's object-oriented framework and thus cater for developers of embedded software.

4.3 Interrupts

Hardware objects are a vehicle to write device drivers in Java and benefit from the safe language. However, most device drivers also need to handle interrupts. We have not covered the topic of writing interrupt handlers in Java. This topic is covered by a companion paper [7], where we discuss interrupt handlers implemented in Java. Jointly Java hardware objects and interrupt handlers makes it attractive to develop platform dependent middleware fully within an object-oriented framework with excellent structuring facilities and fine grained control over the unavoidable unsafe facilities.

Acknowledgement

We thank the anonymous reviewers for their detailed and insightfully comments that helped to improve the paper.

References

[1] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[2] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. Addison-Wesley Longman Publishing Co., Inc., 3rd edition, 2001.

[3] M. Felser, M. Golm, C. Wawersich, and J. Kleinöder. The JX operating system. In *Proceedings of the USENIX Annual Technical Conference*, pages 45–58, 2002.

[4] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, 1994.

[5] R. Grafl. CACAO: A 64-Bit JavaVM Just-in-Time Compiler. Master's thesis, Vienna University of Technology, 1997.

[6] D. S. Hardin. Real-time objects on the bare metal: An efficient hardware realization of the Java virtual machine. In *Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*, page 53. IEEE Computer Society, 2001.

[7] S. Korsholm, M. Schoeberl, and A. P. Ravn. Java interrupt handling. In *Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008)*, Orlando, Florida, USA, May 2008. IEEE Computer Society.

[8] J. Kwon, A. Wellings, and S. King. Ravenscar-Java: A high integrity profile for real-time Java. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 131–140. ACM Press, 2002.

[9] E. Potratz. A practical comparison between Java and Ada in implementing a real-time embedded system. In *SigAda '03: Proceedings of the 2003 annual ACM SIGAda international conference on Ada*, pages 71–83. ACM Press, 2003.

[10] RTJComputing. http://www.rtjcom.com. Visited June 2007.

[11] M. Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[12] M. Schoeberl. Real-time garbage collection for Java. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*, pages 424–432, Gyeongju, Korea, April 2006.

[13] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Article in press and online: Journal of Systems Architecture*, doi:10.1016/j.sysarc.2007.06.001, 2007.

[14] M. Schoeberl and J. Vitek. Garbage collection for safety critical Java. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 85–93, Vienna, Austria, September 2007. ACM Press.

[15] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java on the bare metal of wireless sensor devices: the squawk java virtual machine. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 78–88. ACM Press, 2006.

Interrupt Handlers in Java

Stephan Korsholm
Department of Computer Science
Aalborg University DK-9220 Aalborg
stk@cs.aau.dk

Martin Schoeberl
Institute of Computer Engineering
Vienna University of Technology, Austria
mschoebe@mail.tuwien.ac.at

Anders P. Ravn
Department of Computer Science
Aalborg University DK-9220 Aalborg
apr@cs.aau.dk

Abstract

An important part of implementing device drivers is to control the interrupt facilities of the hardware platform and to program interrupt handlers. Current methods for handling interrupts in Java use a server thread waiting for the VM to signal an interrupt occurrence. It means that the interrupt is handled at a later time, which has some disadvantages. We present constructs that allow interrupts to be handled directly and not at a later point decided by a scheduler. A desirable feature of our approach is that we do not require a native middelware layer but can handle interrupts entirely with Java code. We have implemented our approach using an interpreter and a Java processor, and give an example demonstrating its use.

1 Introduction

In an embedded system which is implemented as a collection of cooperating threads, interrupt handlers are more similar to subroutines than to threads. The handlers are not scheduled as threads; they may be invoked any time and even interrupt the thread scheduler, because interrupts are dispatched by the hardware as response to external events.

Current profiles for real-time Java [5, 2] attempt to hand over interrupts to normal sporadic threads as quickly as possible. This can be done by installing a piece of native code, called the first level interrupt handler, to be invoked by the hardware, and let this code register the interrupt occurrence and then return. Then at each reschedule point the interrupt mark is checked and a waiting thread is unblocked to handle the interrupt (this is sometimes called the second level interrupt handler). Examples of this approach are found in e.g. the Squawk VM [11] and the JamaicaVM from aicas [1]. Squawk reports an average latency of 0.1 milliseconds and a worst case latency of around 13 milliseconds. From [4] we see that the JamaicaVM has an average latency of 50 microseconds and a worst case latency of 250 microseconds. In both cases the interrupts are handled out-of-context.

An advantage of this approach is that the execution of the second level interrupt handler is controlled by the normal thread scheduler and thus observes the priorities and scheduling principles of it. Less desirable features are:

Latency: A delay is introduced from the occurrence of the interrupt until the point in time when it is handled.

Out-of-context execution: The interrupt is handled out of the context of the first level interrupt handler. This means that any computation that requires this context cannot be done in Java, but must be done in native code. If the context dependent part can be kept stereotypical, this is less of a problem. In other cases where device dependent actions are needed, the native middelware layer becomes complex.

The contribution of this paper consists in the design and implementation of a mechanism for handling interrupts in Java. It does not hand over the interrupt to a sporadic thread, but handles the interrupt completely in the context of the first level interrupt handler. We call this mechanism *in-context interrupt handling*. An important feature of in-context interrupt handling is that actions that need to be done in the context of the first level interrupt handler can now be done in Java, in effect eliminating the need for a native middelware layer. This mechanism does not preclude the standard way of handling interrupts in real-time Java, but complements it and can be used in cases where the standard method is inadequate for one of the reasons given above.

In Section 2 we will describe how interrupts are handled in legacy systems. In Section 3 we introduce our design for in-context interrupt handling, and discuss how the method can be supported in existing Java execution environments. In Section 4 we demonstrate how the design has been implemented for two different execution environments for Java: the JOP Java processor and the SimpleRTJ interpreter. Then in Section 5 we show a simple example of using our interrupt handling implementations. We conclude the paper in Section 6.

```
volatile uint16 P0_UART_RX_TX_REG @ 0xFFE032;
volatile uint16 P0_CLEAR_RX_INT_REG @ 0xFFE036;
volatile uint16 RESET_INT_PENDING_REG @ 0xFFE202;

#define CLR_UART_RX_INT_PENDING 0x0010
#define CLEAR_UART_RI_FLAG P0_CLEAR_RX_INT_REG = 0
#define CLEAR_PENDING_UART_RI_INTERRUPT \
    RESET_INT_PENDING_REG = CLR_UART_RX_INT_PENDING

__interrupt void Uart_RX_Interrupt(void) {
   UartRxBuffer[UartRxWrPtr++] = P0_UART_RX_TX_REG;
   if (UartRxWrPtr>=sizeof(UartRxBuffer)) {
      UartRxWrPtr=0;
   }
   CLEAR_UART_RI_FLAG;
   CLEAR_PENDING_UART_RI_INTERRUPT;
}
```

Figure 1. An example interrupt handler in C

2 Conventional Interrupt Handling

Interrupts are used to signal external events for example, detecting that a button has been pressed. When an interrupt occurs the processor simply stops executing the code it runs, and jumps to an interrupt routine instead. The jump saves the environment of the interrupted process so that it may be restored later; this includes saving the CPU registers and the processor status register. This makes it possible to continue the execution of the original code when the interrupt routine has been executed.

Saving the interrupted context and setting up a new context for the interrupt handler is called a *context switch*. Some hardware platforms implement a *full* context switch in hardware, where other platforms implements a *partial* context switch in hardware. In the latter case the programmer must save those parts of the interrupted context that he needs to overwrite in the interrupt handler.

As an example interrupt handler, Figure 1 shows an excerpt of code implementing the RS232 receive interrupt for an existing legacy system. The RS232 receive interrupt is generated by the hardware when the RS232 interface receives a byte on the serial line. It is the job of the interrupt handler to retrieve the byte from the proper device register and clear the receive interrupt flag.

Though this example contains non-standard compiler directives and runs on a particular piece of hardware, it illustrates the following general features:

I/O Memory: Through the compiler directive name @ address the name, e.g. P0_UART_RX_TX_REG is designated to refer directly to a physical memory location, in this case one of the memory mapped device registers of the UART. Any assignment or query of these names in the code will correspond to reads and writes of the particular register.

Interrupt handlers: Through the compiler directive __interrupt the function void Uart_RX_Interrupt(void) becomes an interrupt routine. This basically means that exit and entry code is generated by the compiler to save and restore the state of the interrupted process.

The circular buffer UartRxBuffer can be read by user code outside the context of the interrupt to handle the bytes received. Some kind of mutual exclusion between user code and the interrupt handler may be required. This is typically implemented by disabling interrupts.

3 In-context Interrupt Handling

Our goal is to be able to implement the interrupt handler from Figure 1 in pure Java. The two important tasks that the Uart_RX_Interrupt handler must do are:

1. Retrieve the received byte from the proper device register and save the byte in a data structure.
2. Clean up from the interrupt, in this case clear the UART receive interrupt flag (CLEAR_UART_RI_FLAG; CLEAR_PENDING_UART_RI_INTERRUPT).

Using the RTSJ profile [2] for Java the above tasks can naturally be solved in the following manner:

Ad 1) Using the raw memory access supplied by RTSJ through the class RawMemoryAccess, it is possible to read the received byte from the proper device register. A detailed example on how this looks is available in [3] (Chapter 15.5).

Ad 2) The suggested way to handle interrupts in RTSJ is to use the AsyncEvent and AsyncEventHandler classes. Again [3] includes a detailed example. In the RTSJ an interrupt occurring is equivalent to the fire method being called on the AsyncEvent object. This in turn will call the run() method on all installed handlers. But, to handle the interrupt from Figure 1 in pure Java, the fire method *must be called in the context of the interrupt*. The reason is that the receive interrupt flag must be cleared before exiting the interrupt context. Failing to do so will cause the interrupt to recur. In all implementations of RTSJ that we know of, handling of the AsyncEvent corresponding to the interrupt will be scheduled outside the context of the interrupt. This does not allow us to implement the handler from Figure 1 in pure Java.

As a complementary way to handle interrupts in Java we suggest that *the native code implementing the first level interrupt handler is used to call the JVM and start executing the appropriate interrupt handler immediately, or in other words, before returning from the interrupt*. It makes it possible to handle the interrupt completely in its context. Thus, we will be able to implement the example in Figure 1 in pure Java, which includes to clear the interrupt receive flag from inside Java code.

Whether it is possible to reenter the JVM inside the context of the first level interrupt handler in order to execute the Java part of the interrupt handler depends on the scheduling mechanism and the GC strategy. In the remainder of this

section we will look at the consequences of the proposal for different types of Java execution environments.

3.1 Scheduling

If native threads are used and attached to the VM e.g through the JNI [6] function JNI_AttachCurrentThread it should be straightforward to reenter the JVM while it is interrupted, because from the point of view of the JVM the interrupt handler is not different from a normal high priority thread that has been switched in by the external scheduler.

If an internal scheduler is used (also called *green threads*) it will most likely require some work to refactor the JVM implementation to support reentry at any time. The reason is that the JVM implementation knows when thread switching can occur and explicitly or implicitly has used this knowledge when accessing global data. The SimpleRTJ VM [7], used for one of the experiments described in Section 4, includes an internal scheduler and the section shows the work required to make the JVM reenterable.

3.2 Garbage Collection

When executing the Java first level interrupt handler[1] in the context of the interrupt, it becomes very important that the handler is short lived. The reason for this restriction is that while an interrupt handler is executing, no other interrupts of the same type can be handled. In many cases no other interrupts at all can be handled, thus making it particularly important to complete the interrupt handler swiftly. In particular, this means that the interrupt handler cannot block waiting for a thread.

In normal execution environments for Java, threads synchronize around garbage collection (GC) to avoid disturbing an ongoing GC. In the case of interrupt handlers, this become impossible. Fruthermore, it is not feasible to let interrupt handlers start a lengthy GC process. Both these facts affect the interoperability of interrupt handlers with a GC.

3.2.1 Stop-the-world GC

Using this strategy the entire heap is collected at once and the collection is not interleaved with execution. The collector can safely assume that data required to do an accurate collection will not change during the collection. Using stop-the-world collection an interrupt handler may not change data used by the GC to complete the collection. In the general case this means that the interrupt handler is not allowed to create new objects, or change the graph of live objects.

3.2.2 Incremental GC

The heap is collected in small incremental steps. Write barriers in the mutator threads and non-preemption sections in the GC thread synchronize the view of the object graph between the mutator threads and the GC thread. Using concurrent collection it should be possible to allow for allocation of objects and changing references inside an interrupt handler (as it is allowed in any normal thread). With a real-time GC the maximum blocking time due to GC synchronization with the mutator threads should be known.

3.2.3 Moving Objects

Interruption of the GC during an object move can result in access to a stale copy of the object inside the handler. A possible solution to this problem is to allow for pinning of objects reachable by the handler (similar to immortal memory in the RTSJ). Concurrent collectors have to solve this issue anyway for the concurrent threads. The simplest approach is to disable thread switching and interrupt handling during the object copy. As this operation can be quite long for large arrays, several approaches to split the array into smaller chunks have been proposed.

4 Supporting Interrupt Handlers

To experiment with our design for in-context interrupt handling we have added such support to the SimpleRTJ interpreter [7] and the experimental Java processor JOP.

4.1 Interrupt Handlers in SimpleRTJ

The SimpleRTJ JVM uses a simple stop-the-world garbage collection scheme. This means that within handlers, we prohibited use of the new keyword and writing references to the heap. Additionally we have turned off the compaction phase of the GC to avoid the problems with moving objects mentioned in Section 3.2.3.

4.1.1 Reentering the JVM

The SimpleRTJ JVM uses green threads. This means that it had to be refactored quite a bit to allow for reentering the JVM from inside the first level interrupt handler. What we did was to get rid of all global state (all global variables) used by the JVM and instead allocate shared data on the C stack. For all parts of the JVM to still be able to access shared data we pass around a single pointer to the shared data now allocated on the stack.

4.1.2 Context Switching at Interrupt

The SimpleRTJ JVM contains support for a skimmed down version of the RTSJ style interrupt handling facilities using the AsyncEvent and AsyncEventHandler classes. Using the javax.events package supplied with the JVM a server thread can be started waiting for events to occur. This server thread runs at highest priority. The SimpleRTJ JVM reschedule points are in between the execution of each bytecode. This

[1] In this section when we use the term "interrupt handler" we mean an interrupt handler executed in-context as described in Section 3

means that before the execution of each bytecode the JVM checks if a new event has been signaled. If so the server thread is scheduled immediately and released to handle the event. To achieve in-context interrupt handling we force a reentry of the JVM from inside the first level interrupt handler by calling the main interpreter loop. Prior to this we have marked that an event is indeed pending, resulting in the server thread being scheduled immediately. To avoid interference with the GC we switch the heap and stack with a new temporary (small) Java heap and a new temporary (small) Java stack. Currently we use 512 bytes for each of these items, which have proven sufficient for running non-trivial interrupt handlers so far.

The major part of the work was to get rid of the global state. How difficult this is, will vary from one JVM implementation to another, but since global state is a bad idea in any case, JVMs of high quality should use very little global state. Using these changes we have experimented with handling the RS232 receive interrupt. The final receive interrupt handler implemented in pure Java is shown in Section 5.

4.2 Interrupt Handlers on JOP

We have implemented a priority based interrupt controller in JOP. The numbers of interrupt lines can be configured. An interrupt can also be triggered in software. There is one global interrupt enable and a local enable for each interrupt line.

In JOP there is a translation stage between Java bytecodes and the JOP internal microcode [8]. On a pending interrupt (or exception generated by the hardware) we can use this translation stage to insert a *special* bytecode into the instruction stream. This trick keeps the interrupt completely transparent to the core pipeline. Interrupts are accepted at bytecode boundaries and clear the global enable flag when accepted. This feature avoids immediate handling of an arriving higher priority interrupt during the first part of the handler. Therefore, the execution of the interrupt handler starts with global disable. The interrupts have to be enabled again by the handler at a *convenient* time.

The special bytecode can be handled in JOP as any other bytecode: execute microcode, invoke a special method from a helper class, or execute Java bytecode from JVM.java.

4.2.1 Interrupt Handling

All interrupts are mapped to one bytecode. Therefore, we perform the dispatch of the correct handler in Java. On an interrupt the static method interrupt() from a system internal class gets invoked. The method reads the interrupt number and performs the dispatch to the registered Runnable. The timer interrupt (index 0) is handled specially. On a timer interrupt the real-time scheduler of JOP gets invoked. At system startup the table of Runnables is initialized with a no-op handler.

```
public class InterruptHandler implements Runnable {
    public static void main(String[] args) {
        InterruptHandler ih = new InterruptHandler();
        IOFactory fact = IOFactory.getFactory();
        // register the handler
        fact.registerInterruptHandler(1, ih);
        // enable interrupt 1
        fact.enableInterrupt(1);
        .....
    }

    public void run() {
        System.out.println("Interrupt fired!");
    }
}
```

Figure 2. An interrupt handler as Runnable

Applications provide handlers via objects that implements Runnable and register the object for a interrupt number. We reuse here the I/O Factory presented in [9]. Figure 2 shows a simple example of an interrupt handler implemented in Java.

For interrupts that should be handled by a sporadic thread under the control of the scheduler, the following needs to be performed on JOP: (1) Create a SwEvent (similar to the RTSJ AsyncEventHandler) that performs the second level interrupt handler work; (2) create a short first level interrupt handler as Runnable and invoke fire() of the corresponding software event handler; (3) register the first level interrupt handler as shown in Figure 2 and start the real-time scheduler.

4.2.2 Garbage Collection

The runtime environment of JOP contains a concurrent real-time GC [10]. The GC can be interrupted at a very fine grain level. During sections that are not preemptive (e.g. data structure manipulation for a new, write barriers on reference field write, object copy during compaction) interrupts are simply turned off. The longest blocking time due to the GC work is on an object or array copy. In [10] we have observed maximum blocking times of 40 μs induced by the GC with medium sized arrays.

5 Using Interrupt Handler

We have not seen any need for adding to the RTSJ style of programming with interrupts (described in Section 3). We have just changed the way that the AsyncEvent gets scheduled. In our approach the server thread bound to the handling of the event gets released immediately inside the context of the first level interrupt handler and not at some later point. Using the skimmed down version of the javax.events package distributed with the SimpleRTJ JVM, the legacy interrupt handler for the RS232 receive interrupt illustrated in Figure 1, can be translated into pure Java as it is shown in Figure 3.

```
public class RS232ReceiveInterruptHandler
  extends InterruptHandler {
    private RS232 rs232;
    private InterruptControl interruptControl;
    private short UartRxBuffer[];
    private byte UartRxWrPtr;

    public RS232ReceiveInterruptHandler(RS232 rs232,
                InterruptControl interruptControl) {
      // Subscribe to the UART receive int.
      super(INT_RS232RX);
      this.rs232 = rs232;
      this.interruptControl = interruptControl;
      UartRxBuffer = new short[32];
      UartRxWrPtr = 0;
    }
    protected void handleInterrupt() {
      UartRxBuffer[UartRxWrPtr++] =
                    rs232.P0_UART_RX_TX_REG;
      if (UartRxWrPtr >= UartRxBuffer.length) {
        UartRxWrPtr = 0;
      }
      rs232.P0_CLEAR_RX_INT_REG = 0;
      interruptControl.RESET_INT_PENDING_REG =
      RS232.CLR_UART_RX_INT_PENDING;
    }
}
```

Figure 3. An example RS232 interrupt handler

5.1 Accessing Device Registers

A very important part of what interrupt handlers normally need to do is to access device registers. To perform this access efficiently, which is a requirement for interrupt handlers, we use hardware objects as defined in [9]. The hardware objects rs232 and interruptControl has been defined to fit the physical hardware platform and allows the interrupt handler to access appropriate device registers directly.

6 Conclusion

We have introduced the concept of in-context interrupt handling and shown its implementation in an interpreter and on a Java processor. An example shows that in-context interrupt handling allows for a greater portion of the interrupt handler to be written in Java. On legacy systems implemented in C/assembler the default is for interrupt handlers to be executed in-context, so adding this option as well on Java based systems will seem natural to experienced programmers of embedded systems.

The proposal has an impact on the safety, portability and maintainability of an application. It is clear that Java code with interrupt handlers may bring the system down, but that is not different from having the handlers in middleware. Yet, the basic safety features of Java (pointer checks, index checks, type checking) are with the proposal brought to bear on such low level code and thus the safety is improved. Interrupt handlers are highly platform dependent and not portable; but they are essential for applications, so placing them outside the Java application only seemingly makes it portable. With the good structuring facilities of packages, classes and interfaces, a well-architected application will preserve portability by placing interrupt handlers in separate hardware abstraction packages. Finally, maintainability will be improved by having one language for an application, where common documentation standards are more likely to be applied.

The current proposal comes in this paper with a proof of concept, but in order for it to really succeed, it needs at some point in time to enter as part of a standard profile and most importantly be included in the JVM platforms.

Acknowledgements

We are grateful to the reviewers for their useful comments.

References

[1] aicas. http://www.aicas.com/jamaica.html. Visited June 2007.

[2] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[3] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[4] J. M. Enery, D. Hickey, and M. Boubekeur. Empirical evaluation of two main-stream rtsj implementations. In *JTRES '07: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 47–54, New York, NY, USA, 2007. ACM.

[5] J. Kwon, A. Wellings, and S. King. Ravenscar-Java: A high integrity profile for real-time Java. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 131–140. ACM Press, 2002.

[6] S. Liang. *The Java Native Interface - Programmers Guide and Specification*. Addison-Wesley, 1999.

[7] RTJComputing. http://www.rtjcom.com. Visited June 2007.

[8] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Article in press and online: Journal of Systems Architecture*, doi:10.1016/j.sysarc.2007.06.001, 2007.

[9] M. Schoeberl, S. Korsholm, C. Thalinger, and A. P. Ravn. Hardware objects for Java. In *Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008)*, Orlando, Florida, USA, May 2008. IEEE Computer Society.

[10] M. Schoeberl and J. Vitek. Garbage collection for safety critical Java. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 85–93, Vienna, Austria, September 2007. ACM Press.

[11] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java on the bare metal of wireless sensor devices: the squawk java virtual machine. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 78–88. ACM Press, 2006.

A Hardware Abstraction Layer in Java

MARTIN SCHOEBERL
Vienna University of Technology, Austria
STEPHAN KORSHOLM
Aalborg University, Denmark
TOMAS KALIBERA
Purdue University, USA
and
ANDERS P. RAVN
Aalborg University, Denmark

Embedded systems use specialized hardware devices to interact with their environment, and since they have to be dependable, it is attractive to use a modern, type-safe programming language like Java to develop programs for them. Standard Java, as a platform independent language, delegates access to devices, direct memory access, and interrupt handling to some underlying operating system or kernel, but in the embedded systems domain resources are scarce and a Java virtual machine (JVM) without an underlying middleware is an attractive architecture. The contribution of this paper is a proposal for Java packages with hardware objects and interrupt handlers that interface to such a JVM. We provide implementations of the proposal directly in hardware, as extensions of standard interpreters, and finally with an operating system middleware. The latter solution is mainly seen as a migration path allowing Java programs to coexist with legacy system components. An important aspect of the proposal is that it is compatible with the Real-Time Specification for Java (RTSJ).

Categories and Subject Descriptors: D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; D.3.3 [**Programming Languages**]: Language Classifications—*Object-oriented languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Input/output*

General Terms: Languages, Design, Implementation

Additional Key Words and Phrases: Device driver, embedded system, Java, Java virtual machine

1. INTRODUCTION

When developing software for an embedded system, for instance an instrument, it is necessary to control specialized hardware devices, for instance a heating element or an interferometer mirror. These devices are typically interfaced to the processor through device registers and may use interrupts to synchronize with the processor. In order to make the

Author's address: Martin Schoeberl, Institute of Computer Engineering, Vienna University of Technology, Treitlstr. 3, A-1040 Vienna, Austria; email: mschoebe@mail.tuwien.ac.at. Stephan Korsholm and Anders P. Ravn, Department of Computer Science, Aalborg University, Selma Lagerlöfs vej 300, DK-9220 Aalborg, Denmark; email stk,apr@cs.aau.dk. Tomas Kalibera, Department of Computer Science, Purdue University, 305 N. University Street, West Lafayette, IN 47907-2107, USA; email: kalibera@cs.purdue.edu.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2009 ACM 1539-9087/2009/0?00-0001 $5.00

programs easier to understand, it is convenient to introduce a *hardware abstraction layer* (HAL), where access to device registers and synchronization through interrupts are hidden from conventional program components. A HAL defines an interface in terms of the constructs of the programming language used to develop the application. Thus, the challenge is to develop an abstraction that gives efficient access to the hardware, while staying within the computational model provided by the programming language.

Our first ideas on a HAL for Java have been published in [Schoeberl et al. 2008] and [Korsholm et al. 2008]. This paper combines the two papers, provides a much wider background of related work, gives two additional experimental implementations, and gives performance measurements that allow an assessment of the efficiency of the implementations. The remainder of this section introduces the concepts of the Java based HAL.

1.1 Java for Embedded Systems

Over the nearly 15 years of its existence Java has become a popular programming language for desktop and server applications. The concept of the Java virtual machine (JVM) as the execution platform enables portability of Java applications. The language, its API specification, as well as JVM implementations have matured; Java is today employed in large scale industrial applications. The automatic memory management takes away a burden from the application programmers and together with type safety helps to isolate problems and, to some extent, even run untrusted code. It also enhances security – attacks like stack overflow are not possible. Java integrates threading support and dynamic loading into the language, making these features easily accessible on different platforms. The Java language and JVM specifications are proven by different implementations on different platforms, making it relatively easy to write platform independent Java programs that run on different JVM implementations and underlying OS/hardware. Java has a standard API for a wide range of libraries, the use of which is thus again platform independent. With the ubiquity of Java, it is easy to find qualified programmers which know the language, and there is strong tool support for the whole development process. According to an experimental study [Phipps 1999], Java has lower bug rates and higher productivity rates than C++. Indeed, some of these features come at a price of larger footprint (the virtual machine is a non-trivial piece of code), typically higher memory requirements, and sometimes degraded performance, but this cost is accepted in industry.

Recent real-time Java virtual machines based on the Real-Time Specification for Java (RTSJ) provide controlled and safe memory allocation. Also there are platforms for less critical systems with real-time garbage collectors. Thus, Java is ready to make its way into the embedded systems domain. Mobile phones, PDAs, or set-top boxes run Java Micro Edition, a Java platform with a restricted set of standard Java libraries. Real-time Java has been and is being evaluated as a potential future platform for space avionics both by NASA and ESA space agencies. Some Java features are even more important for embedded than for desktop systems because of missing features of the underlying platform. For instance the RTEMS operating system used by ESA for space missions does not support hardware memory protection even for CPUs that do support it (like LEON3, a CPU for ESA space missions). With Java's type safety hardware protection is not needed to spatially isolate applications. Moreover, RTEMS does not support dynamic libraries, but Java can load classes dynamically.

