

# 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 [18], JamaicaVM [26] and PERC [15] show that Java can be executed efficiently on high-end embedded devices, thus allowing embedded software engineers to reap the benefits from using Java with its tools and method; benefits that desktop and server developers have had for some time.

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 2012 October 24-26, 2012, Copenhagen, Denmark  
Copyright 2012 ACM 978-1-4503-1688-0 ...\$15.00.

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 run-time errors, and for specializing programs so they become more resource efficient.

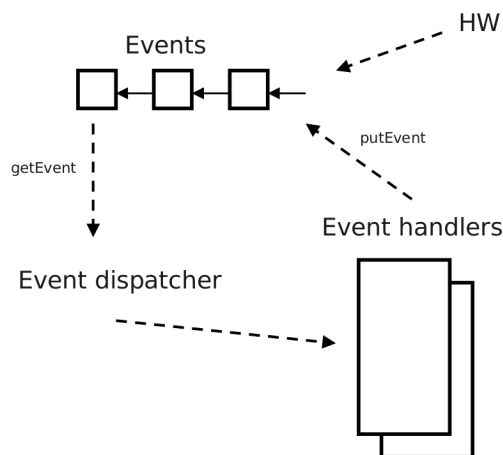


Figure 1: Event Driven Scheduling

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 [30] implementation that runs on systems with a few kB 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* [21] from Polycorn [20], also called the KT4585. This module is used to wirelessly transmit voice and data using the DECT protocol. It has the following features:

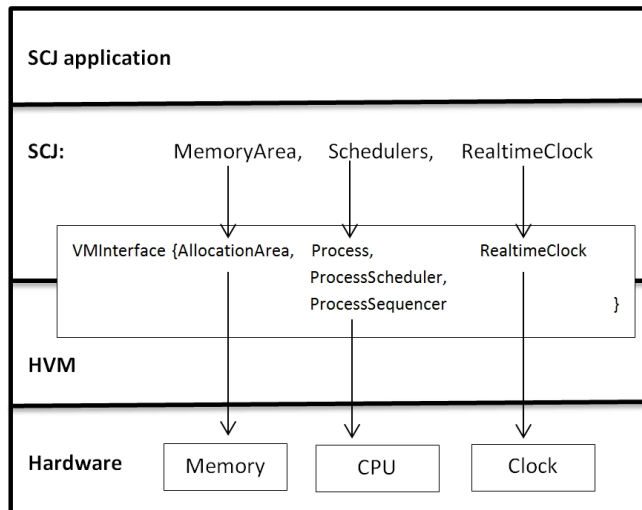
- 40 Mhz, 16 bit RISC architecture

- 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, such legacy event-driven implementations are common, although the architecture is difficult 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 improve programming 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 can be made to run at this level and fit well with known architectural concepts.



**Figure 2: SCJ architecture with the VMInterface to HVM.**

## 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 [14] 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 the `VMInterface`. According to Figure 2 this interface is divided into three parts:

- Memory allocation is controlled through the class `AllocationArea`. Using facilities of this class, the SCJ infrastructure controls 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 implement SCJ scoped memory semantics
- Process scheduling is done through an interface to the CPU that defines a minimal `Process` object<sup>1</sup>, 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; also, in order to avoid busy waits, it has a `delayUntil` method.

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. 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 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 [13], called miniCDj [19]. 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 (described in Section 5.2).

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.

<sup>1</sup>So named in recognition of the Modula 2 ancestry [32]

## 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 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 such as the KT4585.* We call this feature *integrability*, 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 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 immediate handling of an interrupt in Java space
- Hardware object support. Small systems do not use stream oriented character I/O, they use specialized devices for instance diodes or buttons. Therefore the HVM implements Hardware Objects according to [24]. Hardware objects are 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 [16].

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 look at how each part of this interface is implemented in the HVM, keeping in mind that this will form the basis for the SCJ implementation 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, 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 C variable with the same name.

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 allocates a piece of un-initialized 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 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

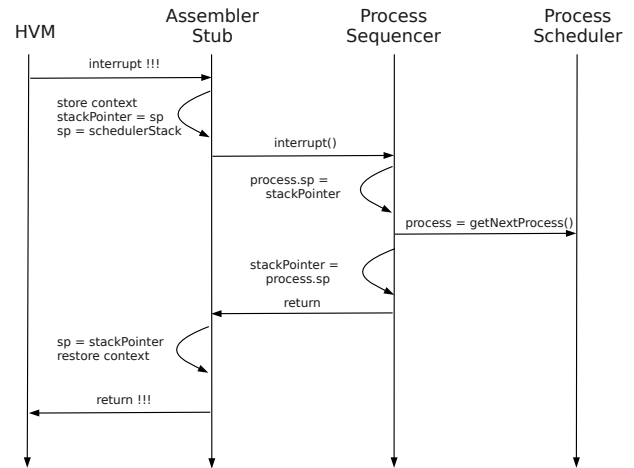


Figure 4: Context switch through the layers.

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 come from a hardware clock of the underlying micro controller. Since the HVM supports Hardware Objects as described in [24], 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 device 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;

```

Thus reading the time is a simple access to an integer variable from a Java program.

### 3. THE SCJ IMPLEMENTATION

For readers that are not familiar with the SCJ profile, we give a brief introduction to its main concepts, before key implementation decisions are highlighted.

#### 3.1 The SCJ concepts

Initially, note that the event model mentioned in the introduction as suitable for programming low-end embedded platforms is supported by the SCJ profile. Briefly, an application executes a sequence of missions. The missions are executed serially by an infrastructure *MissionSequencer* that operates as 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 profile does not support *sporadic* handlers; thus the assumptions necessary to check schedulability of aperiodic handlers are not part of the profile. Level 2 of the profile, which is not implemented here, has *MissionSequencers* as a third kind of handler. Thereby Level 2 opens up for nested, concurrent missions.

Level 0 of the profile targets a *Cyclic executive* and therefore admit periodic handlers only. Level 1 has a *Fixed Priority Preemptive Scheduler* and admits aperiodic event triggered handlers as well.

#### Memory Model

The SCJ memory model is inherited from the RTSJ scoped memory model, but much simplified, and SCJ does not support a heap with a garbage collector.

Each handler has a *private memory* for objects that live only during activation of the handler's application logic, which is a *handleAsyncEvent* method. This memory is reset at the end of every *handleAsyncEvent* call.

A mission has a *mission memory* that holds the handlers of the mission as well as objects that are shared among 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 entire application.

#### 3.2 SCJ implementation strategies

Different SCJ implementations exist:

- on top of RTSJ
- with a native function layer to the JVM
- a bare metal version.

The first kind of SCJ implementation was quite naturally the first to appear [30]. The second kind is exemplified by the oSCJ/L0 implementation [19], whereas no SCJ software implementation until now has relied on hardware objects and other hardware near features only.

We have implemented Level 0 and Level 1, because they target applications running on low-end embedded platforms. The implementation follows the specification in [31]. Some