Many embedded applications require very small platforms, therefore it is interesting to remove as much as possible of an underlying operating system or kernel, where a major

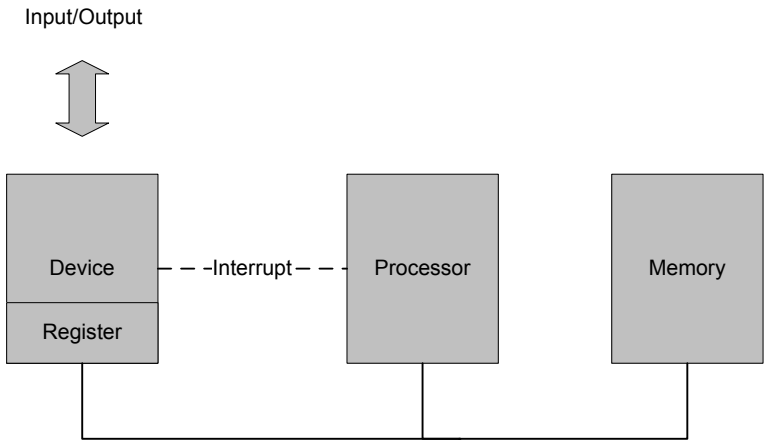


Fig. 1. The hardware: a bus connects a processor to device registers and memory, and an interrupt bus connects devices to a processor

part of code is dedicated to handling devices. Furthermore, Java is considered as the future language for safety-critical systems [Henties et al. 2009]. As certification of safety-critical systems is very expensive, the usual approach is to minimize the code base and supporting tools. Using two languages (e.g., C for programming device handling in the kernel and Java for implementing the processing of data) increases the complexity of generating a safety case. A Java only system reduces the complexity of the tool support and therefore the certification effort. Even in less critical systems the same issues will show up as decreased productivity and dependability of the software. Thus it makes sense to investigate a general solution that interfaces Java to the hardware platform; that is the objective of the work presented here.

1.2 Hardware Assumptions

The hardware platform is built up along one or more buses – in small systems typically only one – that connect the processor with memory and device controllers. Device controllers have reserved some part of the address space of a bus for its device registers. They are accessible for the processor as well, either through special I/O instructions or by ordinary instructions when the address space is the same as the one for addressing memory, a so called memory mapped I/O solution. In some cases the device controller will have direct memory access (DMA) as well, for instance for high speed transfer of blocks of data. Thus the basic communication paradigm between a controller and the processor is shared memory through the device registers and/or through DMA. With these facilities only, synchronization has to be done by testing and setting flags, which means that the processor has to engage in some form of busy waiting. This is eliminated by extending the system with an interrupt bus, where device controllers can generate a signal that interrupts the normal flow of execution in the processor and direct it to an interrupt handling program. Since communication is through shared data structures, the processor and the controllers need a locking mechanism; therefore interrupts can be enabled or disabled by the processor through an interrupt control unit. The typical hardware organization is summarized in Figure 1.

```
public final class ParallelPort {
    public volatile int data;
    public volatile int control;
}

int inval, outval;
myport = JVMMechanism.getParallelPort();
...
inval = myport.data;
myport.data = outval;
```

Fig. 2. The parallel port device as a simple Java class

1.3 A Computational Model

In order to develop a HAL, the device registers and interrupt facilities must be mapped to programming language constructs, such that their use corresponds to the computational model underlying the language. In the following we give simple device examples which illustrate the solution we propose for doing it for Java.

1.3.1 *Hardware Objects.* Consider a simple parallel input/output (PIO) device controlling a set of input and output pins. The PIO uses two registers: the *data register* and the *control register*. Writing to the data register stores the value into an internal latch that drives the output pins. Reading from the data register returns the value that is present on the input pins. The control register configures the direction for each PIO pin. When bit n in the control register is set to 1, pin n drives out the value of bit n of the data register. A 0 at bit n in the control register configures pin n as input pin. At reset the port is usually configured as input port – a safe default configuration.

In an object oriented language the most natural way to represent a device is as an object – the *hardware object*. Figure 2 shows a class definition, object instantiation, and use of the hardware object for the simple parallel port. An instance of the class ParallelPort is the hardware object that represents the PIO. The reference myport points to the hardware object. To provide this convenient representation of devices as objects, a JVM internal mechanism is needed to access the device registers via object fields and to *create* the device object and receive a reference to it. We elaborate on the idea of hardware objects in Section 3.1 and present implementations in Section 4.

1.3.2 *Interrupts.* When we consider an interrupt, it must invoke some program code in a method that handles it. We need to map the interruption of normal execution to some language concept, and here the concept of an asynchronous event is useful. The resulting computational model for the programmer is shown in Figure 3. The signals are external, asynchronous events that map to interrupts.

A layered implementation of this model with a kernel close to the hardware and applications on top has been very useful in general purpose programming. Here one may even extend the kernel to manage resources and provide protection mechanisms such that applications are safe from one another, as for instance when implementing trusted interoperable computing platforms [Group 2008]. Yet there is a price to pay which may make the solution less suitable for embedded systems: adding new device drivers is an error-prone activity [Chou et al. 2001], and protection mechanisms impose a heavy overhead on context switching when accessing devices.

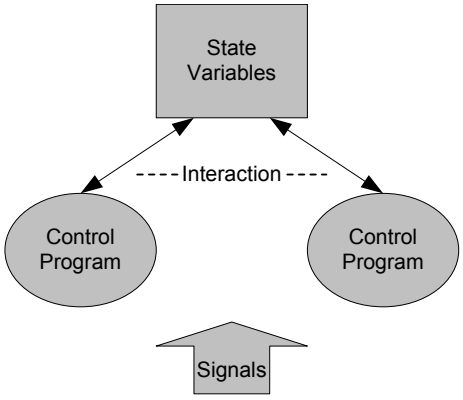


Fig. 3. Computational model: several threads of execution communicate via shared state variables and receive signals.

```
public class RS232ReceiveInterruptHandler extends InterruptHandler {
    private RS232 rs232;
    private InterruptControl interruptControl;

    private byte UartRxBuffer[];
    private short UartRxWrPtr;

    ...

    protected void handle() {

        synchronized(this) {
            UartRxBuffer[UartRxWrPtr++] = rs232.P0_UART_RX_TX_REG;
            if (UartRxWrPtr >= UartRxBuffer.length) UartRxWrPtr = 0;
        }
        rs232.P0_CLEAR_RX_INT_REG = 0;
        interruptControl.RESET_INT_PENDING_REG = RS232.CLR_UART_RX_INT_PENDING;
    }
}
```

Fig. 4. An example interrupt handler for an RS232 interface. On an interrupt the method handle() is invoked. The private objects rs232 and interruptControl are hardware objects that represent the device registers and the interrupt control unit.

The alternative we propose is to use Java directly since it already supports multithreading and use methods in the special InterruptHandler objects to handle interrupts. The idea is illustrated in Figure 4, and the details, including synchronization and interaction with the interrupt control, are elaborated in Section 3.2. Implementations are found in Section 4.

1.4 Mapping Between Java and the Hardware

The proposed interfacing from hardware to Java does not require language extensions. The Java concepts of packages, classes and synchronized objects turn out to be powerful enough to formulate the desired abstractions. The mapping is done at the level of the JVM. The JVM already provides typical OS functions handling:

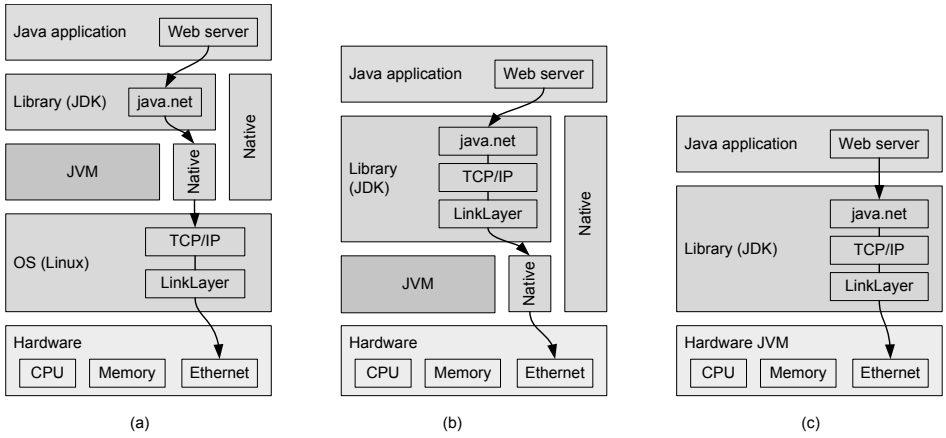


Fig. 5. Configurations for an embedded JVM: (a) standard layers for Java with an operating system – equivalent to desktop configurations, (b) a JVM on the bare metal, and (c) a JVM as a Java processor

—Address space and memory management

—Thread management

—Inter-process communication

These parts need to be modified so they cater for interfaces to the hardware.

Yet, the architectures of JVMs for embedded systems are more diverse than on desktop or server systems. Figure 5 shows variations of Java implementations in embedded systems and an example of the control flow for a web server application. The standard approach with a JVM running on top of an operating system (OS) is shown in sub-figure (a).

A JVM without an OS is shown in sub-figure (b). This solution is often called *running on the bare metal*. The JVM acts as the OS and provides thread scheduling and low-level access to the hardware. In this case the network stack can be written entirely in Java. JNode[1] is an approach to implement the OS entirely in Java. This solution has become popular even in server applications.[2]

Sub-figure (c) shows an embedded solution where the JVM is part of the hardware layer: it is implemented in a Java processor. With this solution the native layer can be completely avoided and all code (application and system code) is written entirely in Java.

Figure 5 shows also the data and control flow from the application down to the hardware. The example consists of a web server and an Internet connection via Ethernet. In case (a) the application web server talks with java.net in the Java library. The flow goes down via a native interface to the TCP/IP implementation and the link layer device driver within the OS (usually written in C). The device driver talks with the Ethernet chip. In (b) the OS layer is omitted: the TCP/IP layer and the link layer device driver are now part of the Java library. In (c) the JVM is part of the hardware layer, and direct access from the link layer driver to the Ethernet hardware is mandatory.

With our proposed HAL, as shown in Figure 6, the native interface within the JVM in (a) and (b) disappears. Note how the network stack moves up from the OS layer to the Java

[1] `http://www.jnode.org/`

[2] BEA System offers the JVM LiquidVM that includes basic OS functions and does not need a guest OS.

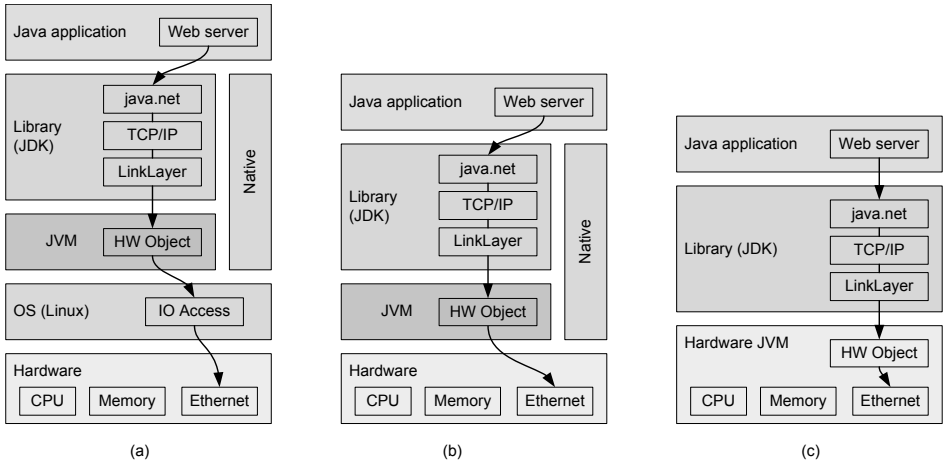


Fig. 6. Configurations for an embedded JVM with hardware objects and interrupt handlers: (a) standard layers for Java with an operating system – equivalent to desktop configurations, (b) a JVM on the bare metal, and (c) a JVM as a Java processor

library in example (a). All three versions show a pure Java implementation of the whole network stack. The Java code is the same for all three solutions. Version (b) and (c) benefit from hardware objects and interrupt handlers in Java as access to the Ethernet device is required from Java source code. In Section 5 we show a simple web server application implemented completely in Java as evaluation of our approach.

1.5 Contributions

The key contribution of this paper is a proposal for a Java HAL that can run on the bare metal while still being *safe*. This idea is investigated in quite a number of places which are discussed in the related work section where we comment on our initial ideas as well. In summary, the proposal gives an interface to hardware that has the following benefits:

Object-oriented. An object representing a device is the most natural integration into an object oriented language, and a method invocation to a synchronized object is a direct representation of an interrupt.

Safe. The safety of Java is not compromised. Hardware objects map object fields to device registers. With a correct class that represents the device, access to it is safe. Hardware objects can be created only by a factory residing in a special package.

Generic. The definition of a hardware object and an interrupt handler is independent of the JVM. Therefore, a common standard for different platforms can be defined.

Efficient. Device register access is performed by single bytecodes getfield and putfield. We avoid expensive native calls. The handlers are first level handlers; there is no delay through event queues.

The proposed Java HAL would not be useful if it had to be modified for each particular kind of JVM; thus a second contribution of this paper is a number of prototype implementations illustrating the architectures presented in Figure 6: implementations in Kaffe [Wilkinson 1996] and OVM [Armbruster et al. 2007] represent the architecture with an

OS (sub-figure (a)), the implementation in SimpleRTJ [RTJ Computing 2000] represents the bare metal solution (sub-figure (b)), and the implementation in JOP [Schoeberl 2008] represents the Java processor solution (sub-figure (c)).

Finally, we must not forget the claim for efficiency, and therefore the paper ends with some performance measurements that indicate that the HAL layer is generally as efficient as native calls to C code external to the JVM.

2. RELATED WORK

Already in the 1970s it was recognized that an operating system might not be the optimal solution for special purpose applications. Device access was integrated into high level programming languages like Concurrent Pascal [Hansen 1977; Ravn 1980] and Modula (Modula-2) [Wirth 1977; 1982] along with a number of similar languages, e.g., UCSD Pascal. They were meant to eliminate the need for operating systems and were successfully used in a variety of applications. The programming language Ada, which has been dominant in defence and space applications till this day, may be seen as a continuation of these developments. The advent of inexpensive microprocessors, from the mid 1980s and on, lead to a regression to assembly and C programming. The hardware platforms were small with limited resources and the developers were mostly electronic engineers, who viewed them as electronic controllers. Program structure was not considered a major issue in development. Nevertheless, the microcomputer has grown, and is now far more powerful than the minicomputer that it replaced. With powerful processors and an abundance of memory, the ambitions for the functionality of embedded systems grow, and programming becomes a major issue because it may turn out to be the bottleneck in development. Consequently, there is a renewed interest in this line of research.

An excellent overview of historical solutions to access hardware devices from and implement interrupt handlers in high-level languages, including C, is presented in Chapter 15 of [Burns and Wellings 2001]. The solution to device register access in Modula-1 (Ch. 15.3) is very much like C; however the constructs are safer because they are encapsulated in modules. Interrupt handlers are represented by threads that block to wait for the interrupt. In Ada (Ch 15.4) the representation of individual fields in registers can be described precisely by *representation* classes, while the corresponding structure is bound to a location using the Address attribute. An interrupt is represented in the current version of Ada by a protected procedure, although initially represented (Ada 83) by task entry calls.

The main ideas in having device objects are thus found in the earlier safe languages, and our contribution is to align them with a Java model, and in particular, as discussed in Section 4, implementation in a JVM. From the Ada experience we learn that direct handling of interrupts is a desired feature.

2.1 The Real-Time Specification for Java

The Real-Time Specification for Java (RTSJ) [Bollella et al. 2000] defines a JVM extension which allows better timeliness control compared to a standard JVM. The core features are: fixed priority scheduling, monitors which prevent priority inversion, scoped memory for objects with limited lifetime, immortal memory for objects that are never finalized, and asynchronous events with CPU time consumption control.

The RTSJ also defines an API for direct access to physical memory, including hardware registers. Essentially one uses RawMemoryAccess at the level of primitive data types. Although the solution is efficient, this representation of physical memory is not object

oriented, and there are some safety issues: When one raw memory area represents an address range where several devices are mapped to, there is no protection between them. Yet, a type safe layer with support for representing individual registers can be implemented on top of the RTSJ API.

The RTSJ specification suggests that asynchronous events are used for interrupt handling. Yet, it neither specifies an API for interrupt control nor semantics of the handlers. Any interrupt handling application thus relies on some proprietary API and proprietary event handler semantics. Second level interrupt handling can be implemented within the RTSJ with an AsyncEvent that is bound to a *happening*. The happening is a string constant that represents an interrupt, but the meaning is implementation dependent. An AsyncEventHandler or BoundAsyncEventHandler can be added as handler for the event. Also an AsyncEventHandler can be added via a POSIXSignalHandler to handle POSIX signals. An interrupt handler, written in C, can then use one of the two available POSIX user signals.

RTSJ offers facilities very much in line with Modula or Ada for encapsulating memory-mapped device registers. However, we are not aware of any RTSJ implementation that implements RawMemoryAccess and AsyncEvent with support for low-level device access and interrupt handling. Our solution could be used as specification of such an extension. It would still leave the first level interrupt handling hidden in an implementation; therefore an interesting idea is to define and implement a two-level scheduler for the RTSJ. It should provide the first level interrupt handling for asynchronous events bound to interrupts and delegate other asynchronous events to an underlying second level scheduler, which could be the standard fixed priority preemptive scheduler. This would be a fully RTSJ compliant implementation of our proposal.

2.2 Hardware Interface in JVMs

The aJile Java processor [aJile 2000] uses native functions to access devices. Interrupts are handled by registering a handler for an interrupt source (e.g., a GPIO pin). Systronix suggests[3] to keep the handler short, as it runs with interrupts disabled, and delegate the real handling to a thread. The thread waits on an object with ceiling priority set to the interrupt priority. The handler just notifies the waiting thread through this monitor. When the thread is unblocked and holds the monitor, effectively all interrupts are disabled.

Komodo [Kreuzinger et al. 2003] is a multithreaded Java processor targeting real-time systems. On top of the multiprocessing pipeline the concept of interrupt service threads is implemented. For each interrupt one thread slot is reserved for the interrupt service thread. It is unblocked by the signaling unit when an interrupt occurs. A dedicated thread slot on a fine-grain multithreading processor results in a very short latency for the interrupt service routine. No thread state needs to be saved. However, this comes at the cost to store the complete state for the interrupt service thread in the hardware. In the case of Komodo, the state consists of an instruction window and the on-chip stack memory. Devices are represented by Komodo specific I/O classes.

Muvium [Caska 2009] is an ahead-of-time compiling JVM solution for very resource constrained microcontrollers (Microchip PIC). Muvium uses an Abstract Peripheral Toolkit (APT) to represent devices. APT is based on an event driven model for interaction with the external world. Device interrupts and periodic activations are represented by events. Internally, events are mapped to threads with priority dispatched by a preemptive sched-

[3] A template can be found at `http://practicalembeddedjava.com/tutorials/aJileISR.html`

uler. APT contains a large collection of classes to represent devices common in embedded systems.

In summary, access to device registers is handled in both aJile, Komodo, and Muvium by abstracting them into library classes with access methods. This leaves the implementation to the particular JVM and does not give the option of programming them at the Java level. It means that extension with new devices involve programming at different levels, which we aim to avoid. Interrupt handling in aJile is essentially first level, but with the twist that it may be interpreted as RTSJ event handling, although the firing mechanism is atypical. Our mechanism would free this binding and allow other forms of programmed notification, or even leaving out notification altogether. Muvium follows the line of RTSJ and has a hidden first level interrupt handling. Komodo has a solution with first level handling through a full context switch; this is very close to the solution advocated in Modula 1, but it has in general a larger overhead than we would want to incur.

2.3 Java Operating Systems

The JX Operating System [Felser et al. 2002] is a microkernel system written mostly in Java. The system consists of components which run in *domains*, each domain having its own garbage collector, threads, and a scheduler. There is one global preemptive scheduler that schedules the domain schedulers which can be both preemptive and non-preemptive. Inter-domain communication is only possible through communication channels exported by services. Low level access to the physical memory, memory mapped device registers, and I/O ports are provided by the core ("zero") domain services, implemented in C. At the Java level ports and memory areas are represented by objects, and registers are methods of these objects. Memory is read and written by access methods of Memory objects. Higher layers of Java interfaces provide type safe access to the registers; the low level access is not type safe.

Interrupt handlers in JX are written in Java and are run through portals – they can reside in any domain. Interrupt handlers cannot interrupt the garbage collector (the GC disables interrupts), run with CPU interrupts disabled, must not block, and can only allocate a restricted amount of memory from a reserved per domain heap. Execution time of interrupt handlers can be monitored: on a deadline violation the handler is aborted and the interrupt source disabled. The first level handlers can unblock a waiting second level thread either directly or via setting a state of a AtomicVariable synchronization primitive.

The Java New Operating System Design Effort (JNode[4]) [Lohmeier 2005] is an OS written in Java where the JVM serves as the OS. Drivers are written entirely in Java. Device access is performed via native function calls. A first level interrupt handler, written in assembler, unblocks a Java interrupt thread. From this thread the device driver level interrupt handler is invoked with interrupts disabled. Some device drivers implement a synchronized handleInterrupt(int irq) and use the driver object to signal the upper layer with notifyAll(). During garbage collection all threads are stopped including the interrupt threads.

The Squawk VM [Simon et al. 2006], now available open-source,[5] is a platform for wireless sensors. Squawk is mostly written in Java and runs without an OS. Device drivers are written in Java and use a form of peek and poke interface to access the device registers. Interrupt handling is supported by a device driver thread that waits for an event from the

[4] http://jnode.org/

[5] https://squawk.dev.java.net/

JVM. The first level handler, written in assembler, disables the interrupt and notifies the JVM. On a rescheduling point the JVM resumes the device driver thread. It has to re-enable the interrupt. The interrupt latency depends on the rescheduling point and on the activity of the garbage collector. For a single device driver thread an average case latency of 0.1 ms is reported. For a realistic workload with an active garbage collector a worst-case latency of 13 ms has been observed.

Our proposed constructs should be able to support the Java operating systems. For JX we observe that the concepts are very similar for interrupt handling, and actually for device registers as well. A difference is that we make device objects distinct from memory objects which should give better possibilities for porting to architectures with separate I/O-buses. JNode is more traditional and hides first level interrupt handling and device accesses in the JVM, which may be less portable than our implementation. The Squawk solution has to have a very small footprint, but on the other hand it can probably rely on having few devices. Device objects would be at least as efficient as the peeks and pokes, and interrupt routines may eliminate the need for multithreading for simple systems, e.g., with cyclic executives. Overall, we conclude that our proposed constructs will make implementation of a Java OS more efficient and perhaps more portable.

2.4 TinyOS and Singularity

TinyOS [Hill et al. 2000] is an operating system designed for low-power, wireless sensor networks. TinyOS is not a a traditional OS, but provides a framework of components that are linked with the application code. The component-based programming model is supported by nesC [Gay et al. 2003], a dialect of C. TinyOS components provide following abstractions: *commands* represent requests for a service of a component; *events* signal the completion of a service; and *tasks* are functions executed non-preemptive by the TinyOS scheduler. Events also represent interrupts and preempt tasks. An event handler may post a task for further processing, which is similar to a 2nd level interrupt handler.

I/O devices are encapsulated in components and the standard distribution of TinyOS includes a rich set of standard I/O devices. A Hardware Presentation Layer (HPL) abstracts the platform specific access to the hardware (either memory or port mapped). Our proposed HAL is similar to the HPL, but represents the I/O devices as Java objects. A further abstractions into I/O components can be built above our presented Java HAL.

Singularity [Hunt et al. 2005] is a research OS based on a runtime managed language (an extension of C#) to build a software platform with the main goal to be dependable. A small HAL (IoPorts, IoDma, IoIrq, and IoMemory) provides access to PC hardware. C# style attributes (similar to Java annotations) on fields are used to define the mapping of class fields to I/O ports and memory addresses. The Singularity OS clearly uses device objects and interrupt handlers, thus demonstrating that the ideas presented here transfer to a language like C#.

2.5 Summary

In our analysis of related work we see that our contribution is a selection, adaptation, refinement, and implementation of ideas from earlier languages and platforms for Java. A crucial point, where we have spent much time, is to have a clear interface between the Java layer and the JVM. Here we have used the lessons from the Java OS and the JVM interfaces. Finally, it has been a concern to be consistent with the RTSJ because

```
public abstract class HardwareObject {
    HardwareObject() {};
}
```

Fig. 7. The marker class for hardware objects

```
public final class SerialPort extends HardwareObject {

    public static final int MASK_TDRE = 1;
    public static final int MASK_RDRF = 2;

    public volatile int status;
    public volatile int data;

    public void init(int baudRate) {...}
    public boolean rxFull() {...}
    public boolean txEmpty() {...}
}
```

Fig. 8. A serial port class with device methods

this standard and adaptations of it are the instruments for developing embedded real-time software in Java.

3. THE HARDWARE ABSTRACTION LAYER

In the following section the hardware abstraction layer for Java is defined. Low-level access to devices is performed via hardware objects. Synchronization with a device can be performed with interrupt handlers implemented in Java. Finally, portability of hardware objects, interrupt handlers, and device drivers is supported by a generic configuration mechanism.

3.1 Device Access

Hardware objects map object fields to device registers. Therefore, field access with bytecodes putfield and getfield accesses device registers. With a correct class that represents a device, access to it is safe – it is not possible to read or write to an arbitrary memory address. A memory area (e.g., a video frame buffer) represented by an array is protected by Java's array bounds check.

In a C based system the access to I/O devices can either be represented by a C struct (similar to the class shown in Figure 2) for memory mapped I/O devices or needs to be accessed by function calls on systems with a separate I/O address space. With the hardware object abstraction in Java the JVM can represent an I/O device as a class independent of the underlying low-level I/O mechanism. Furthermore, the strong typing of Java avoids hard to find programming errors due to wrong pointer casts or wrong pointer arithmetic.

All hardware classes have to extend the abstract class HardwareObject (see Figure 7). This empty class serves as type marker. Some implementations use it to distinguish between plain objects and hardware objects for the field access. The package visible only constructor disallows creation of hardware objects by the application code that resides in a different package. Figure 8 shows a class representing a serial port with a status register and a data register. The status register contains flags for receive register full and transmit

```
public final class SysCounter extends HardwareObject {

    public volatile int counter;
    public volatile int timer;
    public volatile int wd;
}

public final class AppCounter extends HardwareObject {

    public volatile int counter;
    private volatile int timer;
    public volatile int wd;
}

public final class AppGetterSetter extends HardwareObject {

    private volatile int counter;
    private volatile int timer;
    private volatile int wd;

    public int getCounter() {
        return counter;
    }

    public void setWd(boolean val) {
        wd = val ? 1 : 0;
    }
}
```

Fig. 9. System and application classes, one with visibility protection and one with setter and getter methods, for a single hardware device

register empty; the data register is the receive and transmit buffer. Additionally, we define device specific constants (bit masks for the status register) in the class for the serial port. All fields represent device registers that can change due to activity of the hardware device. Therefore, they must be declared volatile.

In this example we have included some convenience methods to access the hardware object. However, for a clear separation of concerns, the hardware object represents only the device state (the registers). We do not add instance fields to represent additional state, i.e., mixing device registers with heap elements. We cannot implement a complete device driver within a hardware object; instead a complete device driver owns a number of private hardware objects along with data structures for buffering, and it defines interrupt handlers and methods for accessing its state from application processes. For device specific operations, such as initialization of the device, methods in hardware objects are useful.

Usually each device is represented by exactly one hardware object (see Section 3.3.1). However, there might be use cases where this restriction should be relaxed. Consider a device where some registers should be accessed by system code only and some other by application code. In JOP there is such a device: a system device that contains a 1 MHz counter, a corresponding timer interrupt, and a watchdog port. The timer interrupt is programmed relative to the counter and used by the real-time scheduler – a JVM internal service. However, access to the counter can be useful for the application code. Access

```
import com.jopdesign.io.*;

public class Example {

    public static void main(String[] args) {

        BaseBoard fact = BaseBoard.getBaseFactory();
        SerialPort sp = fact.getSerialPort();

        String hello = "Hello World!";

        for (int i=0; i<hello.length(); ++i) {
            // busy wait on transmit buffer empty
            while ((sp.status & SerialPort.MASK_TDRE) == 0)
                ;
            // write a character
            sp.data = hello.charAt(i);
        }
    }
}
```

Fig. 10. A 'Hello World' example with low-level device access via a hardware object

to the watchdog register is required from the application level. The watchdog is used for a sign-of-life from the application. If not triggered every second the complete system is restarted. For this example it is useful to represent one hardware device by two *different* classes – one for system code and one for application code. We can protect system registers by private fields in the hardware object for the application. Figure 9 shows the two class definitions that represent the same hardware device for system and application code respectively. Note how we changed the access to the timer interrupt register to private for the application hardware object.

Another option, shown in class AppGetterSetter, is to declare all fields private for the application object and use setter and getter methods. They add an abstraction on top of hardware objects and use the hardware object to implement their functionality. Thus we still do not need to invoke native functions.

Use of hardware objects is straightforward. After obtaining a reference to the object all that has to be done (or can be done) is to read from and write to the object fields. Figure 10 shows an example of client code. The example is a *Hello World* program using low-level access to a serial port via a hardware object. Creation of hardware objects is more complex and described in Section 3.3. Furthermore, it is JVM specific and Section 4 describes implementations in four different JVMs.

For devices that use DMA (e.g., video frame buffer, disk, and network I/O buffers) we map that memory area to Java arrays. Arrays in Java provide access to raw memory in an elegant way: the access is simple and safe due to the array bounds checking done by the JVM. Hardware arrays can be *used* by the JVM to *implement* higher-level abstractions from the RTSJ such as RawMemory or scoped memory.

Interaction between the garbage collector (GC) and hardware objects needs to be crafted into the JVM. We do not want to *collect* hardware objects. The hardware object should not

ISR	Context switches	Priorities
Handler	2	Hardware
Event	3–4	Software

Table I. Dispatching properties of different ISR strategies

be scanned for references.[6] This is permissible when only primitive types are used in the class definition for hardware objects – the GC scans only reference fields. To avoid collecting hardware objects, we *mark* the object to be skipped by the GC. The type inheritance from HardwareObject can be used as the marker.

3.2 Interrupt Handling

An interrupt service routine (ISR) can be integrated with Java in two different ways: as a first level *handler* or a second level *event* handler.

ISR handler. The interrupt is a method call initiated by the device. Usually this abstraction is supported in hardware by the processor and called a first level handler.

ISR event. The interrupt is represented by an asynchronous notification directed to a thread that is unblocked from a wait-state. This is also called deferred interrupt handling.

An overview of the dispatching properties of the two approaches is given in Table I. The ISR handler approach needs only two context switches and the priority is set by the hardware. With the ISR event approach, handlers are scheduled at software priorities. The initial first level handler, running at hardware priority, fires the event for the event handler. Also the first level handler will notify the scheduler. In the best case three context switches are necessary: one to the first level handler, one to the ISR event handler, and one back to the interrupted thread. If the ISR handler has a lower priority than the interrupted thread, an additional context switch from the first level handler back to the interrupted thread is necessary.

Another possibility is to represent an interrupt as a thread that is released by the interrupt. Direct support by the hardware (e.g., the interrupt service thread in Komodo [Kreuzinger et al. 2003]) gives fast interrupt response times. However, standard processors support only the handler model directly.

Direct handling of interrupts in Java requires the JVM to be prepared to be interrupted. In an interpreting JVM an initial handler will reenter the JVM to execute the Java handler. A compiling JVM or a Java processor can directly invoke a Java method as response to the interrupt. A compiled Java method can be registered directly in the ISR dispatch table.

If an internal scheduler is used (also called *green threads*) the JVM will need some refactoring in order to support asynchronous method invocation. Usually JVMs control the rescheduling at the JVM level to provide a lightweight protection of JVM internal data structures. These preemption points are called pollchecks or yield points; also some or all can be GC preemption points. In fact the preemption points resemble cooperative scheduling at the JVM level and use priority for synchronization. This approach works only for uniprocessor systems, for multiprocessors explicit synchronization has to be introduced.

[6] If a hardware coprocessor, represented by a hardware object, itself manipulates an object on the heap and holds the only reference to that object it has to be scanned by the GC.

In both cases there might be critical sections in the JVM where reentry cannot be allowed. To solve this problem the JVM must disable interrupts around critical non-reenterable sections. The more fine grained this disabling of interrupts can be done, the more responsive to interrupts the system will be.

One could opt for second level handlers only. An interrupt fires and releases an associated schedulable object (handler). Once released, the handler will be scheduled by the JVM scheduler according to the release parameters. This is the RTSJ approach. The advantage is that interrupt handling is done in the context of a normal Java thread and scheduled as any other thread running on the system. The drawback is that there will be a delay from the occurrence of the interrupt until the thread gets scheduled. Additionally, the meaning of interrupt priorities, levels and masks used by the hardware may not map directly to scheduling parameters supported by the JVM scheduler.

In the following we focus on the ISR handler approach, because it allows programming the other paradigms within Java.

3.2.1 *Hardware Properties.* We assume interrupt hardware as it is found in most computer architectures: interrupts have a fixed priority associated with them – they are set with a *solder iron*. Furthermore, interrupts can be globally disabled. In most systems the first level handler is called with interrupts globally disabled. To allow nested interrupts – being able to interrupt the handler by a higher priority interrupt as in preemptive scheduling – the handler has to enable interrupts again. However, to avoid priority inversion between handlers only interrupts with a higher priority will be enabled, either by setting the interrupt level or setting the interrupt mask. Software threads are scheduled by a timer interrupt and usually have a lower priority than interrupt handlers (the timer interrupt has the lowest priority of all interrupts). Therefore, an interrupt handler is never preempted by a software thread.

Mutual exclusion between an interrupt handler and a software thread is ensured by disabling interrupts: either all interrupts or selectively. Again, to avoid priority inversion, only interrupts of a higher priority than the interrupt that shares the data with the software thread can be enabled. This mechanism is in effect the priority ceiling emulation protocol [Sha et al. 1990], sometimes called immediate ceiling protocol. It has the virtue that it eliminates the need for explicit locks (or Java monitors) on shared objects. Note that mutual exclusion with interrupt disabling works only in a uniprocessor setting. A simple solution for multiprocessors is to run the interrupt handler and associated software threads on the same processor core. A more involved scheme would be to use spin-locks between the processors.

When a device asserts an interrupt request line, the interrupt controller notifies the processor. The processor stops execution of the current thread. A partial thread context (program counter and processor status register) is saved. Then the ISR is looked up in the interrupt vector table and a jump is performed to the first instruction of the ISR. The handler usually saves additional thread context (e.g. the register file). It is also possible to switch to a new stack area. This is important for embedded systems where the stack sizes for all threads need to be determined at link time.

3.2.2 *Synchronization.* Java supports synchronization between Java threads with the synchronized keyword, either as a means of synchronizing access to a block of statements or to an entire method. In the general case this existing synchronization support is not

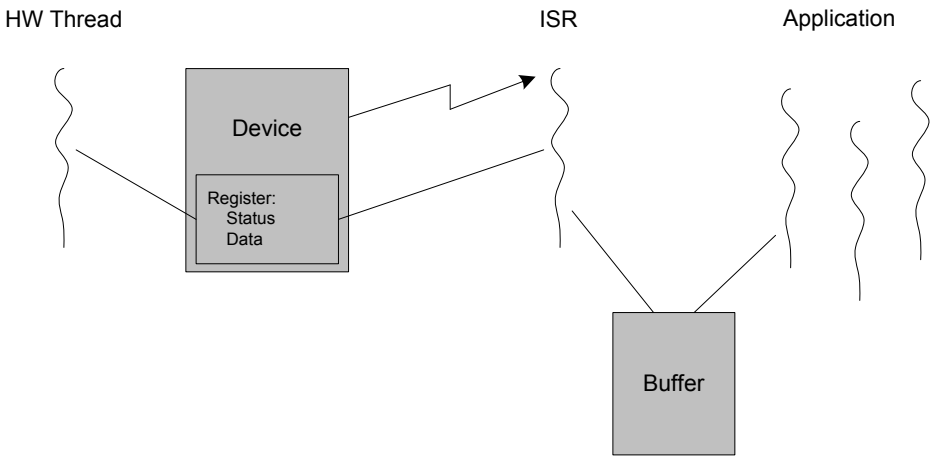


Fig. 11. Threads and shared data

sufficient to synchronize between interrupt handlers and threads.

Figure 11 shows the interacting active processes and the shared data in a scenario involving the handling of an interrupt. Conceptually three threads interact: (1) a hardware device thread representing the device activity, (2) the ISR, and (3) the application or device driver thread. These three share two types of data:

Device data. The hardware thread and ISR share access to the device registers of the device signaling the interrupt

Application data. The ISR and application or device driver share access to e.g., a buffer conveying information about the interrupt to the application

Regardless of which interrupt handling approach is used in Java, synchronization between the ISR and the device registers must be handled in an ad hoc way. In general there is no guarantee that the device has not changed the data in its registers; but if the ISR can be run to completion within the minimum inter-arrival time of the interrupt the content of the device registers can be trusted.

For synchronization between the ISR and the application (or device driver) the following mechanisms are available. When the ISR handler runs as a software thread, standard synchronization with object monitors can be used. When using the ISR handler approach, the handler is no longer scheduled by the normal Java scheduler, but by the hardware. While the handler is running, all other executable elements are suspended, including the scheduler. This means that the ISR cannot be suspended, must not block, or must not block via a language level synchronization mechanism; the ISR must run to completion in order not to freeze the system. This means that when the handler runs, it is guaranteed that the application will not get scheduled. It follows that the handler can access data shared with the application without synchronizing with the application. As the access to the shared data by the interrupt handler is not explicitly protected by a synchronized method or block, the shared data needs to be declared volatile.

On the other hand the application must synchronize with the ISR because the ISR may be dispatched at any point. To ensure mutual exclusion we redefine the semantics of the

```
public class SerialHandler extends InterruptHandler {

    // A hardware object represents the serial device
    private SerialPort sp;

    final static int BUF_SIZE = 32;
    private volatile byte buffer[];
    private volatile int wrPtr, rdPtr;

    public SerialHandler(SerialPort sp) {
        this.sp = sp;
        buffer = new byte[BUF_SIZE];
        wrPtr = rdPtr = 0;
    }

    // This method is scheduled by the hardware
    public void handle() {
        byte val = (byte) sp.data;
        // check for buffer full
        if ((wrPtr+1)%BUF_SIZE!=rdPtr) {
            buffer[wrPtr++] = val;
        }
        if (wrPtr>=BUF_SIZE) wrPtr=0;
        // enable interrupts again
        enableInterrupt();
    }

    // This method is invoked by the driver thread
    synchronized public int read() {
        if (rdPtr!=wrPtr) {
            int val = ((int) buffer[rdPtr++]) & 0xff;
            if (rdPtr>=BUF_SIZE) rdPtr=0;
            return val;
        } else {
            return -1;        // empty buffer
        }
    }
}
```

Fig. 12. An interrupt handler for a serial port receive interrupt

monitor associated with an InterruptHandler object: acquisition of the monitor disables all interrupts of the same and lower priority; release of the monitor enables the interrupts again. This procedure ensures that the software thread cannot be interrupted by the interrupt handler when accessing shared data.

3.2.3 *Using the Interrupt Handler.* Figure 12 shows an example of an interrupt handler for the serial port receiver interrupt. The method handle() is the interrupt handler method and needs no synchronization as it cannot be interrupted by a software thread. However, the shared data needs to be declared volatile as it is changed by the device driver thread. Method read() is invoked by the device driver thread and the shared data is protected by the InterruptHandler monitor. The serial port interrupt handler uses the hardware object SerialPort to access the device.

```
package com.board-vendor.io;

public class IOSystem {

    // some JVM mechanism to create the hardware objects
    private static ParallelPort pp = jvmPPCreate();
    private static SerialPort sp = jvmSPCreate(0);
    private static SerialPort gps = jvmSPCreate(1);

    public static ParallelPort getParallelPort() {
        return pp;
    }

    public static SerialPort getSerialPort() {..}
    public static SerialPort getGpsPort() {..}
}
```

Fig. 13. A factory with static methods for Singleton hardware objects

3.2.4 *Garbage Collection.* When using the ISR handler approach it is not feasible to let interrupt handlers be paused during a lengthy stop-the-world collection. Using this GC strategy the entire heap is collected at once and it is not interleaved with execution. The collector can safely assume that data required to perform the collection will not change during the collection, and an interrupt handler shall not change data used by the GC to complete the collection. In the general case, this means that the interrupt handler is not allowed to create new objects, or change the graph of live objects.

With an incremental GC the heap is collected in small incremental steps. Write barriers in the mutator threads and non-preemption sections in the GC thread synchronize the view of the object graph between the mutator threads and the GC thread. With incremental collection it is possible to allow object allocation and changing references inside an interrupt handler (as it is allowed in any normal thread). With a real-time GC the maximum blocking time due to GC synchronization with the mutator threads must be known.

Interruption of the GC during an object move can result in access to a stale copy of the object inside the handler. A solution to this problem is to allow for pinning of objects reachable by the handler (similar to immortal memory in the RTSJ). Concurrent collectors have to solve this issue for threads anyway. The simplest approach is to disable interrupt handling during the object copy. As this operation can be quite long for large arrays, several approaches to split the array into smaller chunks have been proposed [Siebert 2002] and [Bacon et al. 2003]. A Java processor may support incremental array copying with redirection of the access to the correct part of the array [Schoeberl and Puffitsch 2008]. Another solution is to abort the object copy when writing to the object. It is also possible to use replication – during an incremental copy operation, writes are performed on both from-space and to-space object replicas, while reads are performed on the from-space replica.

3.3 Generic Configuration

An important issue for a HAL is a safe abstraction of device configurations. A definition of factories to create hardware and interrupt objects should be provided by board vendors. This configuration is isolated with the help of Java packages – only the objects and the factory methods are visible. The configuration abstraction is independent of the JVM.

```
public class BaseBoard {

    private final static int SERIAL_ADDRESS = ...;
    private SerialPort serial;
    BaseBoard() {
        serial = (SerialPort) jvmHWOCreate(SERIAL_ADDRESS);
    };
    static BaseBoard single = new BaseBoard();
    public static BaseBoard getBaseFactory() {
        return single;
    }
    public SerialPort getSerialPort() { return serial; }

    // here comes the JVM internal mechanism
    Object jvmHWOCreate(int address) {...}
}

public class ExtendedBoard extends BaseBoard {

    private final static int GPS_ADDRESS = ...;
    private final static int PARALLEL_ADDRESS = ...;
    private SerialPort gps;
    private ParallelPort parallel;
    ExtendedBoard() {
        gps = (SerialPort) jvmHWOCreate(GPS_ADDRESS);
        parallel = (ParallelPort) jvmHWOCreate(PARALLEL_ADDRESS);
    };
    static ExtendedBoard single = new ExtendedBoard();
    public static ExtendedBoard getExtendedFactory() {
        return single;
    }
    public SerialPort getGpsPort() { return gps; }
    public ParallelPort getParallelPort() { return parallel; }
}
```

Fig. 14. A base class of a hardware object factory and a factory subclass

A device or interrupt can be represented by an identical hardware or interrupt object for different JVMs. Therefore, device drivers written in Java are JVM independent.

3.3.1 *Hardware Object Creation.* The idea to represent each device by a single object or array is straightforward, the remaining question is: How are those objects created? An object that represents a device is a typical Singleton [Gamma et al. 1994]. Only a single object should map to one instance of a device. Therefore, hardware objects cannot be instantiated by a simple new: (1) they have to be mapped by some JVM mechanism to the device registers and (2) each device instance is represented by a single object.

Each device object is created by its own factory method. The collection of these methods is the board configuration, which itself is also a Singleton (the application runs on a single board). The Singleton property of the configuration is enforced by a class that contains only static methods. Figure 13 shows an example for such a class. The class IOSystem represents a system with three devices: a parallel port, as discussed before to interact with the environment, and two serial ports: one for program download and one which is an interface to a GPS receiver.

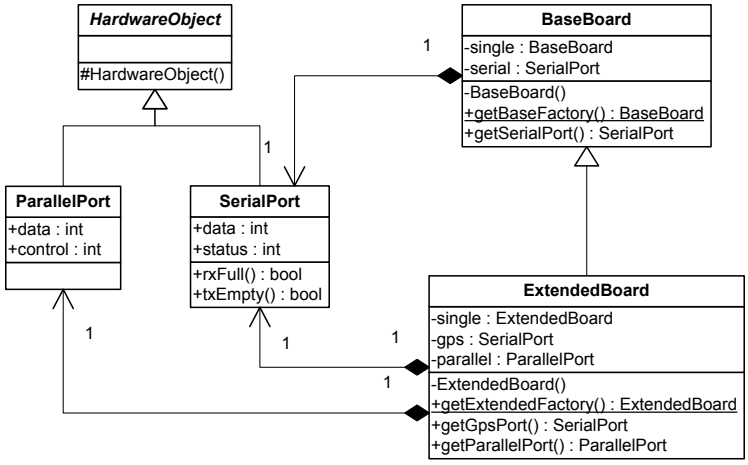


Fig. 15. Device object classes and board factory classes

This approach is simple, but not very flexible. Consider a vendor who provides boards in slightly different configurations (e.g., with different number of serial ports). With the above approach each board requires a different (or additional) IOSystem class that lists all devices. A more elegant solution is proposed in the next section.

3.3.2 *Board Configurations.* Replacing the static factory methods by instance methods avoids code duplication; inheritance then gives configurations. With a factory object we represent the common subset of I/O devices by a base class and the variants as subclasses.

However, the factory object itself must still be a Singleton. Therefore the board specific factory object is created at class initialization and is retrieved by a static method. Figure 14 shows an example of a base factory and a derived factory. Note how getBaseFactory() is used to get a single instance of the factory. We have applied the idea of a factory two times: the first factory generates an object that represents the board configuration. That object is itself a factory that generates the objects that interface to the hardware device.

The shown example base factory represents the minimum configuration with a single serial port for communication (mapped to System.in and System.out) represented by a SerialPort. The derived configuration ExtendedBoard contains an additional serial port for a GPS receiver and a parallel port for external control.

Furthermore, we show in Figure 14 a different way to incorporate the JVM mechanism in the factory: we define well known constants (the memory addresses of the devices) in the factory and let the native function jvmHWOCreate() return the correct device type.

Figure 15 gives a summary example of hardware object classes and the corresponding factory classes as an UML class diagram. The figure shows that all device classes subclass the abstract class HardwareObject. Figure 15 represents the simple abstraction as it is seen by the user of hardware objects.

3.3.3 *Interrupt Handler Registration.* We provide a base interrupt handling API that can be used both for non-RTSJ and RTSJ interrupt handling. The base class that is extended by an interrupt handler is shown in Figure 16. The handle() method contains the device server code. Interrupt control operations that have to be invoked before serving the device

```
abstract public class InterruptHandler implements Runnable {
    ...

    public InterruptHandler(int index) { ... };

    protected void startInterrupt() { ... };
    protected void endInterrupt() { ... };

    protected void disableInterrupt() { ... };
    protected void enableInterrupt() { ... };
    protected void disableLocalCPUInterrupts() { ... };
    protected void enableLocalCPUInterrupts() { ... };

    public void register() { ... };
    public void unregister() { ... };

    abstract public void handle() { ... };

    public void run() {
        startInterrupt();
        handle();
        endInterrupt();
    }
}
```

Fig. 16. Base class for the interrupt handlers

```
ih = new SerialInterruptHandler(); // logic of new BAEH

serialFirstLevelEvent = new AsyncEvent();
serialFirstLevelEvent.addHandler(
    new BoundAsyncEventHandler( null, null, null, null, null, false, ih )
    );

serialFirstLevelEvent.bindTo("INT4");
```

Fig. 17. Creation and registration of a RTSJ interrupt handler

(i.e. interrupt masking and acknowledging) and after serving the device (i.e. interrupt re-enabling) are hidden in the run() method of the base InterruptHandler, which is invoked when the interrupt occurs.

The base implementation of InterruptHandler also provides methods for enabling and disabling a particular interrupt or all local CPU interrupts and a special monitor implementation for synchronization between an interrupt handler thread and an application thread. Moreover, it provides methods for non-RTSJ registering and deregistering the handler with the hardware interrupt source.

Registration of a RTSJ interrupt handler requires more steps (see Figure 17). The InterruptHandler instance serves as the RTSJ logic for a (bound) asynchronous event handler, which is added as handler to an asynchronous event which then is bound to the interrupt source.

	Direct (no OS)	Indirect (OS)
Interpreted	SimpleRTJ	Kaffe VM
Native	JOP	OVM

Table II. Embedded Java Architectures

3.4 Perspective

An interesting topic is to define a common standard for hardware objects and interrupt handlers for different platforms. If different device types (hardware chips) that do not share a common register layout are used for a similar function, the hardware objects will be different. However, if the structure of the devices is similar, as it is the case for the serial port on the three different platforms used for the implementation (see Section 4), the driver code that *uses* the hardware object is identical.

If the same chip (e.g., the 8250 type and compatible 16x50 devices found in all PCs for the serial port) is used in different platforms, the hardware object and the device driver, which also implements the interrupt handler, can be shared. The hardware object, the interrupt handler, and the visible API of the factory classes are independent of the JVM and the OS. Only the *implementation* of the factory methods is JVM specific. Therefore, the JVM independent HAL can be used to start the development of drivers for a Java OS on any JVM that supports the proposed HAL.

3.5 Summary

Hardware objects are easy to use for a programmer, and the corresponding definitions are comparatively easy to define for a hardware designer or manufacturer. For a standardized HAL architecture we proposed factory patterns. As shown, interrupt handlers are easy to use for a programmer that knows the underlying hardware paradigm, and the definitions are comparatively easy to develop for a hardware designer or manufacturer, for instance using the patterns outlined in this section. Hardware objects and interrupt handler infrastructure have a few subtle implementation points which are discussed in the next section.

4. IMPLEMENTATION

We have implemented the core concepts on four different JVMs[7] to validate the proposed Java HAL. Table II classifies the four execution environments according to two important properties: (1) whether they run on bare metal or on top of an OS and (2) whether Java code is interpreted or executed natively. Thereby we cover the whole implementation spectrum with our four implementations. Even though the suggested Java HAL is intended for systems running on bare metal, we include systems running on top of an OS because most existing JVMs still require an OS, and in order for them to migrate incrementally to run directly on the hardware they can benefit from supporting a Java HAL.

In the direct implementation a JVM without an OS is extended with I/O functionality. The indirect implementation represents an abstraction mismatch – we actually re-map the concepts. Related to Figure 6 in the introduction, OVM and Kaffe represent configuration (a), SimpleRTJ configuration (b), and JOP configuration (c).

[7] On JOP the implementation of the Java HAL is already in use in production code.

```
public final class SerialPort extends HardwareObject {
    // LSR (Line Status Register)
    public volatile int status;
    // Data register
    public volatile int data;
    ...
}
```

Fig. 18. A simple hardware object

The SimpleRTJ JVM [RTJ Computing 2000] is a small, interpreting JVM that does not require an OS. JOP [Schoeberl 2005; 2008] is a Java processor executing Java bytecodes directly in hardware. Kaffe JVM [Wilkinson 1996] is a complete, full featured JVM supporting both interpretation and JIT compilation; in our experiments with Kaffe we have used interpretative execution only. The OVM JVM [Armbruster et al. 2007] is an execution environment for Java that supports compilation of Java bytecodes into the C language, and via a C compiler into native machine instructions for the target hardware. Hardware objects have also been implemented in the research JVM, CACAO [Krall and Grafl 1997; Schoeberl et al. 2008].

In the following we provide the different implementation approaches that are necessary for the very different JVMs. Implementing hardware objects was straightforward for most JVMs; it took about one day to implement them in JOP. In Kaffe, after familiarizing us with the structure of the JVM, it took about half a day of pair programming.

Interrupt handling in Java is straightforward in a JVM not running on top of an OS (JOP and SimpleRTJ). Kaffe and OVM both run under vanilla Linux or the real-time version Xenomai Linux [Xenomai developers 2008]. Both versions use a distinct user/kernel mode and it is not possible to register a user level method as interrupt handler. Therefore, we used threads at different levels to simulate the Java handler approach. The result is that the actual Java handler is the 3rd or even 4th level handler. This solution introduces quite a lot of overheads due to the many context switches. However, it is intended to provide a stepping stone to allow device drivers in Java; the goal is a real-time JVM that runs on the bare hardware.

In this section we provide more implementation details than usual to help other JVM developers to add a HAL to their JVM. The techniques used for the JVMs can probably not be used directly. However, the solutions (or sometimes work-arounds) presented here should give enough insight to guide other JVM developers.

4.1 SimpleRTJ

The SimpleRTJ JVM is a small, simple, and portable JVM. We have ported it to run on the bare metal of a small 16 bit microcontroller. We have successfully implemented the support for hardware objects in the SimpleRTJ JVM. For interrupt handling we use the ISR handler approach described in Section 3.2. Adding support for hardware objects was straightforward, but adding support for interrupt handling required more work.

4.1.1 *Hardware Objects.* Given an instance of a hardware object as shown in Figure 18 one must calculate the base address of the I/O port range, the offset to the actual I/O port, and the width of the port at runtime. We have chosen to store the base address of the I/O port range in a field in the common super-class for all hardware objects (HardwareObject).

```
SerialPort createSerialPort(int baseAddress ) {
    SerialPort sp = new SerialPort(baseAddress);
    return sp;
}
```

Fig. 19. Creating a simple hardware object

The hardware object factory passes the platform and device specific base address to the constructor when creating instances of hardware objects (see Figure 19).

In the put/getfield bytecodes the base address is retrieved from the object instance. The I/O port offset is calculated from the offset of the field being accessed: in the example in Figure 18 status has an offset of 0 whereas data has an offset of 4. The width of the field being accessed is the same as the width of the field type. Using these values the SimpleRTJ JVM is able to access the device register for either read or write.

4.1.2 *Interrupt Handler.* The SimpleRTJ JVM uses a simple stop-the-world garbage collection scheme. This means that within handlers, we prohibit use of the new keyword and writing references to the heap. These restrictions can be enforced at runtime by throwing a pre-allocated exception or at class loading by an analysis of the handler method. Additionally we have turned off the compaction phase of the GC to avoid the problems with moving objects mentioned in Section 3.2.4.

The SimpleRTJ JVM implements thread scheduling within the JVM. This means that it had to be refactored to allow for reentering the JVM from inside the first level interrupt handler. We got rid of all global state (all global variables) used by the JVM and instead allocate shared data on the C stack. For all parts of the JVM to still be able to access the shared data we pass around a single pointer to that data. In fact we start a new JVM for the interrupt handler with a temporary (small) Java heap and a temporary (small) Java stack. Currently we use 512 bytes for each of these items, which have proven sufficient for running non-trivial interrupt handlers so far.

The major part of the work was making the JVM reentrant. The effort will vary from one JVM implementation to another, but since global state is a bad idea in any case JVMs of high quality use very little global state. Using these changes we have experimented with handling the serial port receive interrupt.

4.2 JOP

JOP is a Java processor intended for hard real-time systems [Schoeberl 2005; 2008]. All architectural features have been carefully designed to be time-predictable with minimal impact on average case performance. We have implemented the proposed HAL in the JVM for JOP. No changes inside the JVM (the microcode in JOP) were necessary. Only the creation of the hardware objects needs a JOP specific factory.

4.2.1 *Hardware Objects.* In JOP, objects and arrays are referenced through an indirection called *handle*. This indirection is a lightweight read barrier for the compacting real-time GC [Schoeberl 2006; Schoeberl and Vitek 2007]. All handles for objects in the heap are located in a distinct memory region, the handle area. Besides the indirection to the *real* object the handle contains auxiliary data, such as a reference to the class information, the array length, and GC related data. Figure 20 shows an example with a small object that contains two fields and an integer array of length 4. The object and the array on the heap

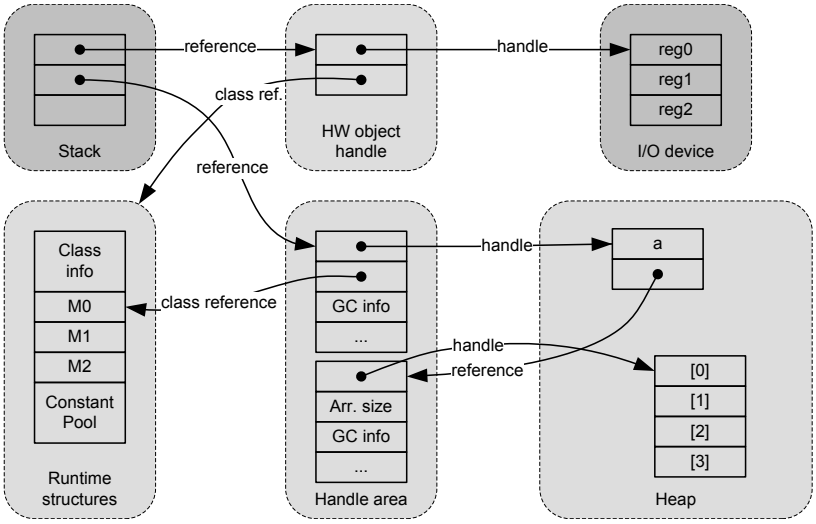


Fig. 20. Memory layout of the JOP JVM

just contain the data and no additional hidden fields. This object layout greatly simplifies our object to device mapping. We just need a handle where the indirection points to the memory mapped device registers instead of into the heap. This configuration is shown in the upper part of Figure 20. Note that we do not need the GC information for the hardware object handles. The factory, which creates the hardware objects, implements this indirection.

As described in Section 3.3.1 we do not allow applications to create hardware objects; the constructor is private (or package visible). Figure 21 shows part of the hardware object factory that creates the hardware object SerialPort. Two static fields (SP_PTR and SP_MTAB) are used to store the handle to the serial port object. The first field is initialized with the base address of the I/O device; the second field contains a pointer to the class information.[8] The address of the static field SP_PTR is returned as the reference to the serial port object.

The class reference for the hardware object is obtained by creating a *normal* instance of SerialPort with new on the heap and copying the pointer to the class information. To avoid using native methods in the factory class we delegate JVM internal work to a helper class in the JVM system package as shown in Figure 21. That helper method returns the address of the static field SP_PTR as reference to the hardware object. All methods in class Native, a JOP system class, are *native*[9] methods for low-level functions – the code we want to avoid in application code. Method toInt(Object o) defeats Java's type safety and returns a reference as an int. Method toObject(int addr) is the inverse function to map an address to a Java reference. Low-level memory access methods are used to manipulate the JVM data structures.

[8]In JOP's JVM the class reference is a pointer to the method table to speed-up the invoke instruction. Therefore, the name is XX_MTAB.

[9]There are no *real* native functions in JOP – bytecode is the native instruction set. The very few native methods in class Native are replaced by special, unused bytecodes during class linking.

```
package com.jopdesign.io;

public class BaseFactory {

    // static fields for the handle of the hardware object
    private static int SP_PTR;
    private static int SP_MTAB;

    private SerialPort sp;

    IOFactory() {
        sp = (SerialPort) makeHWObject(new SerialPort(), Const.IO_UART1_BASE, 0);
    };

    ...

    // That's the JOP version of the JVM mechanism
    private static Object makeHWObject(Object o, int address, int idx) {
        int cp = Native.rdIntMem(Const.RAM_CP);
        return JVMHelp.makeHWObject(o, address, idx, cp);
    }
}

package com.jopdesign.sys;

public class JVMHelp {

    public static Object makeHWObject(Object o, int address, int idx, int cp) {
        // usage of native methods is allowed here as
        // we are in the JVM system package
        int ref = Native.toInt(o);
        // fill in the handle in the two static fields
        // and return the address of the handle as a
        // Java object
        return Native.toObject(address);
    }
}
```

Fig. 21. Part of a factory and the helper method for the hardware object creation in the factory

To disallow the creation with new in normal application code, the visibility is set to package. However, the package visibility of the hardware object constructor is a minor issue. To access private static fields of an arbitrary class from the system class we had to change the runtime class information: we added a pointer to the first static primitive field of that class. As addresses of static fields get resolved at class linking, no such reference was needed so far.

4.2.2 *Interrupt Handler.* The original JOP [Schoeberl 2005; 2008] was a very puristic hard real-time processor. There existed only one interrupt – the programmable timer interrupt as time is the primary source for hard real-time events. All I/O requests were handled by periodic threads that polled for pending input data or free output buffers. During the course of this research we have added an interrupt controller to JOP and the necessary software layers.

```
static Runnable ih[] = new Runnable[Const.NUM_INTERRUPTS];
static SysDevice sys = IOFactory.getFactory().getSysDevice();

static void interrupt() {

    ih[sys.intNr].run();
}
```

Fig. 22. Interrupt dispatch with the static interrupt() method in the JVM helper class

Interrupts and exact exceptions are considered the hard part in the implementation of a processor pipeline [Hennessy and Patterson 2002]. The pipeline has to be drained and the complete processor state saved. In JOP there is a translation stage between Java bytecodes and the JOP internal microcode [Schoeberl 2008]. On a pending interrupt (or exception generated by the hardware) we use this translation stage to insert a special bytecode in the instruction stream. This approach keeps the interrupt completely transparent to the core pipeline. The special bytecode that is unused by the JVM specification [Lindholm and Yellin 1999] is handled in JOP as any other bytecode: execute microcode, invoke a special method from a helper class, or execute Java bytecode from JVM.java. In our implementation we invoke the special method interrupt() from a JVM helper class.

The implemented interrupt controller (IC) is priority based. The number of interrupt sources can be configured. Each interrupt can be triggered in software by a IC register write as well. There is one global interrupt enable and each interrupt line can be enabled or disabled locally. The interrupt is forwarded to the bytecode/microcode translation stage with the interrupt number. When accepted by this stage, the interrupt is acknowledged and the global enable flag cleared. This feature avoids immediate handling of an arriving higher priority interrupt during the first part of the handler. The interrupts have to be enabled again by the handler at a *convenient* time. All interrupts are mapped to the same special bytecode. Therefore, we perform the dispatch of the correct handler in Java. On an interrupt the static method interrupt() from a system internal class gets invoked. The method reads the interrupt number and performs the dispatch to the registered Runnable as illustrated in Figure 22. Note how a hardware object of type SysDevice is used to read the interrupt number.

The timer interrupt, used for the real-time scheduler, is located at index 0. The scheduler is just a plain interrupt handler that gets registered at mission start at index 0. At system startup, the table of Runnables is initialized with dummy handlers. The application code provides the handler via a class that implements Runnable and registers that class for an interrupt number. We reuse the factory presented in Section 3.3.1. Figure 23 shows a simple example of an interrupt handler implemented in Java.

For interrupts that should be handled by an event handler under the control of the scheduler, the following steps need to be performed on JOP:

(1) Create a SwEvent with the correct priority that performs the second level interrupt handler work

(2) Create a short first level interrupt handler as Runnable that invokes fire() of the corresponding software event handler

(3) Register the first level interrupt handler as shown in Figure 23 and start the real-time scheduler

```
public class InterruptHandler implements Runnable {

    public static void main(String[] args) {

        InterruptHandler ih = new InterruptHandler();
        IOFactory fact = IOFactory.getFactory();
        // register the handler
        fact.registerInterruptHandler(1, ih);
        // enable interrupt 1
        fact.enableInterrupt(1);
        .....
    }

    public void run() {
        System.out.println("Interrupt fired!");
    }
}
```

Fig. 23. An example Java interrupt handler as Runnable

In Section 5 we evaluate the different latencies of first and second level interrupt handlers on JOP.

4.3 Kaffe

Kaffe is an open-source[10] implementation of the JVM which makes it possible to add support for hardware objects and interrupt handlers. Kaffe requires a fully fledged OS such as Linux to compile and run. Although ports of Kaffe exist on uCLinux we have not been able to find a bare metal version of Kaffe. Thus even though we managed to add support of hardware objects and interrupt handling to Kaffe, it still cannot be used without an OS.

4.3.1 *Hardware Objects.* Hardware objects have been implemented in the same manner as in the SimpleRTJ, described in Section 4.1.

4.3.2 *Interrupt Handler.* Since Kaffe runs under Linux we cannot directly support the ISR handler approach. Instead we used the ISR event approach in which a thread blocks waiting for the interrupt to occur. It turned out that the main implementation effort was spent in the signaling of an interrupt occurrence from the kernel space to the user space.

We wrote a special Linux kernel module in the form of a character device. Through proper invocations of ioctl() it is possible to let the module install a handler for an interrupt (e.g. the serial interrupt, normally on IRQ 7). Then the Kaffe VM can make a blocking call to read() on the proper device. Finally the installed kernel handler will release the user space application from the blocked call when an interrupt occurs.

Using this strategy we have performed non-trivial experiments implementing a full interrupt handler for the serial interrupt in Java. Still, the elaborate setup requiring a special purpose kernel device is far from our ultimate goal of running a JVM on the bare metal. Nevertheless the experiment has given valuable experience with interrupt handlers and hardware objects at the Java language level.

[10] http://www.kaffe.org/

4.4 OVM

OVM [Armbruster et al. 2007] is a research JVM allowing many configurations; it is primarily targeted at implementing a large subset of RTSJ while maintaining reasonable performance. OVM uses ahead of time compilation via the C language: it translates both application and VM bytecodes to C, including all classes that might be later loaded dynamically at run-time. The C code is then compiled by GCC.

4.4.1 *Hardware Objects.* To compile Java bytecode into a C program, the OVM's Java-to-C compiler internally converts the bytecode into an intermediate representation (IR) which is similar to the bytecode, but includes more codes. Transformations at the IR level are both optimizations and operations necessary for correct execution, such as insertion of null-pointer checks. The produced IR is then translated into C, allowing the C compiler to perform additional optimizations. Transformations at the IR level, which is similar to the bytecode, are also typical in other JVM implementations, such as Sun's HotSpot.

We base our access to hardware objects on IR instruction transformations. We introduce two new instructions outb and inb for byte-wide access to I/O ports. Then we employ OVM's instruction rewriting framework to translate accesses to hardware object fields, putfield and getfield instructions, into sequences centered around outb and inb where appropriate. We did not implement word-wide or double-word wide access modes supported by a x86 CPU. We discuss how this could be done at the end of this section.

To minimize changes to the OVM code we keep the memory layout of hardware objects as if they were ordinary objects, and store port addresses into the fields representing the respective hardware I/O ports. Explained with the example from Figure 18, the instruction rewriting algorithm proceeds as follows: SerialPort is a subclass of HardwareObject; hence it is a hardware object, and thus accesses to all its public volatile int fields, status and data, are translated to port accesses to I/O addresses stored in those fields.

The translation (Figure 24) is very simple. In case of reads we append our new inb instruction after the corresponding getfield instruction in the IR: getfield will store the I/O address on the stack and inb will replace it by a value read from this I/O address. In case of writes we replace the corresponding putfield instruction by a sequence of swap, getfield, and outb. The swap rotates the two top elements on stack, leaving the hardware object reference on top of the stack and the value to store to the I/O port below it, The getfield replaces the object reference by the corresponding I/O address, and outb writes the value to the I/O port.

The critical part of hardware object creation is to set I/O addresses into hardware object fields. Our approach allows a method to turn off the special handling of hardware objects. In a hardware object factory method accesses to hardware object fields are handled as if they were fields of regular objects; we simply store I/O addresses to the fields.

A method can turn off the special handling of hardware objects with a marker exception mechanism which is a natural solution within OVM. The method declares to throw a PragmaNoHWIORegistersAccess exception. This exception is neither thrown nor caught, but the OVM IR level rewriter detects the declaration and disables rewriting accordingly. As the exception extends RuntimeException, it does not need to be declared in interfaces or in code calling factory methods. In Java 1.5, not supported by OVM, a standard substitute to the marker exception would be method annotation.

Our solution depends on the representation of byte-wide registers by 16-bit fields to hold

Reading from the device register serial.data, saving the result to the stack

original bytecode	*stack content*		*modified bytecode*	*stack content*
	{serial}			{serial}
GETFIELD data	{io port address}		GETFIELD data	{io port address}
		$\Longrightarrow$	INB	{inval}

Writing a value on the stack into the device register serial.data

original bytecode	*stack content*		*modified bytecode*	*stack content*
	{serial}, {outval}			{serial}, {outval}
PUTFIELD data	*empty*	$\Longrightarrow$	SWAP	{outval}, {serial}
		$\Longrightarrow$	GETFIELD data	{outval}, {io port address}
		$\Longrightarrow$	OUTB	*empty*

Fig. 24. Translation of bytecode for access to regular fields into bytecode for access to I/O port registers.

the I/O address. However, it could still be extended to support multiple-width accesses to I/O ports (byte, 16-bit, and 32-bit) as follows: 32-bit I/O registers are represented by Java long fields, 16-bit I/O registers by Java int fields, and byte-wide I/O registers by Java short fields. The correct access width will be chosen by the IR rewriter based on the field type.

4.4.2 *Interrupt Handler.* Low-level support depends heavily on scheduling and preemption. For our experiments we chose the uni-processor x86 OVM configuration with green threads running as a single Linux process. The green threads, delayed I/O operations, and handlers of asynchronous events, such as POSIX signals, are only scheduled at well-defined points (*pollchecks*) which are by default at back-branches at bytecode level and indirectly at Java-level blocking calls (I/O operations, synchronization calls, etc). When no thread is ready to run, the OVM scheduler waits for events using the POSIX select call.

As OS we use Xenomai RT Linux [Xenomai developers 2008; Gerum 2004]. Xenomai tasks, which are in fact user-space Linux threads, can run either in the Xenomai primary domain or in the Xenomai secondary domain. In the primary domain they are scheduled by the Xenomai scheduler, isolated from the Linux kernel. In the secondary domain Xenomai tasks behave as regular real-time Linux threads. Tasks can switch to the primary domain at any time, but are automatically switched back to the secondary domain whenever they invoke a Linux system call. A single Linux process can have threads of different types: regular Linux threads, Xenomai primary domain tasks, and Xenomai secondary domain tasks. Primary domain tasks can wait on hardware interrupts with a higher priority than the Linux kernel. The Xenomai API provides the interrupts using the ISR event handler approach and supports *virtualization* of basic interrupt operations – disabling and enabling a particular interrupt or all local CPU interrupts. These operations have the same semantics as real interrupts, and disabling/enabling a particular one leads to the corresponding operation being performed at the hardware level.

Before our extension, OVM ran as a single Linux process with a single (native Linux)

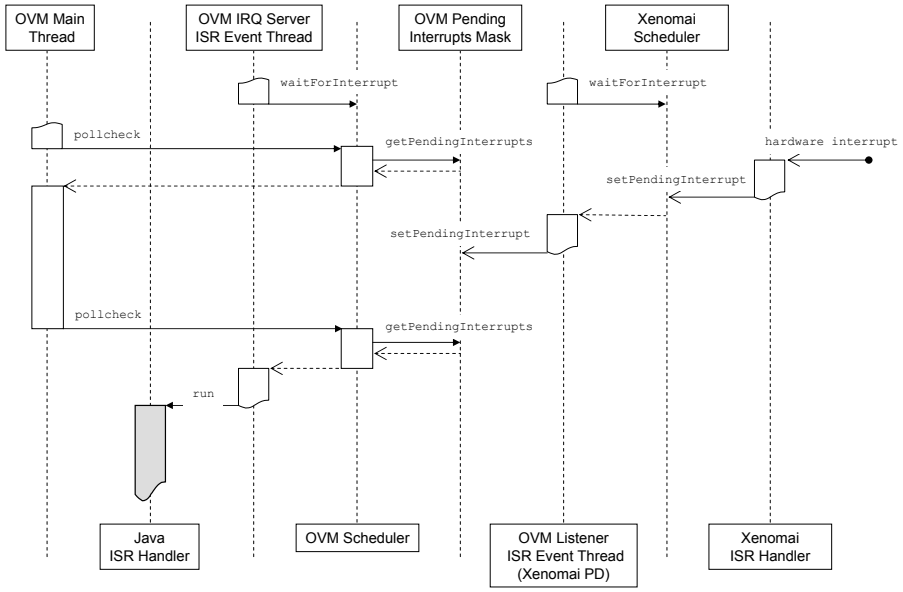


Fig. 25. Invocation of a Java interrupt handler under OVM/Xenomai.

thread, a *main OVM thread*. This native thread implemented Java green threads. To support interrupts we add additional threads to the OVM process: for each interrupt source handled in OVM we dynamically add an interrupt listener thread running in the Xenomai primary domain. The mechanism that leads to invocation of the Java interrupt handler thread is illustrated in Figure 25.

Upon receiving an interrupt, the listener thread marks the pending interrupt in a data structure shared with the main OVM thread. When it reaches a pollcheck, it discovers that an interrupt is pending. The scheduler then immediately wakes-up and schedules the Java green thread that is waiting for the interrupt (IRQ server thread in the figure). To simulate the first level ISR handler approach, this green thread invokes some handler method. In a non-RTSJ scenario the green thread invokes the run() method of the associated InterruptHandler (see Figure 16). In an RTSJ scenario (not shown in Figure 25), a specialized thread fires an asynchronous event bound to the particular interrupt source. It invokes the fire() method of the respective RTSJ's AsyncEvent. As mentioned in Section 3.3.3 the RTSJ logic of AsyncEventHandler (AEH) registered to this event should be an instance of InterruptHandler in order to allow the interrupt handling code to access basic interrupt handling operations.

As just explained, our first level InterruptHandlers virtualize the interrupt handling operations for interrupt enabling, disabling, etc. Therefore, we have two levels of interrupt virtualization, one is provided by Xenomai to our listener thread, and the other one, on top of the first one, is provided by the OVM runtime to the InterruptHandler instance. In particular, disabling/enabling of local CPU interrupts is emulated, hardware interrupts are disabled/enabled and interrupt completion is performed at the interrupt controller level (via

the Xenomai API), and interrupt start is emulated; it only tells the listener thread that the interrupt was received.

The RTSJ scheduling features (deadline checking, inter-arrival time checking, delaying of sporadic events) related to release of the AEH should not require any further adaptations for interrupt handling. We could not test these features as OVM does not implement them.

OVM uses *thin monitors* which means that a monitor is only instantiated (*inflated*) when a thread has to block on acquiring it. This semantic does not match to what we need – disable the interrupt when the monitor is acquired to prevent the handler from interrupting. Our solution provides a special implementation of a monitor for interrupt handlers and inflate it in the constructor of InterruptHandler. This way we do not have to modify the monitorenter and monitorexit instructions and we do not slow down regular thin monitors (non-interrupt based synchronization).

4.5 Summary

Support for hardware objects (see Section 3.1) and interrupt handling (see Section 3.2) to all four JVMs relies on common techniques. Accessing device registers through hardware objects extends the interpretation of the bytecodes putfield and getfield or redirects the pointer to the object. If these bytecodes are extended to identify the field being accessed as inside a hardware object, the implementation can use this information. Similarly, the implementation of interrupt handling requires changes to the bytecodes monitorenter and monitorexit or pre-inflating a specialized implementation of a Java monitor. In case of the bytecode extension, the extended codes specify if the monitor being acquired belongs to an interrupt handler object. If so, the implementation of the actual monitor acquisition must be changed to disable/enable interrupts. Whether dealing with hardware or interrupt objects, we used the same approach of letting the hardware object and interrupt handler classes inherit from the super classes HardwareObject and InterruptHandler respectively.

For JVMs that need a special treatment of bytecodes putfield and getfield (SimpleRTJ, Kaffe, and OVM) bytecode rewriting at runtime can be used to avoid the additional check of the object type. This is a standard approach (called *quick* bytecodes in the first JVM specification) in JVMs to speedup field access of resolved classes.

Historically, registers of most x86 I/O devices are mapped to a dedicated I/O address space, which is accessed using dedicated instructions – port read and port writes. Fortunately, both the processor and Linux allow user-space applications running with administrator privileges to use these instructions and access the ports directly via iopl, inb, and outb calls. For both the Kaffe and OVM implementations we have implemented bytecode instructions putfield and getfield accessing hardware object fields by calls to iopl, inb, and outb.

Linux does not allow user-space applications to handle hardware interrupts. Only kernel space functionality is allowed to register interrupt handlers. We have overcome this issue in two different ways:

—For Kaffe we have written a special purpose kernel module through which the user space application (the Kaffe VM) can register interest in interrupts and get notified about interrupt occurrence.

—For OVM we have used the Xenomai real-time extension to Linux. Xenomai extends the Linux kernel to allow for the creation of real-time threads and allows user space code to wait for interrupt occurrences.

Both these work-arounds allow an incremental transition of the JVMs and the related development libraries into a direct (bare metal) execution environment. In that case the work-arounds would no longer be needed.

If a compiling JVM is used (either as JIT or ahead-of-time) the compiler needs to be aware of the special treatment of hardware objects and monitors on interrupt handlers. One issue which we did not face in our implementations was the alignment of object fields. When device registers are represented by differently sized integer fields, the compiler needs to pack the data structure.

The restrictions within an interrupt handler are JVM dependent. If an interruptible, real-time GC is used (as in OVM and JOP) objects can be allocated in the handler and the object graph may be changed. For a JVM with a stop-the-world GC (SimpleRTJ and Kaffe) allocations are not allowed because the handler can interrupt the GC.

5. EVALUATION AND CONCLUSION

Having implemented the Java HAL on four different JVMs we evaluate it on a several test applications, including a tiny web server, and measure the performance of hardware accesses via hardware objects and the latency of Java interrupt handlers.

5.1 Qualitative Observations

For first tests we implemented a serial port driver with hardware objects and interrupt handlers. As the structure of the device registers is exactly the same on a PC, the platform for SimpleRTJ, and JOP, we were able to use the exact same definition of the hardware object SerialPort and the test programs on all four systems.

Using the serial device we run an embedded TCP/IP stack, implemented completely in Java, over a SLIP connection. The TCP/IP stack contains a tiny web server and we serve web pages with a Java only solution similar to the one shown in the introduction in Figure 6. The TCP/IP stack, the tiny web server, and the hardware object for the serial port are the same for all platforms. The only difference is in the hardware object creation with the platform dependent factory implementations. The web server uses hardware objects and polling to access the serial device.

5.1.1 *A Serial Driver in Java.* For testing the interrupt handling infrastructure in OVM we implemented a serial interrupt based driver in Java and a demo application that sends back the data received through a serial interface. The driver part of the application is a full-duplex driver with support for hardware flow control and with detection of various error states reported by the hardware. The driver uses two circular buffers, one for receiving and the other for sending. The user part of the driver implements blocking getChar and putChar calls, which have (short) critical sections protected by the interrupt-disabling monitor. To reduce latencies the getChar call sets the DSR flag to immediately allow receiving more data and the putChar, after putting the character into the sending buffer, initiates immediately the sending, if this is not currently being done already by the interrupt machinery. The driver supports serial ports with a FIFO buffer. The user part of the demo application implements the loop-back using getChar and putChar. The user part is a RTSJ AsyncEventHandler which is fired when a new character is received. From a Java perspective this is a 2nd level interrupt handler, invoked after the corresponding serial event is fired from the 1st level handler. To test the API described in the paper we implemented two versions that differ in how the first level handler is bound to the interrupt: (a) a RTSJ style version where

the first level handler is also a RTSJ event handler bound using bindTo to the JVM provided 1st level serial event, and (b) a non-RTSJ style version where the 1st level handler is registered using a InterruptHandler.register call. We have stress-tested the demo application and the underlying modified OVM infrastructure by sending large files to it through the serial interface and checked that they were returned intact.

5.1.2 *The HAL in Daily Use.* The original idea for hardware objects evolved during development of low-level software on the JOP platform. The abstraction with read and write functions and using constants to represent I/O addresses just *felt* wrong with Java. Currently hardware objects are used all over in different projects with JOP. Old code has been refactored to some extent, but new low-level code uses only hardware objects. By now low-level I/O is integrated into the language, e.g., auto completion in the Eclipse IDE makes it easy to access the factory methods and fields in the hardware object.

For experiments with an on-chip memory for thread-local scope caching [Wellings and Schoeberl 2009] in the context of a chip-multiprocessor version of JOP, the hardware array abstraction greatly simplified the task. The on-chip memory is mapped to a hardware array and the RTSJ based scoped memory uses it. Creation of an object within this special scope is implemented in Java and is safe because the array bounds checks are performed by the JVM.

5.1.3 *JNI vs Hardware Objects.* JNI provides a way to access the hardware without changing the code of the JVM. Nevertheless, with a lack of commonly agreed API, using it for each application would be redundant and error prone. It would also add dependencies to the application: hardware platform and the operating system (the C API for accessing the hardware is not standardized). The build process is complicated by adding C code to it as well. Moreover, the system needs to support shared libraries, which is not always the case for embedded operating systems (example is RTEMS, used by ESA).

In addition, JNI is typically too heavy-weight to implement trivial calls such as port or memory access efficiently (no GC interaction, no pointers, no threads interaction, no blocking). Even JVMs that implement JNI usually have some other internal light-weight native interface which is the natural choice for hardware access. This leads us back to a Java HAL as illustrated here.

5.1.4 *OVM Specific Experience.* Before the addition of hardware objects, OVM did not allow hardware access because it did not and does not have JNI or any other native interface for user Java code. OVM has a simplified native interface for the virtual machine code which indeed we used when implementing the hardware objects. This native interface can as well be used to modify OVM to implement user level access to hardware via regular method calls. We have done this to implement a benchmark to measure HWO/native overheads (later in this section). As far as simple port access is concerned, none of the solutions is strictly better from the point of the JVM: the bytecode manipulation to implement hardware objects was easy, as well as adding code to propagate native port I/O calls to user code. Thanks to ahead-of-time compilation and the simplicity of the native interface, the access overhead is the same.

The OVM compiler is fortunately not "too smart" so it does not get in the way of supporting hardware objects: if a field is declared volatile side-effects of reading of that field are not a problem for any part of the system.

The API for interrupt handling added to OVM allows full control over interrupts, typ-

	JOP		OVM		SimpleRTJ		Kaffe	
	read	write	read	write	read	write	read	write
native	5	6	5517	5393	2588	1123	11841	11511
HW Object	13	15	5506	5335	3956	3418	9571	9394

Table III. Access time to a device register in clock cycles

ically available only to the operating system. The serial port test application has shown that, at least for a simple device; it really allows us to write a driver. An interesting feature of this configuration is that OVM runs in user space and therefore it greatly simplifies development and debugging of Java-only device drivers for embedded platforms.

5.2 Performance

Our main objective for hardware objects is a clean object oriented interface to hardware devices. Performance of device register access is an important goal for relatively slow embedded processors; thus we focus on that in the following. It matters less on general purpose processors where the slow I/O bus essentially limits the access time.

5.2.1 *Measurement Methodology.* Execution time measurement of single instructions is only possible on simple in-order pipelines when a cycle counter is available. On a modern super-scalar architecture, where hundreds of instructions are in flight each clock cycle, direct execution time measurement becomes impossible. Therefore, we performed a bandwidth based measurement. We measure how many I/O instructions per second can be executed in a tight loop. The benchmark program is self-adapting and increases the loop count exponentially till the measurement run for more than one second and the iterations per second are reported. To compensate for the loop overhead we perform an overhead measurement of the loop and subtract that overhead from the I/O measurement. The I/O bandwidth b is obtained as follows:

$$b = \frac{cnt}{t_{test} - t_{ovhd}}$$

Figure 26 shows the measurement loop for the read operation in method test() and the overhead loop in method overhead(). In the comment above the method the bytecodes of the loop kernel is shown. We can see that the difference between the two loops is the single bytecode getfield that performs the read request.

5.2.2 *Execution Time.* In Table III we compare the access time with native functions to the access via hardware objects. The execution time is given in clock cycles. We scale the measured I/O bandwidth b with the clock frequency f of the system under test by $n = \frac{f}{b}$.

We have run the measurements on a 100 MHz version of JOP. As JOP is a simple pipeline, we can also measure short bytecode instruction sequences with the cycle counter. Those measurements provided the exact same values as the ones given by our benchmark, such that they validated our approach. On JOP the native access is faster than using hardware objects because a native access is a special bytecode and not a native function call. The special bytecode accesses memory directly where the bytecodes putfield and getfield perform a null pointer check and indirection through the handle for the field access. Despite the slower I/O access via hardware objects on JOP, the access is fast enough for all

```
public class HwoRead extends BenchMark {

    SysDevice sys = IOFactory.getFactory().getSysDevice();

/*  Bytecodes in the loop kernel
    ILOAD 3
    ILOAD 4
    IADD
    ALOAD 2
    GETFIELD com/jopdesign/io/SysDevice.uscntTimer : I
    IADD
    ISTORE 3
*/
    public int test(int cnt) {

        SysDevice s = sys;
        int a = 0;
        int b = 123;
        int i;

        for (i=0; i<cnt; ++i) {
            a = a+b+s.uscntTimer;
        }
        return a;
    }

/*  Bytecodes in the loop kernel
    ILOAD 3
    ILOAD 4
    IADD
    ILOAD 2
    IADD
    ISTORE 3
*/
    public int overhead(int cnt) {

        int xxx = 456;
        int a = 0;
        int b = 123;
        int i;

        for (i=0; i<cnt; ++i) {
            a = a+b+xxx;
        }
        return a;
    }
}
```

Fig. 26. Benchmark for the read operation measurement

currently available devices. Therefore, we will change all device drivers to use hardware objects.

The measurement for OVM was run on a Dell Precision 380 (Intel Pentium 4, 3.8 GHz, 3G RAM, 2M 8-way set associative L2 cache) with Linux (Ubuntu 7.10, Linux 2.6.24.3 with Xenomai-RT patch). OVM was compiled without Xenomai support and the generated virtual machine was compiled with all optimizations enabled. As I/O port we used the printer port. Access to the I/O port via a hardware object is just slightly faster than access via native methods. This was expected as the slow I/O bus dominates the access time.

On the SimpleRTJ JVM the native access is faster than access to hardware objects. The reason is that the JVM does not implement JNI, but has its own proprietary, more efficient, way to invoke native methods. It is done in a pre-linking phase where the invokestatic bytecode is instrumented with information to allow an immediate invocation of the target native function. On the other hand, using hardware objects needs a field lookup that is more time consuming than invoking a static method. With bytecode-level optimization at class load time it would be possible to avoid the expensive field lookup.

We measured the I/O performance with Kaffe on an Intel Core 2 Duo T7300, 2.00 GHz with Linux 2.6.24 (Fedora Core 8). We used access to the serial port for the measurement. On the interpreting Kaffe JVM we notice a difference between the native access and hardware object access. Hardware objects are around 20% faster.

5.2.3 *Summary.* For practical purposes the overhead on using hardware objects is insignificant. In some cases there may even be an improvement in performance. The benefits in terms of safe and structured code should make this a very attractive option for Java developers.

5.3 Interrupt Handler Latency

5.3.1 *Latency on JOP.* To measure interrupt latency on JOP we use a periodic thread and an interrupt handler. The periodic thread records the value of the cycle counter and triggers the interrupt. In the handler the counter is read again and the difference between the two is the measured interrupt latency. A plain interrupt handler as Runnable takes a constant 234 clock cycles (or 2.3 µs for a 100 MHz JOP system) between the interrupt occurrence and the execution of the first bytecode in the handler. This quite large time is the result of two method invocations for the interrupt handling: (1) invocation of the system method interrupt() and (2) invocation of the actual handler. For more time critical interrupts the handler code can be integrated in the system method. In that case the latency drops down to 0.78 µs. For very low latency interrupts the interrupt controller can be changed to emit different bytecodes depending on the interrupt number, then we avoid the dispatch in software and can implement the interrupt handler in microcode.

We have integrated the two-level interrupt handling at the application level. We set up two threads: one periodic thread, that triggers the interrupt, and a higher priority event thread that acts as second level interrupt handler and performs the handler work. The first level handler just invokes fire() for this second level handler and returns. The second level handler gets scheduled according to the priority. With this setup the interrupt handling latency is 33 µs. We verified this time by measuring the time between fire of the software event and the execution of the first instruction in the handler directly from the periodic thread. This took 29 µs and is the overhead due to the scheduler. The value is consistent with the measurements in [Schoeberl and Vitek 2007]. There we measured a minimum

	Median (µs)	3rd Quartile (µs)	95% Quantile (µs)	Maximum (µs)
Polling	3	3	3	8
Kernel	14	16	16	21
Hard	14	16	16	21
User	17	19	19	24
Ovm	59	59	61	203

Table IV. Interrupt (and polling) latencies in microseconds.

useful period of 50 µs for a high priority periodic task.

The runtime environment of JOP contains a concurrent real-time GC [Schoeberl and Vitek 2007]. The GC can be interrupted at a very fine granularity. During sections that are not preemptive (data structure manipulation for a new and write-barriers on a reference field write) interrupts are simply turned off. The copy of objects and arrays during the compaction phase can be interrupted by a thread or interrupt handler [Schoeberl and Puffitsch 2008]. Therefore, the maximum blocking time is in the atomic section of the thread scheduler and not in the GC.

5.3.2 *Latency on OVM/Xenomai.* For measuring OVM/Xenomai interrupt latencies, we have extended an existing interrupt latency benchmark, written by Jan Kiszka from the Xenomai team [Xenomai developers 2008]. The benchmark uses two machines connected over a serial line. The *log* machine, running a regular Linux kernel, toggles the RTS state of the serial line and measures the time it takes for the *target* machine to toggle it back.

To minimize measuring overhead the *log* machine uses only polling and disables local CPU interrupts while measuring. Individual measurements are stored in memory and dumped at shutdown so that they can be analyzed offline. We have made 400,000 measurements in each experiment, reporting only the last 100,000 (this was to warm-up the benchmark, including memory storage for the results). The *log* machine toggles the RTS state regularly with a given period.

We have tested 5 versions of the benchmark on the *target* machine: a polling version written in C (*polling*), a kernel-space interrupt handler in C/Xenomai running out of control of the Linux scheduler (*kernel*), a hard-realtime kernel-space interrupt handler running out of control of both the Xenomai scheduler and the Linux scheduler (*hard*), a user-space interrupt handler written in C/Xenomai (*user*), and finally an interrupt handler written in Java/OVM/Xenomai (*ovm*).

The results are shown in Table IV. The median latency is 3 µs for polling, 14 µs for both kernel-space handlers (hard and kernel), 17 µs for user-space C handler (user), and 59 µs for Java handler in OVM (ovm). Note that the table shows that the overhead of using interrupts over polling is larger than the overhead of handling interrupts in user-space over kernel-space. The maximum latency of OVM was 203 µs, due to infrequent pauses. Their frequency is so low that the measured 95% quantile is only 61 µs.

The experiment was run on Dell Precision 380 (Intel Pentium 4 3.8 GHz, 3G RAM, 2M 8-way set associative L2 cache) with Linux (Ubuntu 7.10, Linux 2.6.24.3 with Xenomai-RT patch). As Xenomai is still under active development we had to use Xenomai workarounds and bugfixes, mostly provided by Xenomai developers, to make OVM on Xenomai work.

5.3.3 *Summary.* The overhead for implementing interrupt handlers is very acceptable since interrupts are used to signal relatively infrequently occurring events like end of transmission, loss of carrier etc. With a reasonable work division between first level and second level handlers, the proposal does not introduce dramatic blocking terms in a real-time schedulability analysis, and thus it is suitable for embedded systems.

5.4 Discussion

5.4.1 *Safety Aspects.* Hardware objects map object fields to the device registers. When the class that represents a device is correct, access to it is safe – it is not possible to read from or write to an arbitrary memory address. A memory area represented by an array is protected by Java's array bounds check.

5.4.2 *Portability.* It is obvious that hardware objects are platform dependent; after all the idea is to have an interface to the bare metal. Nevertheless, hardware objects give device manufacturers an opportunity to supply supporting factory implementations that fit into Java's object-oriented framework and thus cater for developers of embedded software. If the same device is used on different platforms, the hardware object is portable. Therefore, standard hardware objects can evolve.

5.4.3 *Compatibility with the RTSJ Standard.* As shown for the OVM implementation, the proposed HAL is compatible with the RTSJ standard. We consider it to be a very important point since many existing systems have been developed using such platforms or subsets thereof. In further development of such applications existing and future interfacing to devices may be refactored using the proposed HAL. It will make the code safer and more structured and may assist in possible ports to new platforms.

5.5 Perspective

The many examples in the text show that we achieved a representation of the hardware close to being platform independent. Also, they show that it is possible to implement system level functionality in Java. As future work we consider to add devices drivers for common devices such as network interfaces[11] and hard disc controllers. On top of these drivers we will implement a file system and other typical OS related services towards our final goal of a Java only system.

An interesting question is whether a common set of *standard* hardware objects is definable. The SerialPort was a lucky example. Although the internals of the JVMs and the hardware were different one compatible hardware object worked on all platforms. It should be feasible that a chip manufacturer provides, beside the data sheet that describes the registers, a Java class for the register definitions of that chip. This definition can be reused in all systems that use that chip, independent of the JVM or OS.

Another interesting idea is to define the interaction between the GC and hardware objects. We stated that the GC should not collect hardware objects. If we relax this restriction we can redefine the semantics of collecting an object: on running the finalizer for a hardware object the device can be put into sleep mode.

[11] A device driver for a CS8900 based network chip is already part of the Java TCP/IP stack.

ACKNOWLEDGMENTS

We wish to thank Andy Wellings for his insightful comments on an earlier version of the paper. We also thank the reviewers for their detailed comments that helped to enhance the original submission. The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 216682 (JEOPARD).

REFERENCES

aJILE. 2000. aj-100 real-time low power Java processor. preliminary data sheet.

ARMBRUSTER, A., BAKER, J., CUNEI, A., FLACK, C., HOLMES, D., PIZLO, F., PLA, E., PROCHAZKA, M., AND VITEK, J. 2007. A real-time Java virtual machine with applications in avionics. *Trans. on Embedded Computing Sys. 7,* 1, 1–49.

BACON, D. F., CHENG, P., AND RAJAN, V. T. 2003. A real-time garbage collector with low overhead and consistent utilization. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, NY, USA, 285–298.

BOLLELLA, G., GOSLING, J., BROSGOL, B., DIBBLE, P., FURR, S., AND TURNBULL, M. 2000. *The Real-Time Specification for Java*. Java Series. Addison-Wesley.

BURNS, A. AND WELLINGS, A. J. 2001. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*, 3rd ed. Addison-Wesley Longman Publishing Co., Inc.

CASKA, J. accessed 2009. micro [μ] virtual-machine. Available at `http://muvium.com/`.

CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. 2001. An empirical study of operating systems errors. *SIGOPS Oper. Syst. Rev. 35,* 5, 73–88.

FELSER, M., GOLM, M., WAWERSICH, C., AND KLEINÖDER, J. 2002. The JX operating system. In *Proceedings of the USENIX Annual Technical Conference*. 45–58.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. M. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional.

GAY, D., LEVIS, P., VON BEHREN, R., WELSH, M., BREWER, E., AND CULLER, D. 2003. The nesC language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 1–11.

GERUM, P. 2004. Xenomai - implementing a RTOS emulation framework on GNU/Linux. `http://www.xenomai.org/documentation/branches/v2.4.x/pdf/xenomai.pdf`.

GROUP, T. C. 2008. Trusted computing. Available at `https://www.trustedcomputinggroup.org/`.

HANSEN, P. B. 1977. *The Architecture of Concurrent Programs*. Prentice-Hall Series in Automatic Computing. Prentice-Hall.

HENNESSY, J. AND PATTERSON, D. 2002. *Computer Architecture: A Quantitative Approach, 3rd ed.* Morgan Kaufmann Publishers Inc., Palo Alto, CA 94303.

HENTIES, T., HUNT, J. J., LOCKE, D., NILSEN, K., SCHOEBERL, M., AND VITEK, J. 2009. Java for safety-critical applications. In *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*.

HILL, J., SZEWCZYK, R., WOO, A., HOLLAR, S., CULLER, D. E., AND PISTER, K. S. J. 2000. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX), ACM SIGPLAN*. ACM, Cambridge, MA, 93–104. Published as Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX), ACM SIGPLAN, volume 35, number 11.

HUNT, G., LARUS, J. R., ABADI, M., AIKEN, M., BARHAM, P., FAHNDRICH, M., HAWBLITZEL, C., HODSON, O., LEVI, S., MURPHY, N., STEENSGAARD, B., TARDITI, D., WOBBER, T., AND ZILL, B. D. 2005. An overview of the singularity project. Tech. Rep. MSR-TR-2005-135, Microsoft Research (MSR). Oct.

KORSHOLM, S., SCHOEBERL, M., AND RAVN, A. P. 2008. Interrupt handlers in Java. In *Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008)*. IEEE Computer Society, Orlando, Florida, USA.

KRALL, A. AND GRAFL, R. 1997. CACAO – A 64 bit JavaVM just-in-time compiler. In *PPoPP'97 Workshop on Java for Science and Engineering Computation*, G. C. Fox and W. Li, Eds. ACM, Las Vegas.

KREUZINGER, J., BRINKSCHULTE, U., PFEFFER, M., UHRIG, S., AND UNGERER, T. 2003. Real-time event-handling and scheduling on a multithreaded Java microcontroller. *Microprocessors and Microsystems 27,* 1, 19–31.

LINDHOLM, T. AND YELLIN, F. 1999. *The Java Virtual Machine Specification*, Second ed. Addison-Wesley, Reading, MA, USA.

LOHMEIER, S. 2005. Jini on the Jnode Java os. Online article at `http://monochromata.de/jnodejini.html`.

PHIPPS, G. 1999. Comparing observed bug and productivity rates for java and c++. *Softw. Pract. Exper. 29,* 4, 345–358.

RAVN, A. P. 1980. Device monitors. *IEEE Transactions on Software Engineering 6,* 1 (Jan.), 49–53.

RTJ COMPUTING. 2000. simpleRTJ a small footprint Java VM for embedded and consumer devices. Available at `http://www.rtjcom.com/`.

SCHOEBERL, M. 2005. Jop: A java optimized processor for embedded real-time systems. Ph.D. thesis, Vienna University of Technology.

SCHOEBERL, M. 2006. Real-time garbage collection for Java. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*. IEEE, Gyeongju, Korea, 424–432.

SCHOEBERL, M. 2008. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture 54/1–2*, 265–286.

SCHOEBERL, M., KORSHOLM, S., THALINGER, C., AND RAVN, A. P. 2008. Hardware objects for Java. In *Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008)*. IEEE Computer Society, Orlando, Florida, USA.

SCHOEBERL, M. AND PUFFITSCH, W. 2008. Non-blocking object copy for real-time garbage collection. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)*. ACM Press.

SCHOEBERL, M. AND VITEK, J. 2007. Garbage collection for safety critical Java. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*. ACM Press, Vienna, Austria, 85–93.

SHA, L., RAJKUMAR, R., AND LEHOCZKY, J. P. 1990. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput. 39,* 9, 1175–1185.

SIEBERT, F. 2002. *Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages*. Number ISBN: 3-8311-3893-1. aicas Books.

SIMON, D., CIFUENTES, C., CLEAL, D., DANIELS, J., AND WHITE, D. 2006. Java on the bare metal of wireless sensor devices: the squawk Java virtual machine. In *Proceedings of the 2nd international conference on Virtual execution environments (VEE 2006)*. ACM Press, New York, NY, USA, 78–88.

WELLINGS, A. AND SCHOEBERL, M. 2009. Thread-local scope caching for real-time Java. In *Proceedings of the 12th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2009)*. IEEE Computer Society, Tokyo, Japan.

WILKINSON, T. 1996. Kaffe – a virtual machine to run java code. Available at http://www.kaffe.org.

WIRTH, N. 1977. Design and implementation of modula. *Software - Practice and Experience 7*, 3–84.

WIRTH, N. 1982. *Programming in Modula-2*. Springer Verlag.

XENOMAI DEVELOPERS. 2008. Xenomai: Real-time framework for Linux. `http://www.xenomai.org`.

Received August 2008; revised April 2009 and July 2009; accepted September 2009

Flash memory in embedded Java programs

Stephan Korsholm
VIA University College
Horsens, Denmark
sek@viauc.dk

ABSTRACT

This paper introduces a Java execution environment with the capability for storing constant heap data in Flash, thus saving valuable RAM. The extension is motivated by the structure of three industrial applications which demonstrate the need for storing constant data in Flash on small embedded devices.

The paper introduces the concept of *host initialization of constant data* to prepare a Flash image of constant data that can be kept outside the heap during runtime.

The concept is implemented in an interpreter based Java execution environment.

Categories and Subject Descriptors

B.3.1 [**Memory Structures**]: Read-only memory (ROM); C.3 [**Special-purpose and Application-based Systems**]: Real-time and embedded systems; D.3.4 [**Processors**]: Interpreters

General Terms

Languages, Design.

Keywords

Java/C Integration, Flash Memory, Embedded Systems, Real-Time.

1. INTRODUCTION

The C programming language together with appropriate runtime libraries is widely used when implementing software for tiny devices. C environments support a range of features such as

1. accessing device registers and raw memory,
2. handling first level interrupts,
3. storing constant data in Flash.

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.
JTRES 2011 September 26-28, 2011, York, UK
Copyright 2011 ACM 978-1-4503-0731-4I/11/09 ...$10.00.

These features are often controlled by compiler directives that instruct the compiler to generate code for e.g. (1) accessing hardware through memory mapped device registers (2) marking C-functions as interrupt handlers and generating proper save and restore code stubs and (3) marking a piece of data as constant and placing it in a read-only data segment.

Previous work in [15] and [12] has demonstrated how (1) and (2) can be added to Java execution environments. This paper describes a method to support (3) for Java by marking parts or all of an object as constant and eligible for storing in Flash, thereby freeing valuable RAM storage.

The method presented here shows how to store constant data (e.g. in the form of large arrays) in Flash instead of the heap. The idea is to pre-execute the initial part of the program on a host platform and in that part of the program build the arrays and other data that are considered constant and thus can be kept in Flash. After the pre-execution on the host the constant data is converted into an image that can be kept in Flash on the target. When running on the target the Flash image is accessible for reading by the application while writing to the Flash image is implemented as null operations.

The methods used in the process have been used in popular environments like C (see Section 3) and Ada (see Section 7) for many years in order to solve the same problem. The contribution of this paper is to apply these methods to the Java domain. By doing this we have solved problems from industrial settings. Problems which until now have prevented us from using Java on a certain class of embedded targets. We describe these applications further in Section 1.1.

We consider our proposal a starting point to bridge an important gap many embedded developers have to cross if they consider using a high level, well structured language like Java in the embedded domain. The method can be improved in several ways and we discuss these in Section 8.

1.1 Constant Data

Constant data are data that are initialized at application load time and do not change during the entire lifetime of the application. To find out how much constant data a given C application is using, the unstripped binary file can be inspected. We have looked at three industrial embedded C applications and examined their use of constant data. The example applications are

1. The DECT protocol stack from Polycom. It is used by Polycom, Horsens, DK [14] for implementing wireless communication between devices. It is used on the

CR16 16bit platform from National with 8KB RAM and 768KB Flash

2. The Modbus control and monitoring application from Grundfos. It is used by Grundfos, Bjerringbro, DK [9] for monitoring and controlling pumps. It is used on the NEC v850 32bit platform with 4KB and upwards of RAM and 64-1024KB Flash

3. The HVM Java virtual machine [11]. It is used by the author for programming tiny embedded devices in Java. It is used on the ATMega 8bit platform from AVR with 8KB RAM and 256KB Flash

Table 1 divides the used data memory into two types: variable data and constant data. Variable data may be written during the execution of the program and have to be kept in RAM, whereas constant data are only written once during initialization and can be kept in ROM. For reference, the size of the code itself is included in Table 1 as well.

	Code	Variable Data	Constant Data
DECT	234KB	7KB (8%)	78KB (92%)
Modbus	171KB	207KB (83%)	41KB (17%)
HVM	22KB	4KB (29%)	10KB (71%)

Table 1: Overview of data usage

All three applications use a significant amount of constant data. For the DECT and HVM applications it would not be possible to hold the constant data in RAM together with the variable data as that would require more RAM than is available on the devices.

Further scrutinizing the source code of the applications above reveals the following examples of use of constant data

- DECT. Arrays of bytes used as input to the DSP (Digital Signal Processor) to generate ringing tones

- Modbus. Large tables of approx. 512 entries each. The tables map names to data locations called registers. The indirection through the tables are used to control access to the registers - some tables allow full access to all registers while others allow limited access to only some registers. Section 6 looks into this example in more detail

- HVM. A selection of the class file content is initialized as a graph-like structure for easy access to important information when the code is interpreted

It seems reasonable to expect that other applications for similar types of platforms use constant data to a significant degree as well and it follows that being able to store constant data in Flash is desirable when implementing similar applications in Java for similar types of platforms.

1.2 Why Flash?

Producers of embedded solutions seek to use Flash as opposed to RAM whenever possible. The reason is that the production price per unit is significantly lower by using Flash as much as possible instead of adding more RAM to the design. The Polycom and Grundfos units are produced and sold in large quantities, and even minor savings in unit production cost have a great impact on the profitability of their businesses.

The remainder of this paper will describe a design for using Flash for constant data in Java programs and show an implementation in an interpreter based execution environment.

2. RUNNING EMBEDDED JAVA ON TINY DEVICES

We will use the HVM [11] to illustrate how constant data can be integrated into a Java execution environment. The HVM is a small interpreter based JVM intended for use on tiny devices. An architectural overview of the HVM Java execution environment is shown in Figure 1.

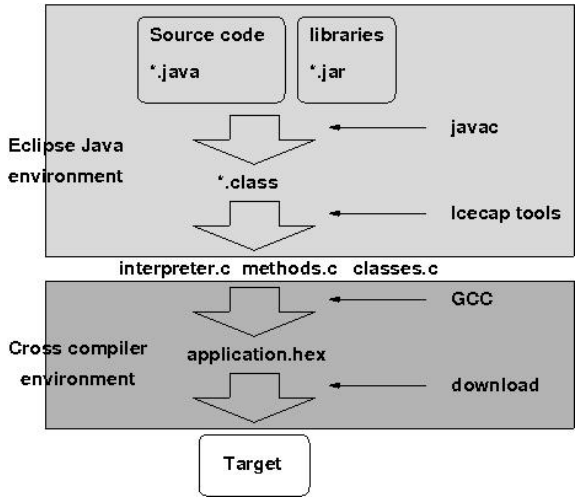


Figure 1: HVM Architecture

The Eclipse IDE integrated with any Java compiler and SDK is used to compile the Java source code into class files.

Next a plugin tool, called icecap [11], analyzes the generated class files and convert them into C code by storing the Java byte codes and other class file content in byte arrays in two auto-generated C files - *methods.c* and *classes.c*.

Finally the developer can use *any C based cross compiler environment* (e.g. the avr-gcc tool chain for the AVR family of processors) or the commercially available IAR compiler tool-chain (for a wide variety of industry standard embedded platforms). The C cross compiler compiles the HVM interpreter itself (*interpreter.c*) and the auto-generated C files into a final executable for the target. A non-cross compiler can be used as well to generate an executable for the host platform (e.g. Linux or Windows) to run the executable in the host environment for testing or debugging.

3. A DESIGN FOR FLASH DATA IN JAVA

Consider the following C example of constant data taken from the DECT application

```
const unsigned char wav_num_0[] =
    { 23, 112, -1, -1 };
```

This is a simplified version of a longer array used by the DSP to generate ringing tones. When the C compiler translates this into assembler it will look like the following

```
.globl wav_num_0
.section .rodata
.type wav_num_0, @object
.size wav_num_0, 4
wav_num_0:
.byte 23
.byte 112
.byte -1
.byte -1
```

Based on this assembler output from the compiler, the linker will create an array of bytes in the section *rodata* which means that the array will eventually be packaged along side the code in Flash. Additionally it will link the use of the variable names to the location of the data in Flash. We notice that preparing the constant data for Flash is done on the host, before the application is started, since it is difficult and usually avoided to write to Flash during runtime. It is only when the application is loaded, that the boot-loader will be able to write code and constant data to the Flash. When control is handed over from the boot-loader to the application, the Flash memory becomes read-only.

In Java the example above looks like this

```
public class ConstantData
{
    final byte wav_num_0[] =
        { 23, 112, -1, -1 };
}
```

And now the assembler output (the Java byte codes) from the compiler is

```
public class ConstantData extends Object{
public ConstantData();
  Code:
   0:  aload_0
   1:  invokespecial #1; //"<init>":()V
   4:  aload_0
   5:  iconstant_4
   6:  newarray byte
   8:  dup
   9:  iconstant_0
   10: bipush 23
   12: bastore
   13: dup
   14: iconstant_1
   15: bipush 112
   17: bastore
   18: dup
   19: iconstant_2
   20: iconstant_m1
   21: bastore
   22: dup
   23: iconstant_3
   24: iconstant_m1
   25: bastore
   26: putfield #2; //wav_num_0:[B
   29: return
}
```

The main difference from the C solution is that the code to set up the constant data is executed on the target during runtime, where as the C solution prepares a read-only section of data on the host. In Java the *wav_num_0* array will be a normal array placed in the heap (in RAM) into which the bytes are written at runtime.

The goal of our design for constant data in Java is to place the wav_num_0 array in Flash rather than on the heap.

3.1 Identifying Constant Data

Byte codes to create and access data in general must know which type of data is accessed, mainly because constant data are not located in the heap (RAM) but located in Flash. To facilitate this, constant data must be *inferred implicitly or marked explicitly.*

For simplicity we will mark constant data at the time of declaration. In the examples below we use the *const* keyword to mark constant data. Since this is not a Java keyword any actual implementation could use Java annotations when marking constant data. Below follows an example of marking a piece of data as constant

```
public class ConstantData
{
  private const byte wav_num_0[] =
    { 23, 112, -1, -1 };

  public static void main(String args[])
  {
    ConstantData cdata = new ConstantData();
  }
}
```

3.2 Creating Constant Data - Host Initialization

Creating and initializing constant data must be done before the application is downloaded and run on the target - this is because writing to Flash at runtime is very difficult and inefficient. Similar to what C environments do, the constant data segments must be built and linked with the final executable on the host. To solve this challenge we suggest to use *host initialization of constant data.*

The idea is to run the initialization part of the application on the JVM on the host platform. While running on the host platform, the soon-to-be read-only segment of constant data can be built in RAM on the host. The start of the initialization phase will be the start of the program. The end of the initialization phase could either be inferred implicitly or marked explicitly. For simplicity we have chosen the latter, as in the following example

```
public class ConstantData
{
  private const byte wav_num_0[] =
    { 23, 112, -1, -1 };

  private const byte wav_num_1[] =
    { 43, 12, -1, -1 };

  private const int NUM = 42;

  public boolean flag;

  public static void main(String args[])
  {
    ConstantData cdata = new ConstantData();
    System.lockROM();
    cdata.flag = true;
  }
}
```

The call to *System.lockROM* signals the end of the initialization phase. During the initialization phase three new objects are created

1. *cdata* is an object with three constant data fields and one non-constant field. We allocate 8 bytes for the two array references and 4 bytes for the int *NUM* - these 12 bytes plus some bookkeeping are allocated in the constant heap. The non constant part of the object is one byte field of size 1 byte. This is allocated in the normal heap. For the HVM the bookkeeping information is a class index and a reference into the normal heap pointing to the non-constant part of the object

2. *wav_num_0* is a constant array of 4 bytes. Since it is marked as *const*, it is not allocated in the standard heap, but in the constant heap. The array uses 4 bytes plus some bookkeeping information. For the HVM an additional 2 bytes for the element count and 2 bytes for the type are needed. Thus 8 bytes are allocated in the constant heap

3. Similarly *wav_num_1* is allocated in the constant heap right after the previous item

When the *System.lockROM()* method is called on the host, the content of the host memory is as depicted in Figure 2.

3.3 Creating Constant Data - Target Initialization

After the host initialization phase, the content of the constant heap is transferred into a C array of bytes and saved in a file *rom.c*. Using the cross compiler environment, the interpreter in *interpreter.c*, the Java application in *classes.c* and *methods.c*, and the constant data Flash image in *rom.c* are linked to form the final executable. The overall architecture is depicted in Figure 3.

So, the initialization part will be run twice - once on the host to produce the constant heap, and once on the target when the executable is downloaded and run. On the target it will repeat the initialization step - it will execute the same code, allocate objects and read/write to them in the same sequence,but *the way the allocation and access are handled is different on the target*:

- When *allocating data* in the constant heap, the same allocation procedures can be carried out, but the constant heap is already initialized and placed in Flash. References to the existing objects are returned to the caller.

- When *writing* to the constant heap no actual write takes place since the value is already there.

- When *reading* from the constant heap a Flash read operation is carried out.

In order for this to work, the initialization phase must be deterministic - when executed on the target, objects must be allocated and written in the exact same order as they were on the host. Additionally the allocation of constant data must only take place during the initialization phase.

3.4 Preparing the Constant Data Heap

The heap of constant data has to be useful on both the host environment during initialization and later on the target environment when the application is running. The values in the constant heap can be viewed as a graph of data, where

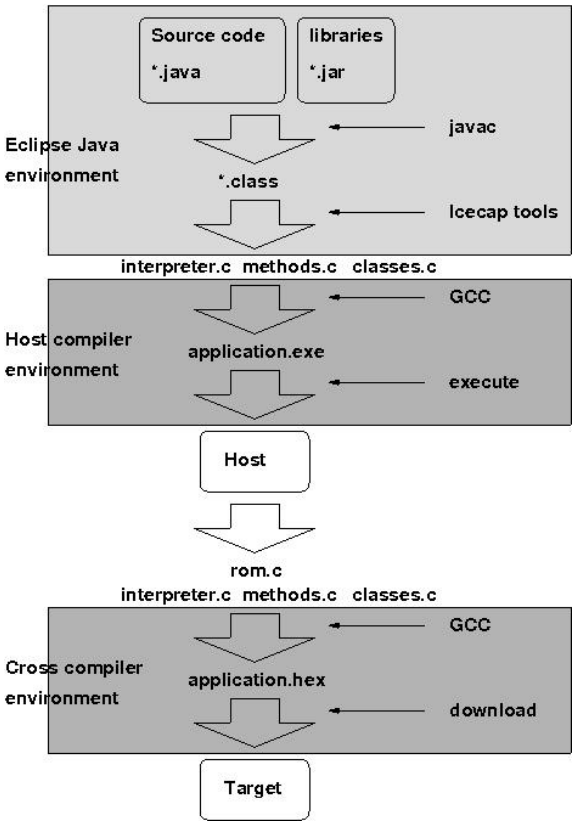


Figure 3: Creating the ROM image

the nodes (objects and arrays) contain basic data like integers and the edge between the nodes are references. This graph is transferred from the host to the target. The consequence of this is

- the edges cannot be actual pointers (addresses), as the addresses are not going to be the same on the host as on the target

- the values in the nodes have to be in the same byte order

Transferring the constant heap is basically a serialization of the data into a platform independent format. In the HVM all references in the constant heap are stored as offsets, either into the constant heap itself, or offsets into the normal heap and all values are stored in little endian format. This incurs an overhead for accessing data in the constant heap, but it does not affect how data are accessed in the normal heap.

3.5 Discussion

By using host initialization of constant data it is now possible to create the Flash image and link it with the final executable - in exactly the same manner as is done in C environments. Reading and writing to the constant heap on the host are done in the same manner as reading and writing to the normal heap, as both heaps are kept in RAM. On the target, accessing Flash from program code is different from

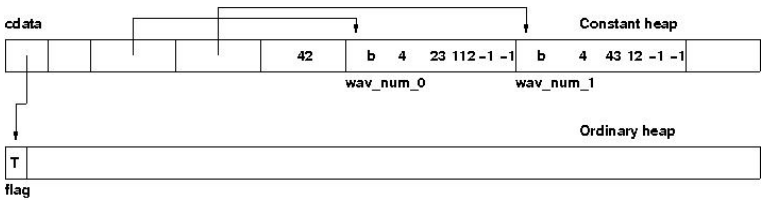


Figure 2: Host memory after initialization

accessing RAM memory. The hardware architecture of embedded devices can be divided into either a Von Neumann architecture or a Harvard Architecture [16]. In the former, both code and data are in the same address space, and instructions for reading and writing to Flash are the same as instructions for reading and writing to RAM. In the latter, architecture code and data are kept in two separate data spaces and accessed using different instructions.

In both cases it is easy and fast reading from Flash at runtime, but it is difficult writing to Flash at runtime.

To support constant data in Java on any embedded architecture, the VM needs to know if it is accessing Flash or normal RAM so that it may use the proper instructions for doing so.

In Section 4 we will show what is required to augment an existing JVM with capabilities for identifying and accessing constant data.

4. IMPLEMENTING CONSTANT DATA IN A VM

All creation and access of constant data take place through byte codes. Constant data can be either arrays or field variables. The byte codes involved in accessing arrays or object fields must be augmented with capabilities to recognize and access constant data as well. The cost of this in terms of execution efficiency is discussed in Section 5.

The byte codes that may be directly involved in accessing constant data are *new*, *newarray*, the array store byte codes - call these *xastore*, the array load byte codes - call these *xaload*, and finally the object field accessors *getfield* and *putfield*. For simplicity we defer the handling of static constant data, but this is a simple extension to the current implementation. In the following we describe the changes that are required to the involved byte codes in order to handle constant data - first on the host, then on the target.

4.1 Host Handling of Constant Data

new contains a reference to the class that is being instantiated. If one of the fields in the class is declared as constant data, then the object is allocated differently than from normal heap objects. Whether the class does indeed contain constant data can either be embedded in the byte code, or looked up at runtime. The HVM checks at runtime if the class contains constant data, and if so the new byte code will allocate two parts of data, one in the constant heap and one in the normal heap as described in Section 3.2.

newarray does not contain enough information to know if the array is constant or not. The array is constant if it is later saved in a constant field, but this is not known when the byte code is executed. To solve this problem, the icecap tools (see Figure 1) perform a static analysis of the code to instrument the *newarray* byte code with a flag indicating if a constant array is being allocated or not.

xastore, xaload retrieve the array to store/load from the stack. On the host it makes no difference to these byte codes if the array is in the constant heap or not.

putfield, getfield As with *new* these byte codes have enough information to decide if the field being written is inside a constant data object or not, and if the field itself is a constant field. This information can either be looked up at runtime or embedded in the byte code. The HVM looks up the information at runtime to properly access the field in either the constant heap or the normal heap.

4.2 Target Handling of Constant Data

new, newarray On the target the *new* byte code should not actually allocate and clear a new piece of memory for constant objects. Rather, if the object being allocated contains constant data, it should return a reference to the data as it is already present in the Flash section. If the allocations of constant objects are proceeding in the same order and with the same sizes as during the initialization phase on the host, the references returned will be correct. Still the non constant part of the object must be allocated in the normal heap.

xastore Whether the array is a constant array or not can be determined by looking at the reference - if it points into the constant heap, it is a constant array. Since we prefer to avoid writing to the Flash on the target, the array store operation can instead check if the value being written is the same as the value already present in Flash.

xaload On Harvard architecture platforms the loading from Flash must be done using a special load instruction - e.g. on the ATMega2560 the instruction to load from Flash is called *lpm* (Load Program Memory) where as the instruction to load from RAM is called *ld*. On many Von Neumann architectures no changes are required to this byte code as loading from Flash is no different than loading from RAM (as is the case with the CR16 platform).

putfield On the target this instruction cannot actually write to constant fields, but should behave in the same manner as the array store instruction. If the target field of the write is a non constant field inside a constant object, then an extra indirection is required.

getfield On the target this instruction works as the array load instructions, so it is necessary to use special load instructions for accessing Flash, but more important, the loading of a non constant field from an object with constant fields requires an extra indirection.

4.3 Checking Constant Data Access

After the initialization phase the program should never write to constant data again. If a new value is written to a constant array after the initialization phase, the interpreter would discover that an illegal write is taking place. In the current implementation on the HVM this will result in a runtime exception.

If constant data access conventions are violated, a static check would be better than introducing new runtime checks as discussed in Section 8.

5. COST OF CONSTANT DATA

The goal of any implementation of constant data for Java should be

1. If a program does not use constant data, its runtime efficiency should not be affected by the option for doing so. Likewise, the resulting size of the executable and its RAM requirements at runtime should not be affected.

2. If a program does use constant data, only the size of constant data or the efficiency by which constant data can be accessed must be affected. In other words - reading and writing to objects or arrays that do not contain any constant data should not be less efficient because of the presence of constant data elsewhere.

To meet these requirements it must be possible to identify and substitute involved byte codes with special purpose byte codes that know about constant data.

In the absence of dynamic class loading, it is statically decidable for the *new*, *putfield*, and *getfield* byte codes if they will be used for objects containing constant data. Creating objects and accessing fields do not involve virtuality in Java - a field cannot be overwritten in a subclass, it can be over shadowed, but it is statically decidable for each *new*, *putfield*, and *getfield* byte code if it will involve constant data fields or not.

For the byte codes used for array creation and access, it is not in general possible to decide statically if they will work on constant arrays or not. Consider the situation where a constant field holding an array is being initialized by calling some virtual method that returns the array. In that case it is generally unknown which *newarray* byte code is being used for creating the array. In the HVM we restrict ourselves to handle array creations as in the following example,

```
public class ConstantData
{
    final byte wav_num_0[] =
        { 23, 112, -1, -1 };
}
```

The code for this can be analyzed and the byte codes annotated so that special purpose byte codes can be used at runtime for the creation and initialization of constant arrays. Even so when reading from an array on Harvard Architecture targets the *xaload* byte code must always check if it is reading from an array in the Flash - this will violate both (1) and (2) above. Currently this problem has not been solved in the HVM implementation.

Other byte codes that are affected by the presence of constant data are the *arraylength*, *instanceof* and *checkcast* byte codes, the first for the same reason as with *xaload*, the latter two because they need to access the object to retrieve its class which again on Harvard Architecture machines must be done differently from accessing normal RAM - hence an extra check is needed in the general case.

To sum up: the cost of supporting constant data in the HVM has been an extra check introduced in the *arraylength*, *instanceof*, *checkcast* and *xaload* byte codes. In Section 6.1 we conclude a worst case cost of approx. 7% increase in running time.

5.1 Impact on GC

Introducing an extra heap (the constant heap) naturally affects the garbage collector (GC). The HVM uses a reference counting GC and to support constant data the following changes have to be made,

- Collecting constant data. Constant data are by nature never garbage. They may become unreachable but they can never be collected since they are placed in Flash which is read-only. The HVM uses 6 bits for the reference count and if the reference count becomes 0x3f the GC will never consider the object garbage. Objects in the constant data heap are given an initial reference count of 0x3f

- Updating reference counts. During the initialization phase it is not needed to update reference counts to objects in the constant heap - they will be marked as immortal anyway, but objects in the normal heap that are referred from the constant heap will have their reference count updated as normal

If a GC strategy uses a marking phase, the constant heap may contain references into the ordinary heap. Such references are additional roots to the marking phase. This set of additional roots never changes after the initialization phase. It seems like a reasonable idea to disable GC during the initialization phase and then after the initialization phase add the constant set of additional roots to the full set of GC roots. Then any mark-sweep collector may safely ignore the presence of constant data.

6. THE MODBUS APPLICATION REWRITTEN FOR JAVA

In the following we will look at the Modbus application from Grundfos. Below is a Java version of some Modbus functionality found in the existing C implementation. This functionality accounts for a large part of the use of constant data in that application. The functionality in the form of constant arrays marks which of a set of 512 registers may be accessed and which may not be accessed. These tables are always consulted before acessing the registers, and depending on the situation, references to different lookup tables are used. If a register may not be accessed, it is marked as *NOACCESS*, otherwise its index into the register array is given. E.g. when a device is configured for the first time by a user with administrator priveleges, it should be possible to access all registers. In this case the *configuratorRegisterMap* is used. When running in the field the software should no longer be able to change everything but only certain parts of the configuration. In that case the *fieldUserRegisterMap*

is used. The register array itself (not shown here) is kept in RAM, but the lookup tables are constant data and can be kept in Flash.

```
class ConstantData {
  public const int[] fieldUserRegisterMap = {
    REG1,
    REG2,
    NOACCESS,
    REG4,
    REG5,
    NOACCESS,
    NOACCESS,
    NOACCESS,
    REG9,
    ...,
    REG512
  };

  public const int[] configuratorRegisterMap = {
    REG1,
    REG2,
    REG3,
    REG4,
    REG5,
    REG6,
    REG7,
    REG8,
    REG9,
    ...,
    REG512
  };
```

The Modbus application contains 7 such lookup tables taking up approx. 3.5 KB of constant memory. In the 4KB version of the target these tables alone would occupy most of the available RAM if translated into Java. Only after adding support for constant data in Flash, is the HVM able to run the code from the example above. Currently, parts of the Modbus application is being ported to Java to gain more experience from using constant data in Flash and from using Java on the Grundfos platforms in general.

6.1 Measuring Cost

When adding a new feature to a Java execution environment, it is reasonable to expect, that programs actually using the feature will have to pay a cost for that in terms of execution time. On the other hand, programs that do not use the feature, should not have to pay for it. In the following we present the result of performing three measurements while executing the same program on the HVM

1. Executing the program while all data are kept and accessed in RAM on a HVM build that does not support constant data in Flash
2. Executing the program while all data are still kept and accessed in RAM but this time on a HVM build that does indeed support constant data in Flash
3. Finally, executing the program while all data are kept and accessed in Flash on a HVM build that does support constant data in Flash

The program we used creates and scans each element in the two lookup tables from the Modbus example described in the previous section. The program was executed on the ATMega2560 and we measured execution time by counting clock cycles.

We would expect a rise in execution time of 3) compared to 1). This corresponds to paying a cost for actually using the feature. But we would like to see as little rise as possible in execution time of 2) as this corresponds to paying a cost for a feature that is not being used.

The results from measuring the cost of adding support for constant data in Flash on the HVM is listed in Table 2.

As can be seen we pay a cost of approximately 7% in scenario 2) above. We consider this a worst case scenario for the following reasons: As described in Section 5 it is difficult to statically decide if array creation and array access op-codes may be used for accessing Flash data, thus they must be able to handle the general case where Flash data may be present. On top of that, array load and array store operations may be executed many times compared to other byte codes, if the scanning of arrays is involved. The test program will execute the byte code *newarray* twice and the byte codes *arraylength*, *iastore* and *iaload* many times corresponding to the length of the arrays.

That there is a cost in actually using the feature and accessing objects in Flash, in 3) above, is not surprising, but of course the implementation should seek to get this cost as low as possible.

6.2 Discussion

The HVM is an interpreter and as such users of it are aware of a degradation in execution efficiency compared to AOT or JIT compiled environments. Such users may be willing to accept an additional degradation of 7.5%. Still, it seems a high price to pay for supporting constant data in Flash. Furthermore the HVM is not optimized and future optimizations of the HVM bringing down the execution time of all byte codes will most likely increase the percentage wise cost of supporting Flash data. A possible option to alleviate this problem is to extend on the static analysis to make it better at deciding which array access byte codes may be used to access Flash and substitute these byte codes for their Flash aware counterparts during compile time. While this is in general an undecidable problem, data flow analysis may in practice be able to bring down the average cost.

7. RELATED WORK

The *final* modifier in the Java language ensures that a variable is not changed after its initialization. For variables of basic type, e.g. *integer*, this works fine but for references it only ensures that the reference itself is not changed - it is still possible to change the target of the reference.

Several suggestions have been made to add “read-only” qualifiers to Java to ensure that both arrays and objects are not changed if marked read-only (e.g. [4] and [2]). The focus for their research is to statically enforce rules about proper use of data and thus be able to catch errors earlier. The focus of this paper is to be able to minimize use of RAM by using Flash instead. These two efforts are synergetic in the sense that any successful suggestion for marking and statically enforcing read-only data in Java would have as a direct side effect that the compiler could generate byte codes to place and access data in Flash.

The suggestion presented in this paper to mark read-only data at the point of declaration does not ensure or enforce that read-only data are indeed read-only, but if the programmer respects the read-only declaration, then we have shown

	Clock Cycles	Cost in %
(1) Without Flash support	179827	0%
(2) With Flash support, not using it	193415	7.5%
(3) With Flash support, using it	240258	33.5%

Table 2: Overview of data usage

how that may carry over to save that data in Flash.

The HVM is similar in its design to the TakaTuka VM [1] and the Darjeeling VM [6]. All three VMs are Flash aware and store byte codes and other runtime structures in Flash. To further reduce RAM use at runtime, this paper describes how runtime items like arrays and objects that are considered read-only can be stored in Flash as well. Since the TakaTuka and Darjeeling VMs are Flash aware interpreter based environments, it is most likely possible to include the design for constant data presented here in those environments as well.

The RTSJ [3] supports accessing physical memory through the physical memory classes and the raw memory access classes. This means that the addressed storage could be of any memory type (RAM, ROM, Flash, etc.). E.g. through the use of the method

```
void getLongs(long off, long[] arr, ...)
```

on the class *RawMemoryAccess* it is possible to read data from Flash and store it in the heap in the *arr* array. As long as this array is in use it will take up heap space. With the design presented in this paper, the array can be referred to directly at its Flash location.

Dividing the execution of a program into several phases is known from e.g. SCJ [17]. A SCJ application is a sequence of mission executions. A mission embodies the life cycle of a safety-critical application. Each mission is comprised of initialization, execution, and cleanup phases. The proper execution of the application is enforced through the user-defined implementation of the *Safelet* interface. The suggestion in this paper for dividing the execution into an initialization phase and a following runtime phase could be greatly improved by adopting similar designs.

The idea of doing a partial execution of the initialization phase is used in other scenarios. An interesting example is the V8 JavaScript Engine used in Google chrome. In order to speed up the instantiation of any additional JavaScript context, the initial setup of built-in objects is done only once to produce what they call a *snapshot*: ".. a snapshot includes a serialized heap which contains already compiled code for the built-in JavaScript code" [8]. The V8 concept of a "snapshot" is very similar to the partially evaluated constant heap described in Section 3.2.

Finally, like C, Ada also supports 'ROM-able objects' [7]. The support stems from the more general concept of 'pre-elaboration', introduced so that real-time programs can start up more quickly. In Ada a certain class of static expressions can be evaluated at compile-time, just like a certain part of expressions in C can. Even though this is introduced to speed up start-up time, the Ada documentation also states that pre-elaborated objects are ROM-able.

8. FUTURE WORK

Explicitly marking constant data (see Section 3.1) and explicitly marking the end of the initialization phase (see Section 3.2) is a reasonable starting point, but this suggestion can be error prone - runtime errors may occur if constant data are marked incorrectly. This invites us to add tool support for identifying constant data automatically and for identifying the end of the initialization phase automatically.

Constant data can informally be described as "data that is written to once, but read from several times". The initialization phase is an execution path that starts from the beginning of the program and until the last constant data have been initialized. Can a static analysis of the application find the maximal subset of the heap that has the "write-once/read-many" propertiy? Can the code point that corrosponds to the transition from the initialization phase to the mission phase be found? If that execution point and subset of data can be found through a static analysis, then the programmer would not have to mark constant heap data, nor have to mark the end of the initialization phase.

The effort described here to use Flash for constant heap data is a small part of a larger effort to make Java an attractive alternative to C for embedded software developers. The overall objective is divided into two parts

1. Make popular features, supported by well known C environments, available to Java developers as well. Prominent examples of such features are (1) device register access (2) 1st level interrupt handling and (3) constant heap data located in Flash.
2. Add tool support to improve on supported features when compared to their support in C environments.

In relation to (1) above, significant improvements have already been made. The support for Hardware Objects [15], 1st level interrupt handling [12] and now constant heap data in Flash may be a prerequisite for many to contemplate making the transition from C environments to Java environments. For example, adding tool support for identifying constant data and the related initialization phase will make the transition from C to Java even more attractive.

The focus of the HVM has not been execution efficiency, but rather to prove that features known from C environments (hardware objects, interrupt handling and constant heap data) - features considered very important by many embedded developers - can indeed be implemented in a Java execution environment. Other Java environments for embedded systems have achieved impressive execution efficiency, e.g.

- Interpretation. The JamVM [10] is considered one of the most efficient interpreters with execution times approximately three times slower than compiled C.
- JIT compilation. The Cacao Java JIT compiler [5] is a highly efficient JIT compiler for embedded systems.
- AOT compilation. The Fiji VM [13], which is an AOT compilation based execution environment for embedded systems delivers "comparable performance to that

of server-class production Java Virtual Machine implementations"

These efficient environments for the embedded target could add support for our features described above, thus taking away an important obstacle for engineers within the embedded industry to consider Java for their next generation development of embedded software.

9. CONCLUSION

This paper motivates why it is important to be able to keep read-only heap data in Flash when executing Java on small embedded devices. Three industrial applications demonstrate the use of constant data in existing C programs - two of these applications will not be able to run in Java if constant data cannot be placed and accessed in Flash. These examples justify the claim that the number of applications that could potentially be implemented in Java on small embedded devices will be significantly greater if constant data in Flash is supported.

Engineers are used to being able to place constant data in Flash in the C environments they use. This paper presents a design for constant data in Java - and an implementation in an interpreter based environment. Engineers will in this respect be able to do in Java what they are used to doing in C. But they will be able to do more than that: the host initialization phase can initialize more complicated structures than is possible in C. As long as data are written just once and the procedure for doing so is deterministic, all usual Java constructs may be used during initialization. In C only a very limited subset of the language can be used to initialize constant data. Thus the design presented here not only adds similar functionality to Java environments, but extends that functionality as well.

10. ACKNOWLEDGEMENT

Thank you to Niels Jørgen Strøm from Grundfos, who stated the original requirement that constant heap data had to be kept in flash, and who formulated the original ideas that formed the basis for our work. Thank you to Dion Nielsen from Polycom for allowing us to use their software. A warm thanks to A.P. Ravn for sharing his great insight and valuable comments on the first drafts of the paper. Also a warm thank you to Hans Søndergaard for reading and commenting on this paper again and again.

Finally, we are grateful for the instructive comments by the reviewers which have helped us clarify several points in this paper.

11. REFERENCES

[1] F. Aslam, C. Schindelhauer, G. Ernst, D. Spyra, J. Meyer, and M. Zalloom. Introducing takatuka: a java virtualmachine for motes. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, SenSys '08, pages 399–400, New York, NY, USA, 2008. ACM.

[2] A. Birka and M. D. Ernst. A practical type system and language for reference immutability. *SIGPLAN Not.*, 39:35–49, October 2004.

[3] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. The real-time specification for java 1.0.2. Available at: http://www.rtsj.org/specjavadoc/book_index.html.

[4] J. Boyland. Why we should not add readonly to java (yet). In *In FTfJP*, pages 5–29, 2005.

[5] F. Brandner, T. Thorn, and M. Schoeberl. Embedded jit compilation with cacao on yari. In *Proceedings of the 2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, ISORC '09, pages 63–70, Washington, DC, USA, 2009. IEEE Computer Society.

[6] N. Brouwers, K. Langendoen, and P. Corke. Darjeeling, a feature-rich vm for the resource poor. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 169–182, New York, NY, USA, 2009. ACM.

[7] T. B. et. al. Ada 9x project report. Revision request report, pages 3.2–3.4, The US Department of Defence, August 1989.

[8] Google. V8 javascript engine, embedder's guide. `http://code.google.com/apis/v8/embed.html`, 2011.

[9] Grundfos. http://www.grundfos.com/. Visited June 2011.

[10] jamvm. http://jamvm.sourceforge.net/. Visited June 2011.

[11] S. Korsholm. Hvm lean java for small devices. `http://www.icelab.dk/`, 2011.

[12] S. Korsholm, M. Schoeberl, and A. P. Ravn. Interrupt handlers in java. In *Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 453–457, Washington, DC, USA, 2008. IEEE Computer Society.

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

[14] Polycom. http://www.polycom.dk/. Visited June 2011.

[15] M. Schoeberl, S. Korsholm, C. Thalinger, and A. P. Ravn. Hardware objects for java. In *In Proceedings of the 11th IEEE International Symposium on Object/component/serviceoriented Real-time distributed Computing (ISORC 2008*. IEEE Computer Society, 2008.

[16] A. S. Tanenbaum and J. R. Goodman. *Structured Computer Organization, fifth edition*, pages 18, 80. Prentice Hall, Upper Saddle River, NJ, USA, 2010.

[17] TheOpenGroup. Safety-critical java technology specification (jsr-302). Draft Version 0.79, TheOpenGroup, May 2011.

Safety-Critical Java for Low-End Embedded Platforms

Hans Søndergaard, Stephan E. Korsholm
VIA University College
Horsens, Denmark
{hso,sek}@viauc.dk

Anders P. Ravn
Department of Computer Science
Aalborg University, Denmark
apr@cs.aau.dk

ABSTRACT

We present an implementation of the Safety-Critical Java profile (SCJ), targeted for low-end embedded platforms with as little as 16 kB RAM and 256 kB flash. The distinctive features of the implementation are a combination of a lean Java virtual machine (HVM), with a bare metal kernel implementing hardware objects, first level interrupt handlers, and native variables, and an infrastructure written in Java which is minimized through program specialization. The HVM allows the implementation to be easily ported to embedded platforms which have a C compiler as part of the development environment; the bare metal approach eliminates the need for a resource consuming operating system or C-library; the program specialization means that the infrastructure for the SCJ profile is optimized for a particular application keeping only the code and data the application needs. The SCJ implementation is evaluated with a known benchmark and shown to reduce this to a size where it can execute on a minimal configuration.

Categories and Subject Descriptors

C.3 [**Special Purpose and application-based systems**]: Real-time and embedded systems; D.3.3 [**Programming Languages**]: Language Constructs and Features

Keywords

Embedded Systems, Real-Time Java, Virtual Machine, Safety-Critical Java

1. INTRODUCTION

Recently efforts have been made to enable the Java language for embedded systems, and environments such as FijiVM [17], JamaicaVM [25] and PERC [14] show that Java can be executed efficiently on high-end embedded devices, thus allowing embedded software engineers to reap the benefits from using Java, tools and methods. The benefits that desktop and server developers have had for some time. Java environments for high-end embedded devices are even more powerful than their C counterparts: explicit memory management is replaced by automatic memory management through real-time garbage collection, and threads and synchronization are supported by APIs such as the RTSJ [7]. Additionally a significant amount of tools exist for performing static program analysis of embedded Java programs both for checking resource consumption, analysing potential runtime errors, and for specializing programs so they become more ressouce efficient.

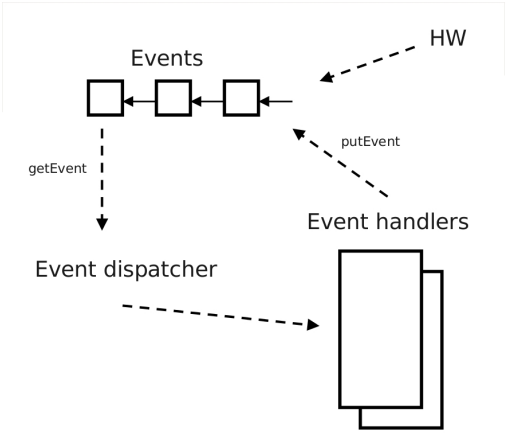


Figure 1: Event Driven Scheduling

For low-end embedded systems, with limited memory and computational resources, and usually without a POSIX-like OS, the evidence that Java can substitute or support the use of C is not as strong, but environments such as KESO [10], PERC Pico [3], and Muvium [8] are paving the way.

To further increase the usability of Java for low-end embedded systems we present a SCJ [29] implementation that runs on systems with a few kBs of RAM and less than 256 kB of ROM. This implementation is, to our knowledge, the smallest SCJ implementation available in terms of footprint.

Low-end platforms

In order to clarify the kind of platform we have in mind, we give the example of a *KIRK DECT Application Module* [20]

from Polycom [19], also called the KT4585. This module is used to wirelessly transmit voice and data using the DECT protocol. It has the following features:

- 40 Mhz, 16 bit RISC architecture
- 2 x 8 kB RAM, 768 kB ROM
- Apart from the main processor (a CR16c) the KT4585 module features processors to do voice encoding and for controlling a radio receiver/transmitter

Polycom uses a C based framework for programming the main processor of the KT4585. It uses a simple event-driven programming model. As observed in [11, 9], event-driven programming is a popular model for writing small embedded systems. Figure 1 illustrates the model: Program logic is implemented by event handlers that may activate other handlers by sending events to them. A simple event is just a signal, but it may also carry a limited amount of data with it. A dispatcher loop will retrieve a next event from a list, find the receiving handler, and call it with the event as actual parameter. Hardware Interrupts (HW) may generate events through interrupt service routines. Periodic processes are implemented by a clock driver having a list of handlers to be signalled at appropriate times.

In our experience, the use of such *home-grown*, event-driven, simple RTOS implementations is a popular choice, although this is not an easy mechanism to analyze for hard real-time properties. Another common attribute of embedded environments, such as the KT4585, is that since memory is limited, a common solution to memory management is to allocate most memory during application start up and only to a limited extent use dynamic memory management.

These observations suggest that Java, together with the SCJ profile, can fulfill the requirements of environments such as the KT4585, both in terms of scheduling mechanisms and memory management. Environments such as KESO and Muvium already show that Java in itself can be enabled for such small devices, and it is our hypothesis that the SCJ profile as well can be made to run at this level and fit well with known architectural concepts.

The SCJ-HVM framework

In order to execute Java programs on low-end platforms we need: a resource efficient Java virtual machine, a minimal hardware interface layer, and a somewhat more elaborate infrastructure that implements the SCJ profile. The architecture of the implementation is shown in Figure 2.

The virtual machine is the HVM and Section 2 gives an overview of the HVM showing how it meets our requirements for resource efficiency.

The minimal hardware interface layer is specified by `VMInterface`. According to Figure 2 this interface is divided into three parts:

- Memory allocation can be controlled through the class `AllocationArea`. Using facilities of this class, the SCJ infrastructure can control where allocations done by e.g. the `new` and `newArray` instructions of the HVM are made. The current allocation area can be exchanged with another to eventually implement SCJ memory semantics
- Process scheduling is done through an interface to the CPU that defines a minimal `Process` object[1], with methods for initialization and an associated static `ProcessSequencer` class that implements a context switch
- Finally the interface to a Clock specifies methods to get the granularity and the absolute time; for efficiency reasons it has a `delayUntil` method.

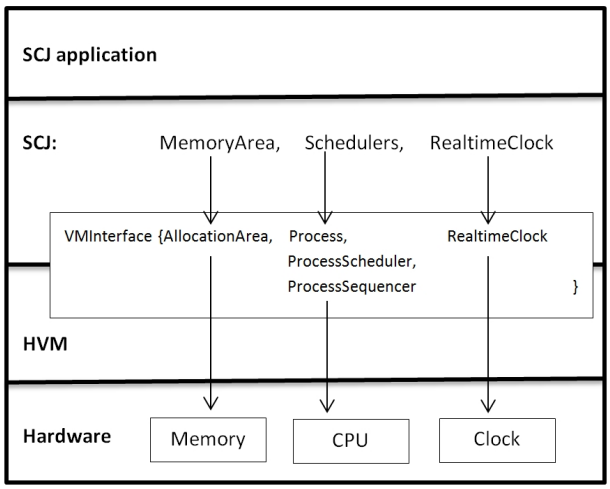


Figure 2: SCJ architecture with the VMInterface to HVM.

Further details of the implementation including the use of the novel feature of *native variables* are in Section 2.

The SCJ infrastructure is discussed in detail in Section 3. It introduces the main concepts of the profile and then focuses on implementation of the infrastructure at the Java level of nested scoped memories and scheduling of the handlers that implement program logic. In particular, we show implementations of a static cyclic executive as well as a fixed priority preemptive scheduler. Thus all parts of the SCJ, including the schedulers, are implemented in pure Java.

A crucial part of the framework does not show up. It is the *program specialization* that works together with the HVM. It is an intelligent class linking, where a static analysis of the Java source base is performed. It computes a conservative estimate of the set of classes and methods that may be executed in a run of the entire program. Only this set is translated into C and included in the final executable. This can for instance eliminate a cyclic executive if missions only use fixed priority preemptive scheduling or vice versa. The implications are shown in Section 4 that evaluates the framework using the refactored CDx benchmark [12], called

[1]So named in recognition of the Modula 2 ancestry [31]

miniCDj [18]. The result is that this application fits on the very small platform mentioned in the beginning.

In summary, the contribution of this paper is a SCJ implementation for low-end embedded platforms with as little as 16 kB RAM and 256 kB flash. This is achieved by combining:

- The HVM virtual machine
- Hardware near features like native variables, hardware objects, and 1st level interrupt handlers, allowing for an implementation almost without a native layer
- Program specialization to eliminate superfluous infrastructure

Yet, this is just one step forward. There are still interesting questions of making the ensemble more space and execution time efficient, perhaps by generalizing some interactions on the interface. This is discussed in the concluding Section 6.

2. THE HVM

To support the execution of the SCJ profile on low-end embedded systems, it has been built on top of the features of the HVM. The HVM is a lean Java virtual machine for low-end embedded devices. It is a Java-To-C compiler but supports interpretation as well. The main distinguishing feature of the HVM is its ability to translate a single piece of Java code into a self contained unit of ANSI-C compatible C code that can be included in an existing build environment without adding any additional dependencies. *The raison d'i£¡tre of the HVM is to support the stepwise addition of Java into an existing C based build and execution environment for low-end embedded systems such as the KT4585.* We call this feature *integrateability*, and it enables the translated Java code to be included in an existing, possibly non-standard, C based build and execution environment. Other important features of the HVM are,

- Intelligent class linking. A static analysis of the Java source base is performed. This computes a conservative estimate of the set of classes and methods that may be executed in a run of the program. Only this set is translated into C and included in the final executable
- Execution on the bare metal (no POSIX-like OS required). The generated source code is completely self contained and can be compiled and run without the presence of an OS or C runtime library
- Hybrid execution style. Individual methods (or all methods) can be marked for compilation into C or interpretation only. Control can flow from interpreted code into compiled code and vice versa. Java exceptions are supported and can be thrown across interpretation/compilation boundaries
- First level interrupt handling. The generated code is reentrant and can be interrupted at any point to allow for the immediate handling of an interrupt in Java space
- Hardware object support. The preferred way to program basic I/O in the HVM is to use Hardware Objects according to [23]. Hardware objects is a way to access device registers from Java code in a controlled manner
- Native variable support. Native variables as described in Section 2.1.1 are supported
- Portability. Generated code does not utilize compiler or runtime specific features and can be compiled by most cross compilers for embedded systems e.g. GCC or the IAR C Compiler from Nohau [15].

The HVM does not support garbage collection, but relies on the use of the SCJ scoped memory model.

2.1 The VM Interface

The minimal hardware interface layer as specified by the `VMInterface`, is implemented in the HVM almost entirely in Java. It contains the following main parts:

```
public interface VMInterface {
  public class AllocationArea {  ...  }

  public class Process { ... }
  public class ProcessScheduler { ... }
  public class ProcessSequencer { ... }

  public class RealtimeClock { ... }
}
```

The `AllocationArea` class defines the connection to a hardware memory area where objects and arrays can be created.

The `Process` classes define an execution context for the virtual machine, the Java stack and the logic to be executed. At a context switch, internal registers of the virtual machine and the native processor are saved on the stack.

The class `RealtimeClock` defines an interface to a hardware clock.

In the following we will look at how each part of this interface: the memory access part, the scheduling part, and the realtime clock part, is implemented in the HVM, keeping in mind that this will later form the basis for our SCJ implementation described in Section 3.

2.1.1 AllocationArea

The `VMInterface` defines a memory area for object allocation by two machine addresses (assumed to be 32 bit integers): the `base` of the area, and `free`, the first free location for a new object. Additionally, the maximal size of the area is given by the integer `size`. Thus the invariant `base + size >= free` is maintained.

During start-up the HVM statically allocates a single consecutive area of RAM memory to be used as the Java memory. Since the HVM does not support GC, but relies on the use of the SCJ memory model, the management of allocation in this consecutive area of RAM is very simple and based on only three variables:

```
unsigned char* HVMbase;
uint32_t HVMfree;
uint32_t HVMsize;
```

Thus there is a direct correlation between the `base`, `free`, and `size` variables of the `AllocationArea` and these variables in the HVM. If it could be supported to set the HVM variables directly from Java space, it could be controlled from Java space where allocation would take place. This is supported in the HVM through the novel concept of *native variables*. Native variables are a facility for changing, from Java space, the value of C data variables in the underlying VM. In the HVM, only static Java variables can be marked as native variables as illustrated in Figure 3 below.

```
@IcecapCVar
private static int HVMbase;
```

Marking a static Java variable as a native variable. Reads and writes to this variable will actually be to the native C variable with the same name instead.

Figure 3: Marking native variables

When this variable is accessed from Java it is not being set as a static class variable, rather the Java-to-C compiler will generate code accessing the C variable `HVMbase` instead.

In a similar manner the other variables are declared as native variables:

```
public class AllocationArea {
  private int base;
  private int size;
  private int free;
  ...
  @IcecapCVar
  private static int HVMbase;
  ...
  @IcecapCVar
  private static int HVMsize;
  ...
  @IcecapCVar
  private static int HVMfree;
  ...
}
```

Using this facility, Java methods can be called at runtime to set the point of allocation for Java heap data, thus directly controlling memory allocations from Java space. These methods are the following,

```
@IcecapCompileMe
public static int allocateBackingStore(
  int backingStoreSize) {...}
@IcecapCompileMe
public static void switchAllocationArea(
  AllocationArea newScope,
  AllocationArea oldScope) {...}
```

The static method `allocateBackingStore` takes a block from the current allocation area of size `backingStoreSize`. It returns the base of the area, given that the precondition

`HVMfree + backingStoreSize <= HVMbase + HVMsize`

holds. This method is really an allocation of a piece of uninitialized memory.

And finally, a static method `switchAllocationArea` saves the current allocation area in `oldScope` and sets it to `newScope`.

Both of these can be written in Java, but they have to be compiled in order to ensure atomic execution with respect to the HVM. The `@IcecapCompileMe` annotation ensures that the annotated method will always get compiled, and never interpreted.

This is a very lean interface to memory allocation. Yet it enables nested scoped allocation of memory areas at the Java level without polluting the virtual machine if these are not used, for instance in a garbage collection based environment.

2.1.2 Process

The context for an executable logical process is initialized by a static method:

```
static Process newProcess(Runnable logic, byte[] stack)
```

Preemption of a currently running process is done in an interrupt method of a private class which calls a scheduler to `getNextProcess` and then does a context switch.

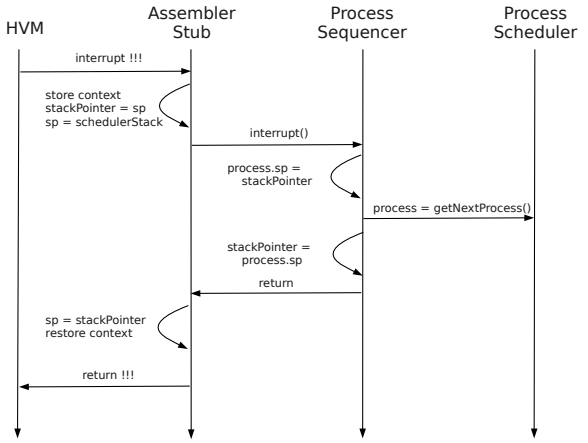


Figure 4: Context switch through the layers.

Figure 4 describes the sequence involved in a process preemption and context switch. First an interrupt occurs and an interrupt service routine implemented in assembler is entered. This routine saves all machine registers on the stack and then sets the C variable `stackPointer` to the current value of the stack pointer. Then the stack pointer is set to the beginning of a special stack used by the `ProcessSequencer` and `ProcessScheduler`. Now the flow of control enters Java space. The C code generated by the HVM tools are such that static Java methods can be called directly from C without any further setup. In this case the `ProcessSequencer` contains a static method `interrupt` which

gets called. It saves the C variable `stackPointer` in the preempted `Process` object, by mapping it to a native variable as described in the previous section. Then the `ProcessScheduler` is called to `getNextProcess`. The stack pointer of the new `Process` object is saved in the native variable `stackPointer` and flow of control returns to the interrupt service routine which sets the stack pointer to the value of `stackPointer` and restores the processor context and returns from interrupt handling to resume the execution of the next process.

The interrupt used to start the preemption and context switch can be based on a hardware clock of the underlying micro controller. Since the HVM supports Hardware Objects as described in [23] it is straightforward to start a hardware timer triggering an interrupt. Because the HVM generates reentrant code, the interrupt can be handled almost entirely in Java. The only part of the sequence in Figure 4 that is not implemented in Java is the assembler code stub manipulating the micro controller stack pointer register. This means that in order for the Process framework to run on a new architecture this assembler stub has to be reimplemented using the instruction set of the new target. Currently implementations exist for 32 and 64 bit Intel, the KT4585 and the AVR ATMega2560 [4].

2.1.3 RealtimeClock

This is a simple class that binds to a simple hardware clock.

```
public class  RealtimeClock {
 public static void getGranularity(RelativeTime grain) {
 ...}
 public static void getCurrentTime(AbsoluteTime now) {
 ...}
 public static void delayUntil(AbsoluteTime time) {
 ...}
}
```

Implementing the `RealtimeClock` class is platform specific. On the KT4585 several hardware clocks can be started through proper configuration of devices registers, which can be done using Hardware Objects. The current development platform for the KT4585 has a counter `uint32_t systemTick`. This variable is continuously updated by the KT4585 RTOS every 10 ms and used to measure system time. Using native variables this counter can be accessed directly from Java space:

```
@IcecapCVar
static int systemTick;
```

3. THE SCJ PROFILE

That particular event model mentioned above, suitable for programming low-end embedded platforms, is supported by the SCJ profile.

An application executes a sequence of missions. The missions are executed serially by an infrastructure `MissionSequencer` that has the flow shown in Figure 5.

A mission is a real-time program where schedulability can be checked. It consists of a fixed number of handlers with the usual temporal properties of a real-time process. Handlers are either *periodic* or *aperiodic*. The necessary assumptions to check schedulability of aperiodic handlers are not part of the profile. Level 2 of the profile, which is not included in this implementation, allows `MissionSequencer`s as a third kind of handler. Thereby Level 2 opens up for nested, concurrent missions.

Level 0 uses a *Cyclic executive* and therefore allows periodic handlers only. Level 1 has a *Fixed Priority Preemptive Scheduler* and admits aperiodic handlers as well.

Each handler has a *private memory* for temporary objects that are created during execution of the handler's application logic which is the `handleAsyncEvent` method. This memory is reset at the end of the `handleAsyncEvent` method call.

A mission has a *mission memory* in which the handlers of the mission are created together with objects that are shared between the handlers. The lifetime of the objects in mission memory is the lifetime of the mission.

Since a SCJ application consists of one or more missions, the application has an *immortal memory* in which the missions are created and which may contain inter-mission data as well. The lifetime of immortal memory is the lifetime of the application.

Thus the SCJ memory model defines the three allocation context memories for allocating objects: *immortal memory*, *mission memory*, and *private memory*. The concept of scoped memories is inherited from RTSJ, but simplified and restricted, and SCJ does not support a heap with a garbage collector.

Different SCJ implementations exist:

- on top of RTSJ
- with a native function layer to the JVM
- a bare metal version

The first two types of SCJ implementations were the first to come into existence [29, 18]. The second one is exemplified by the oSCJ/L0 implementation [18], whereas no SCJ implementation has until now been done using hardware objects and other hardware near features.

Due to the limitations of low-end embedded platforms where the event-driven programming model most often is used, we have implemented Level 0 and Level 1 only. The implementation follows the specification in [30] and is for single core controllers.

Because the implementation of full SCJ is done entirely in Java and looks much like previous implementations, only classes which use the `VMInterface` will be considered in the following, cf. Figure 2.

3.1 Implementing Scoped Memory

The SCJ `MemoryArea` class extends the `AllocationArea` class in `VMInterface`.

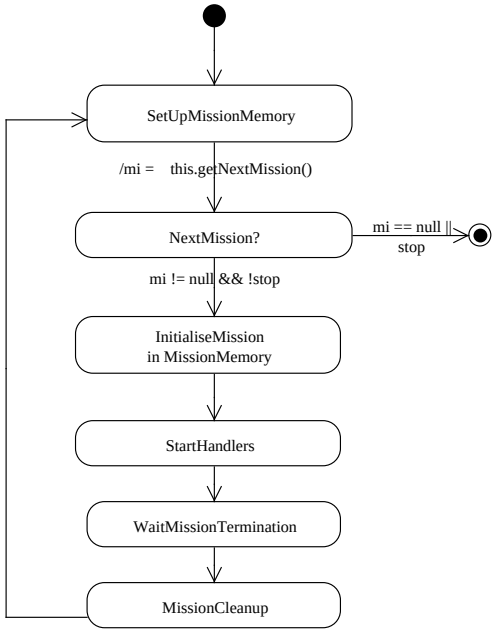


Figure 5: Mission sequencing.

It has a singleton static `AllocationAreaStack` to keep track of the memory area scopes. The stack method `pushAllocArea`, besides pushing the parameter allocation area `aa` on the stack also switches allocation areas; `popAllocArea` is implemented in a correspondingly way.

Two static fields, `immortal` and `currentArea`, hold references to the immortal memory and the current memory area, respectively.

```
public class MemoryArea extends AllocationArea
    implements AllocationContext
{
  static class AllocationAreaStack
  {
    private Stack<AllocationArea> allocationAreaStack;

    private static AllocationAreaStack stack
      = new AllocationAreaStack();
    ...
    void pushAllocArea(AllocationArea aa) {...}
    void popAllocArea() {...}
  }

  static ImmortalMemory immortal;
  static MemoryArea currentArea;
  ...
}
```

The `MemoryArea` class implements the methods specified in the `AllocationContext` interface:

```
public class MemoryArea extends AllocationArea
  implements AllocationContext
{
  ...
  public void executeInArea(Runnable logic)
    throws IllegalArgumentException {...}

  // the rest of the methods from AllocationContext
}
```

In `executeInArea`, the `push-` and `popAllocArea` are used:

```
public void executeInArea(Runnable logic) throws ... {
  ...
  AllocationAreaStack.instance().pushAllocArea(this);

  logic.run();

  AllocationAreaStack.instance().popAllocArea();
}
```

3.2 Implementing Process Scheduling

In SCJ the scheduling is based on the *cyclic executive* approach at Level 0, and the *fixed priority preemptive* approach at Level 1.

An infrastructure class `ScjProcess` uses `Process.newProcess` to instantiate a `Process` object for the HVM. This process object will be created for the specific platform architecture and encapsulates the SCJ handler.

```
class ScjProcess
{
  Process process;
  ManagedEventHandler target;

  ScjProcess(ManagedEventHandler handler, int[] stack)
  {
    this.target = handler;

    this.process = Process.newProcess(
      new Runnable() {
        public void run() {
          target.privateMemory.enter(new Runnable() {
            public void run() {
              target.handleAsyncEvent();
            }
          });
        }
      }, stack);
  }
  ..
}
```

The abstract class `ProcessScheduler` in `VMInterface` is extended by an infrastructure class `ScjProcessScheduler` which implements the `getNextProcess()` method. The encapsulated handler in the returned process object depends on the SCJ Level: At Level 0 the process object is returned with a mission sequencer; at Level 1 the handler with the highest priority is taken from a *priority queue* and returned in the process object.

```
public final Process getNextProcess() {
  ScjProcess scjProcess;
```

```
  if (scjLevel == 0) {
    scjProcess = CyclicScheduler.instance().
      getCurrentProcess();
    ...
  } else {
    scjProcess = PriorityScheduler.instance().
      move();
    ...
  }

  return scjProcess.process;
}
```

The context switch is done by the `ProcessSequencer` class which is instantiated by a scheduler. The call of `getNextProcess()` is in outline:

```
public abstract class ProcessSequencer
{
  public ProcessSequencer(ProcessScheduler scheduler) {...}

  private void interrupt() {
    ...
    saveSP(currentProcess);
    ...
    currentProcess = scheduler.getNextProcess();
    ...
    restoreSP(currentProcess);
    ...
  }
  ...
  public static ProcessSequencer
    getProcessSequencer(ProcessScheduler scheduler) {...}

  public final void start() {...}
}
```

At Level 1, processes waiting for release are stored in another priority queue sorted on absolute release time. This queue is updated as a side effect of the `getNextProcess` call.

Finally, in the two scheduler classes, `CyclicScheduler` and `PriorityScheduler`, the `getProcessSequencer(...).start()` method is called. The `CyclicScheduler` class is an infrastructure class.

3.3 Implementing Realtime Clock

The static methods in the `VMInterface.RealtimeClock` class, cf. Subsection 2.1.3, are used in SCJ by for example the `RealtimeClock` class.

The `delayUntil` construct is not absolutely needed. It can be replaced by a busy wait in the `CyclicExecutive` class where it is used. Yet, it may be more efficient in terms of for instance power to implement it by special processor instructions.

The `getGranularity` method is needed if an initialization needs to check assumptions that have been made in an offline schedulability test. It seems better to record such assumptions in a prelude instead of letting the application run with a platform mismatch.

4. EVALUATION

In this section we will check the ROM and RAM requirements for our SCJ implementation, and we will briefly look at execution efficiency though this is not the main focus of our effort.

ROM requirements are demonstrated with the well known benchmark miniCDj from [18]. The HVM can fully analyze, compile and run the miniCDj benchmark on 32 and 64 bit Intel platforms, but the benchmark requires a backing store of at least 300 kB, so we will not be able to run it on a low-end embedded system. Still, we will compile it for a low-end embedded system to assess how well the HVM program specialization can keep the ROM footprint down.

To demonstrate RAM requirements we will run a simple SCJ application consisting of 1 mission and 3 periodic handlers scheduled by a priority scheduler. This application can run with a backing store of approx 8 kB, thus allowing us to deploy it on the KT4585.

Finally we show preliminary measurements comparing the execution efficiency of the HVM with KESO and FijiVM.

4.1 ROM requirements

After some minor adjustments the miniCDj benchmark compiles against the `javax.safetycritical` package from our SCJ implementation described in Section 3. As JDK we use the OpenJDK 1.6.0 class libraries in this evaluation. After the HVM program specialization has optimized the application, a total of 151 classes and 614 methods are included in the final binary. These classes are divided between the packages as shown in Figure 6.

	Classes	Methods
`java.lang.*`	46	171
`java.util.*`	10	42
`javax.safetycritical.*`	46	185
`minicdj.*`	49	216
Total	151	614

Figure 6: Program specialization results

Since our KT4585 C-runtime does not support `float` and `double` - two data types used heavily by the miniCDj benchmark - we compiled the generated C code for a similar platform with `float` support: the AVR ATMega2560 platform from Atmel. This is a 8 bit architecture with 8 kB of RAM and 256 kB of flash. We compiled the code using the avr-gcc compiler tool chain [4] .

The resulting ROM requirements are listed in Figure 7. Results are listed for a *mostly interpreted* and for a *compilation only* configuration.

Using the *mostly interpreted* configuration, the ROM meets our goal with a large margin and is well below the 256 kB available on the ATMega2560. Using the *compilation only* configuration, the resulting application is approximately 276 kB and no longer fits onto the ATMega2560.

The reason for the difference in ROM size between the compilation and interpretation configuration, is that C code generated by the HVM Java-to-C compiler, requires more code

	ROM
Mostly interpreted	94682
Compilation only	282166

Compiling the miniCDj benchmark for an 8 bit low-end device (ATMega2560), using the HVM and the avr-gcc compiler tool-chain. Numbers in bytes.

Figure 7: HVM-SCJ ROM requirements

space than the original Java byte codes. Whether this is a general rule cannot be inferred from the above, and if the HVM Java-to-C compiler was able to produce tighter code the difference would diminish. But this experiment has an interesting side-effect and shows, that in the particular case of the HVM, the hybrid execution style allows us to run programs on low-end embedded devices, that we would otherwise not be able to fit on the device.

The work reported in [18] shows results from running the miniCDj benchmark on the OVM, but it does not report a resulting ROM size. It states however that the benchmark is run on a target with 8MB flash PROM and 64MB of PC133 SDRAM - a much larger platform than the ATMega2560.

4.2 RAM requirements

In our simple SCJ application with 1 mission and 3 handlers, the RAM usage can be divided into the parts shown in Figure 8. The stack sizes and the required sizes for the SCJ memory areas were found by carefully recording allocations and stack height in an experimental setup on a PC host platform. We then compiled the application for the KT4585 using the gcc cross compiler for the CR16c micro-controller (this benchmark does not utilize `float` or `double`).

SCJ related	
'Main' stack	1024
Mission sequencer stack	1024
Scheduler stack	1024
Idle task stack	256
3xHandler stack	3072
Immortal memory	757
Mission memory	1042
3xHandler memory	3x64 = 192
HVM infrastructure	
Various	959
Class fields	557
Total	9907

Numbers in bytes.

Figure 8: HVM-SCJ RAM requirements

The results show that a total of approx 10 kB RAM is required. The ROM size of the application is approx 35 kB. These numbers enable us to run SCJ applications on low-end embedded systems such as the KT4585.

4.3 Execution efficiency

To give a hint at the execution efficiency of the HVM compared to another well known similar environment we have executed 4 home-made benchmarks using both native C, KESO, and the HVM. This method is based on the same idea as the miniCDj and CDc benchmarks; to create similar programs in both C and Java and compare their execution time.

We executed the benchmarks on the ATMega2560 platform and accurately measured the clock cycles used using the AVR Studio 4 simulator. The results are listed in Figure 9.

	C	KESO	HVM
Quicksort	100	108	130
Trie	100	223	486
Determinant	100	190	408
WordReader	100	331	362
Average	100	213	347

Figure 9: Cycle count comparison

From this we can conclude that for the benchmarks tested, *KESO is approximately 2 times slower than C and the HVM is apporximately 3 times slower than C.*

To compare KESO, FijiVM, and HVM (GCJ included for reference as well), we had to move to a 32 bit Linux platform and arrived at the results listed in Figure 10. These results are obtained by measuring instruction counts using the PAPI API [16], and not clock cycles as in Figure 9.

	C	KESO	FijiVM	HVM	GCJ
Quicksort	100	101	136	111	172
Trie	100	93	54	136	245
Determinant	100	59	37	96	171
WordReader	100	251	218	177	328
Average	100	126	111	130	229

Figure 10: Instruction count comparison

A more elaborate description of the methods used to obtain these numbers are available from the HVM website [13].

5. RELATED WORK

The work reported here would have been impossible without the inspiration from other researchers. Some sources have already been mentioned in the preceding sections. In the following, the related work which we have studied, is more systematically summarized in the main innovation areas: Java virtual machines, program specialization, and bare metal implementation, as well as SCJ concepts and implementations.

5.1 Java Virtual Machines

Our main sources of inspiration have been the FijiVM [17] and the KESO VM [10]. Both are based on Java-to-C compilation, just like the HVM. The FijiVM ensures hard real-time execution guarantees, including real-time GC, of Java software on high-end embedded systems. It produces strikingly efficient C code (see Section 4.3). The FijiVM requires a POSIX-like OS and produces executables too large for low-end embedded systems. The KESO VM comes very close to

offer the same kind of integrateability as the HVM: it produces sufficiently small executables and strips away unused code. The code produced by KESO require the presence of certain OSEK specific header files and certain standard features of the C-runtime. But it is our impression that it would be possible to make KESO just as integrateable as the HVM with only a limited effort. Neither FijiVM nor KESO support interpretation, which is supported by the HVM. In comparison with FijiVM and KESO the main contribution of the HVM is the support for integrateability of Java into an existing C based build and execution environment.

5.2 Program Specialization

The method applied to do program specialization (the act of shrinking a given application to contain only actual dependencies) is an instance of the k-CFA algorithm as described in [24]. The core problem is actually that of doing type inference of virtual method calls. At virtual method call sites the HVM performs a *variable-type analysis* (VTA) as introduced by [28]. For all possible paths leading up to a virtual call site the analysis keeps track of all classes that may have been instantiated. This limits the set of implementations that may be invoked. It may be necessary to repeat the analysis as new possibly instantiated classes are discovered. This method is known to terminate (see [24]). No SCJ specific optimizations are applied.

During the program specialization phase the HVM loads class files using the BCEL API [2].

5.3 Bare Metal Implementation

Here we have essentially been building on the journal article [22] that consolidates work in which we have participated. For background on this subarea, we refer to the article.

5.4 SCJ

Clearly we have been inspired by having access to successive drafts from the committee on the SCJ profile [29, 30]. We have wholeheartedly embraced the main concepts structuring a real-time application, although we have questioned details and design decisions [6, 27]. For the memory model we have learned much from the JOP implementation [21]. Further, we have learned from the early implementations [18, 1]. As a stepping stone toward the implementation presented here, we have done an implementation using RTSJ [26] but based on delegation instead of inheritance as in [1]. Inheritance from RTSJ does not help in making an implementation lean.

6. CONCLUSION

The main result of this work is shown in the evaluation. It is indeed feasible to use an SCJ implementation for low-end platforms. It is important, because SCJ supports a programming style close to what embedded system programmers use, although in a safer language than the usual combination of C and assembly language. On this line we mention that we are already interacting with companies that would like to evaluate the potential for their developments.

Nevertheless, there is a research perspective as well, where this work may inspire others to take up ideas from this implementation. The key is the combination of a lean virtual machine, program specialization, and bare metal implementation, combined with the SCJ concepts in an infrastructure. This has been presented already, but during the work we have found a number of areas where they might be generalized or taken even further. Most of the ideas centre on the interface between the virtual machine and the infrastructure, which could be a more general profile, for instance full Java.

An issue that strikes us is, that a Level 0 application with a cyclic executive is essentially a sequential program. It could become that simple, if the initial thread is the initial mission sequencer, which continues to execute the `CyclicScheduler` until a misssion termination is signalled. That eliminates all context switching for the processor for cyclic executives; we have a sequential program execution.

The idea may be taken a step further. Since the mission sequencer is the only active entity between missions (Level 0 and 1), there is no reason why schedulers should not change between missions, that allows mixed Level 0 and Level 1 missions using the expected scheduling discipline. An added benefit might be that data structures for scheduling can be allocated in mission memory, thus they are reclaimed and do not fill up immortal memory.

Another idea is to include Object layout via `VMInterface`, that would pave the way for a garbage collector in Java, given that roots can be provided.

A simplification of the stub scheduler in the `VMInterface` seems feasible if the `RealtimeClock` becomes a proper interrupt handler. It should then call the 'interrupt' of the stub for every tick and update variables representing the time.

In the implementation, we have not yet included a monitor to realize synchronized methods. Since it must use a priority ceiling protocol for Level 1, it has to interact with the scheduler and thus requires some public disable/enable methods in the virtual machine interface.

Breaking away from these concrete ideas, we would finally like to mention that this implementation, which is already embedded in an Eclipse environment, needs to be supplemented by more tools before we have a development environment for a working programmer. However, this is a longer story. For some of our ideas in this context, we refer to [5].

7. ACKNOWLEDGMENTS

This work is part of the CJ4ES project and received partial funding from the Danish Research Council for Technology and Production Sciences under contract 10-083159. The authors are very grateful to Martin Scoerberl for his comments on an early draft. We would also like to thank the reviewers for their instructive comments which have helped us clarify several points in this paper.

8. REFERENCES

[1] aicas. `http://www.aicas.com/jamaica.html`. Visited June 2012.

[2] Apache. BCEL Manual. Available at: `http://commons.apache.org/bcel/manual.html`,

2012. Visited June 2012.

[3] Atego. Aonix Perc Pico. Available at: http://www.atego.com/products/aonix-perc-pico/.

[4] AVRFeaks. AVR Freaks. http://www.avrfreaks.net/, Visited June 2012.

[5] T. Bøgholm, C. Frost, R. R. Hansen, C. S. Jensen, K. S. Luckow, A. P. Ravn, H. Søndergaard, and B. Thomsen. Towards harnessing theories through tool support for hard real-time Java programming. *Innovations in Systems and Software Engineering*, June 2012.

[6] T. Bøgholm, R. R. Hansen, A. P. Ravn, B. Thomsen, and H. Søndergaard. A Predictable Java profile - rationale and implementations. In *JTRES 2009: Proceedings of the 7th international workshop on Java technologies for real-time and embedded systems*, pages 150–159, New York, NY, USA, 2009. ACM.

[7] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, 2000.

[8] J. Caska and M. Schoeberl. Java dust: how small can embedded Java be? In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '11, pages 125–129, New York, NY, USA, 2011. ACM.

[9] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42, New York, NY, USA, 2006. ACM Press.

[10] C. Erhardt, M. Stilkerich, D. Lohmann, and W. Schröder-Preikschat. Exploiting static application knowledge in a Java compiler for embedded systems: a case study. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '11, pages 96–105, New York, NY, USA, 2011. ACM.

[11] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 2003. ACM Press.

[12] T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer, and J. Vitek. CDx: a family of real-time Java benchmarks. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 41–50, New York, NY, USA, 2009. ACM.

[13] S. E. Korsholm. HVM (hardware near virtual machine). http://www.icelab.dk/, Visited June 2012.

[14] K. Nilsen. Differentiating features of the PERC virtual machine. Available at: http://www.aonix.com/pdf/PERCWhitePaper_e.pdf, 2009.

[15] NOHAU. http://www.nohau.se/iar. Visited January 2012.

[16] PAPI. Papi - the Performance Application Programming Interface. http://icl.cs.utk.edu/papi/index.html, 2012.

[17] F. Pizlo, L. Ziarek, and J. Vitek. Real time Java on resource-constrained platforms with Fiji vm. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 110–119, New York, NY, USA, 2009. ACM.

[18] A. Plsek, L. Zhao, V. H. Sahin, D. Tang, T. Kalibera, and J. Vitek. Developing safety critical Java applications with oSCJ/L0. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 95–101, New York, NY, USA, 2010. ACM.

[19] Polycom. http://www.polycom.dk/. Visited January 2012.

[20] Polycom. The KIRK DECT application module 6.0. http://www.polycom.eu/products/voice/wireless_solutions/dect_communications/modules/dect_krm_application.html, 2012.

[21] M. Schoeberl. Memory management for safety-critical Java. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '11, pages 47–53, New York, NY, USA, 2011. ACM.

[22] M. Schoeberl, S. Korsholm, T. Kalibera, and A. P. Ravn. A Hardware Abstraction Layer in Java. *ACM Trans. Embed. Comput. Syst.*, 10(4):42:1–42:40, Nov. 2011.

[23] M. Schoeberl, S. Korsholm, C. Thalinger, and A. P. Ravn. Hardware objects for Java. In *In Proceedings of the 11th IEEE International Symposium on Object/component/serviceoriented Real-time distributed Computing (ISORC 2008*. IEEE Computer Society, 2008.

[24] O. Shivers. http://www.ccs.neu.edu/home/shivers/citations.html#diss. Visited August 2012.

[25] F. Siebert. Realtime garbage collection in the JamaicaVM 3.0. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, JTRES '07, pages 94–103, New York, NY, USA, 2007. ACM.

[26] H. Søndergaard and A. P. Ravn. Implementation of Predictable Java (PJ) and Safety Critical Java (SCJ). http://it-engineering.dk/HSO/PJ/, Visited June 2012.

[27] H. Søndergaard, B. Thomsen, A. P. Ravn, R. R. Hansen, and T. Bøgholm. Refactoring Real-Time Java profiles. In *ISORC 2011: Proceedings 2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 109 – 116, Los Alamitos, CA, USA, 2011. IEEE.

[28] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for java. *SIGPLAN Not.*, 35(10):264–280, Oct. 2000.

[29] TheOpenGroup. Jsr 302: Safety Critical Java Technology. http://jcp.org/en/jsr/detail?id=302, 2006.

[30] TheOpenGroup. Safety-Critical Java Technology Specification. Draft Version 0.79, TheOpenGroup, May 2011.

[31] N. Wirth. *Programming in Modula-2*. Springer Verlag, 1985.

Towards a Real-Time, WCET Analysable JVM Running in 256 kB of Flash Memory

Stephan Korsholm
VIA University College
8700 Horsens, Denmark
sek@viauc.dk

Kasper Søe Luckow
Aalborg University
9220 Aalborg, Denmark
luckow@cs.aau.dk

Bent Thomsen
Aalborg University
9220 Aalborg, Denmark
bt@cs.aau.dk

1 Introduction

The Java programming language has recently received much attention in the real-time systems community as evidenced by the wide variety of initiatives, including the Real-Time Specification for Java[8], and related real-time profiles such as Safety-Critical Java[6], and Predictable Java[3]. All of these focus on establishing a programming model appropriate for real-time systems development. The motivation for the profiles has been to further tightening the semantics, for accommodating static analyses, such as Worst Case Execution Time (WCET) analysis that serves an integral role in proving temporal correctness.

Evidently, the presence of the Java Virtual Machine (JVM) adds to the complexity of performing WCET analysis. To reduce the complexity, a direction of research has focused on implementing the JVM directly in hardware[9]. To further extend the applicability of real-time Java, we want to allow software implementations of the JVM executed on common embedded hardware, such as ARM and AVR, while still allowing the system to be amenable to static analyses. This necessarily demands that the JVM is amenable to static analyses as well, which is difficult, since the JVM specification is rather loosely defined. Specifically, the JVM specification emphasises on *what* a JVM implementation must do whenever executing a Java Bytecode, but leaves *how* unspecified. This makes JVM vendors capable of tailoring their implementation to their application domain.

In our recent research, we have developed a WCET analysis tool called Tool for Execution Time Analysis of Java bytecode (TetaJ)[5], which allows for taking into account a software implemented JVM and the hardware. The development of TetaJ has made us explicitly reason about JVM design accommodating WCET analysis. The contribution of this paper is to present our preliminary research efforts in making Java tractable for real-time embedded systems on more common execution environments by elaborating on *how* a real-time JVM must handle certain issues. Constraining the degree of freedom of the real-time JVM vendors is a necessity to ensure, that the application running on the JVM is temporally correct, since the WCET is often obtained using static analyses relying on predictable behaviour.

2 TetaJ

TetaJ employs a static analysis approach where the program analysis problem of determining the WCET of a program is viewed as a model checking problem. This is done by reconstructing the control flow of the program and the JVM implementation, and generate a Network of Timed Automata (NTA) amenable to model checking using the state-of-the-art UPPAAL model checker[1]. The NTA is structured such that the model checking process effectively simulates an abstract execution of the Java Bytecode program on the particular JVM and hardware.

TetaJ has proven suitable for iterative development since it analyses on method level, and because analysis time and memory consumption are reasonably low. In a case study[5, 2], an application consisting of 429 lines of Java code was analysed in approximately 10 minutes with a maximum memory consumption of 271 MB. The case study is based on the Atmel AVR ATmega2560 processor and the

Hardware near Virtual Machine (HVM)[1] which is a representative example of a JVM targeted at embedded systems.

Currently, TetaJ provides facilities for automatically generating an NTA representing the JVM by the provision of the HVM executable. The METAMOC[4] hardware models are reused in TetaJ thereby having the desirable effect that TetaJ can benefit from the continuous development of METAMOC.

We have strong indications that TetaJ produces safe WCET estimates, that is, estimates that are at least as high as the actual WCET and TetaJ may therefore be appropriate for analysing hard real-time Java programs. As to the precision, we have results showing that TetaJ produces WCET estimates with as low as 0.6% of pessimism[5].

3 Hardware near Virtual Machine

The HVM is a simplistic and portable JVM implementation targeted at embedded systems with as low as 256 kB of flash memory and 8 kB of RAM and is capable of running bare-bone without operating system support. To support embedded systems development, the HVM implements the concept of *hardware objects*[7], that essentially prescribe an object-oriented abstraction of low-level hardware devices, and allow for first-level interrupt handling in Java space.

The HVM employs iterative interpretation for translating the Java Bytecodes to native machine instructions. The interpreter itself is compact, and continuously fetches the next Java Bytecode, analyses it, and finally executes it. The analyse and execute stages are implemented by means of a large switch-statement with cases corresponding to the supported Java Bytecodes.

A special characteristic of the HVM is that the executable is adapted to the particular Java Bytecode program. Specifically, the Java Bytecode of the compiled program is encapsulated in arrays within the HVM itself. This, however, does not affect the behaviour of the interpreter, and is merely a way of bundling the Java Bytecode with the HVM into a single executable.

4 Preliminary Design Criteria for a Predictable HVM

During the development of TetaJ, the implementation of the HVM has been inspected and modified according to the needs of WCET analysis. Some modifications simply comprise bounding the number of loop iterations while others require more elaborate solutions to be developed. In the following, we present our experiences with modifying the HVM towards predictable and WCET analysable behaviour.

4.1 Eliminating Recursive Solutions

Some Java Bytecode implementations are intuitively based on recursive solutions. Specifically, the Java Bytecodes responsible for method invocations such as *invokevirtual* employ a recursive approach.

The heart of the HVM is the *methodInterpreter* which implements the interpretation facilities. Whenever e.g. *invokevirtual* is executed, the *methodInterpreter* is recursively called to process the code of the invoked method. This, however, is undesirable seen from a static WCET analysis perspective, since it is difficult to statically determine the depth of the recursive call. The problem is circumvented by introducing an iterative approach and introduce the notion of a call stack and stack frames. Using this solution, a stack frame containing the current call context, that is, stack pointer, program counter etc. are pushed onto the stack, and the *methodInterpreter* simply continues iteratively fetching Java Bytecodes from the called method. When the particular method returns, the stack is popped and the context restored to the point prior to the method invocation.

[1] http://www.icelab.dk

4.2 Reducing Pessimism of the Class Hierarchy

Since Java is strongly typed, type casts produce the *checkcast* Java Bytecode which is responsible for iteratively checking the class hierarchy to determine whether the type cast is type compatible. Another example is the *instanceof* operator which similarly consults the class hierarchy iteratively. Establishing a tight bound that applies for every part of the class hierarchy cannot be done statically. Instead it is only possible to establish a global bound corresponding to the maximum depth of the class hierarchy. This gives rise to pessimism that affects the resulting WCET extensively.

This problem has been resolved by harnessing that the HVM is adapted to the particular application. Because this process is performed prior to runtime, it is possible to exercise how the class hierarchy is built and construct a matrix specifying how classes are interrelated. The matrix will be incorporated in the final executable, and can be used for determining type compatibility among classes in constant time, by simply looking up the matrix.

4.3 Constant Time Analyse Stage

Different compilers and different optimisation levels may or may not implement a sufficiently large switch-statement as a look-up table. Because of this uncertainty, we have replaced the analyse stage in the *methodInterpreter* to ensure that this stage is performed in constant time regardless of compiler and optimisation levels. The replacement consists of extracting the individual Java Bytecode implementations from the switch-statement into respective functions. This also has the desirable side-effect that they are easily located in the disassembled HVM executable. An array of function-pointers to each of these functions substitutes the original switch-statement, thereby allowing for constant access time to each of the Java Bytecode implementations using the opcodes as look-up keys.

References

[1] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL - A Tool Suite for Automatic Verification of Real-time Systems. *Hybrid Systems III*, pages 232–243, 1996.

[2] Thomas Bøgholm, Christian Frost, Rene Rydhof Hansen, Casper Svenning Jensen, Kasper Søe Luckow, Anders P. Ravn, Hans Søndergaard, and Bent Thomsen. Harnessing theories for tool support. *Submitted for publication: Innovations in Systems and Software Engineering*, 2011.

[3] Thomas Bøgholm, René R. Hansen, Anders P. Ravn, Bent Thomsen, and Hans Søndergaard. A predictable java profile: Rationale and implementations. In *JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 150–159, New York, NY, USA, 2009. ACM.

[4] Andreas E. Dalsgaard, Mads Chr. Olesen, Martin Toft, René Rydhof Hansen, and Kim Guldstrand Larsen. METAMOC: Modular Execution Time Analysis using Model Checking. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 113–123. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010.

[5] C. Frost, C. S. Jensen, K. S. Luckow, and B. Thomsen. WCET Analysis of Java Bytecode Featuring Common Execution Environments. *Accepted for publication: The 9th International Workshop on Java Technologies for Real-time and Embedded Systems - JTRES 2011*, 2011.

[6] JSR302. The java community process, 2010. `http://www.jcp.org/en/jsr/detail?id=302`.

[7] Stephan Korsholm, Anders P. Ravn, Christian Thalinger, and Martin Schoeberl. Hardware objects for java. In *In Proceedings of the 11th IEEE International Symposium on Object/component/serviceoriented Real-time distributed Computing (ISORC 2008*. IEEE Computer Society, 2008.

[8] Oracle. RTSJ 1.1 Alpha 6, release notes, 2009. `http://www.jcp.org/en/jsr/detail?id=282`.

[9] Martin Schoeberl. JOP: A Java optimized processor. In *On the Move to Meaningful Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2003)*, volume 2889 of *LNCS*, pages 346–359, Catania, Italy, November 2003. Springer.

References

[1] T.J. Watson libraries for analysis (WALA). `http://wala.sourceforge.net`, Visited August 2012 2012.

[2] Uppaal. `http://www.uppaal.com/`, Visited August 2012 2012.

[3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[4] aicas. `http://www.aicas.com/jamaica.html`. Visited June 2012.

[5] Apache. BCEL Manual. Available at: `http://commons.apache.org/bcel/manual.html`, 2012. Visited June 2012.

[6] AVRFeaks. AVR Freaks. `http://www.avrfreaks.net/`, Visited June 2012.

[7] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. *SIGPLAN Not.*, 35(5):1–12, May 2000.

[8] T. Bøgholm, C. Frost, R. R. Hansen, C. S. Jensen, K. S. Luckow, A. P. Ravn, H. Søndergaard, and B. Thomsen. Towards Harnessing Theories Through Tool Support for Hard Real-Time Java Programming. *Innovations in Systems and Software Engineering*, pages 1–12, 2012.

[9] P. Bothner. `http://www.linuxjournal.com/article/4860`. Visited march 2012.

[10] F. Brandner, T. Thorn, and M. Schoeberl. Embedded JIT Compilation with CACAO on YARI. In *Proceedings of the 2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, ISORC '09, pages 63–70, Washington, DC, USA, 2009. IEEE Computer Society.

[11] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, Aug. 2000.

[12] C. Chambers and D. Ungar. Customization: Optimizing Compiler Technology for SELF, a Dynamically-typed Object-Oriented Programming Language. *SIGPLAN Not.*, 24(7):146–160, June 1989.

[13] L. R. Clausen, U. P. Schultz, C. Consel, and G. Muller. Java Bytecode Compression for Low-end Embedded Systems. *ACM Trans. Program. Lang. Syst.*, 22(3):471–489, May 2000.

[14] R. Costa and E. Rohou. Comparing the Size of .NET Applications with Native Code. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '05, pages 99–104, New York, NY, USA, 2005. ACM.

[15] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 25–35, New York, NY, USA, 1989. ACM.

[16] L. P. Deutsch and A. M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM.

[17] Diasemi. Dialog semiconductor. `http://www.diasemi.com/single-chip-dect-cat-iq-solution`. Visited January 2012.

[18] L. Dickman. A Comparison of Interpreted Java, WAT, AOT, JIT, and DAC. `http://www.helmitechnologies.com/campaign/knowledge_kit/esmertec.pdf`, 2002.

[19] DSE. `http://www.dseair.dk/`. Visited June 2012.

[20] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying Event-driven Programming of Memory-constrained Embedded Systems. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42, New York, NY, USA, 2006. ACM Press.

[21] C. Erhardt, M. Stilkerich, D. Lohmann, and W. Schröder-Preikschat. Exploiting Static Application Knowledge in a Java Compiler for Embedded Systems: A Case Study. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '11, pages 96–105, New York, NY, USA, 2011. ACM.

[22] M. A. Ertl, D. Gregg, A. Krall, and B. Paysan. Vmgen: a generator of efficient virtual machine interpreters. *Softw. Pract. Exper.*, 32(3):265–294, Mar. 2002.

[23] C. Frost, C. S. Jensen, K. S. Luckow, and B. Thomsen. Wcet analysis of java bytecode featuring common execution environments. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '11, pages 30–39, New York, NY, USA, 2011. ACM.

[24] M. Fulton and M. Stoodley. Compilation techniques for real-time java programs. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 221–231, Washington, DC, USA, 2007. IEEE Computer Society.

[25] E. Gagnon, L. J. Hendren, and G. Marceau. Efficient inference of static types for java bytecode. In *Proceedings of the 7th International Symposium on Static Analysis*, SAS '00, pages 199–219, London, UK, UK, 2000. Springer-Verlag.

[26] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 2003. ACM Press.

[27] GCC. `http://gcc.gnu.org/install/specific.html`. Visited march 2012.

[28] GCC. `http://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html`. Visited June 2012.

[29] GNU. `http://www.gnu.org/software/classpath/classpath.html`. Visited June 2012.

[30] GNU. `http://ulibgcj.sourceforge.net/`. Visited June 2012.

[31] O. Group. `http://portal.osek-vdx.org/files/pdf/specs/os223.pdf`. Visited May 2012.

[32] Grundfos. `http://www.grundfos.com/`. Visited June 2011.

[33] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, pages 21–38, London, UK, UK, 1991. Springer-Verlag.

[34] jamvm. `http://jamvm.sourceforge.net/`. Visited June 2011.

[35] D.-H. Jung, S.-M. Moon, and S.-H. Bae. Evaluation of a java ahead-of-time compiler for embedded systems. *Comput. J.*, 55(2):232–252, Feb. 2012.

[36] T. Kalibera, J. Hagelberg, P. Maj, F. Pizlo, B. Titzer, and J. Vitek. A family of real-time java benchmarks. *Concurr. Comput. : Pract. Exper.*, 23(14):1679–1700, Sept. 2011.

[37] S. Korsholm and P. Jean. The Java legacy interface. In *JTRES '07: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 187–195, New York, NY, USA, 2007. ACM.

[38] S. Korsholm, M. Schoeberl, and A. P. Ravn. Interrupt handlers in Java. In *Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 453–457, Washington, DC, USA, 2008. IEEE Computer Society.

[39] A. Krall and R. Grafl. Cacao – a 64 bit javavm just-in-time compiler. In *Proceedings of the Workshop on Java for Science and Engineering Computation*, PPoPP 97. G. C. Fox and W. Li Eds. ACM, 1997.

[40] B. B. Kristensen, O. L. Madsen, and B. Møller-Pedersen. The when, why and why not of the beta programming language. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 10–1–10–57, New York, NY, USA, 2007. ACM.

[41] P. Nanthanavoot and P. Chongstitvatana. Code-Size Reduction for Embedded Systems using Bytecode Translation Unit. In *Conf. of Electrical/Electronics, Computer, Telecommunications, and Information Technology (ECTI)*, 2004.

[42] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[43] K. Nilsen. Differentiating features of the perc virtual machine. Available at: `http://www.aonix.com/pdf/PERCWhitePaper_e.pdf`, 2009.

[44] NOHAU. `http://www.nohau.se/iar`. Visited January 2012.

[45] NOHAU. `http://www.iar.com/en/Products/IAR-Embedded-Workbench/`. Visited February 2012.

[46] OPENJDK. `http://openjdk.java.net/`. Visited June 2012.

[47] PAPI. Papi - the Performance Application Programming Interface. `http://icl.cs.utk.edu/papi/index.html`, 2012.

[48] G. Phipps. Comparing observed bug and productivity rates for Java and c++. *Softw. Pract. Exper.*, 29:345–358, April 1999.

[49] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: fragmentation-tolerant real-time garbage collection. *SIGPLAN Not.*, 45(6):146–159, June 2010.

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

[51] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. *SIGPLAN Not.*, 29(10):324–340, Oct. 1994.

[52] A. Plsek, L. Zhao, V. H. Sahin, D. Tang, T. Kalibera, and J. Vitek. Developing safety critical java applications with oscj/l0. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 95–101, New York, NY, USA, 2010. ACM.

[53] Polycom. `http://www.polycom.dk/`. Visited January 2012.

[54] Polycom. The kirk dect application module 6.0. http://www.polycom.eu/products/voice/wireless_solutions/dect_communications/modules/dect_krm_application.html, 2012.

[55] E. Quinn and C. Christiansen. Java pays – positively. Available at: http://www.justice.gov/atr/cases/exhibits/1344.pdf, 1998. Visited February 2012.

[56] V. Research. 2011 EMBEDDED SOFTWARE & TOOLS MARKET INTELLIGENCE SERVICE. http://www.vdcresearch.com/, 2011.

[57] M. Schoeberl, S. Korsholm, C. Thalinger, and A. P. Ravn. Hardware objects for Java. In *In Proceedings of the 11th IEEE International Symposium on Object/component/serviceoriented Real-time distributed Computing (ISORC 2008*. IEEE Computer Society, 2008.

[58] N. Semiconductor. http://www.national.com/. Visited January 2012.

[59] N. Semiconductor. CR16C, Programmers Reference Manual. http://www.national.com/appinfo/cp3000/publicfiles/Prog_16C.pdf. Visited January 2012.

[60] O. Shivers. http://www.ccs.neu.edu/home/shivers/citations.html#diss. Visited August 2012.

[61] F. Siebert. Realtime garbage collection in the jamaicavm 3.0. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, JTRES '07, pages 94–103, New York, NY, USA, 2007. ACM.

[62] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for java. *SIGPLAN Not.*, 35(10):264–280, Oct. 2000.

[63] H. Sndergaard, S. Korsholm, and A. P. Ravn. Safety-Critical Java for Low-End Embedded Platforms. Accepted for JTRES'12, Pending publication, 2012.

[64] A. S. Tanenbaum, H. van Staveren, E. G. Keizer, and J. W. Stevenson. A practical tool kit for making portable compilers. *Commun. ACM*, 26(9):654–660, Sept. 1983.

[65] TheOpenGroup. Safety-Critical Java Technology Specification (JSR-302). Draft Version 0.79, TheOpenGroup, May 2011.

[66] R. Vallée-Rai, P. Co, E. Gagnon, L. H. n, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[67] D. Van Horn and H. G. Mairson. Deciding kcfa is complete for exptime. *SIGPLAN Not.*, 43(9):275–282, Sept. 2008.

[68] G. Wiki. `http://gcc.gnu.org/wiki/History`. Visited march 2012.

[69] Wikipedia. `http://en.wikipedia.org/wiki/P-code_machine`. Visited August 2012.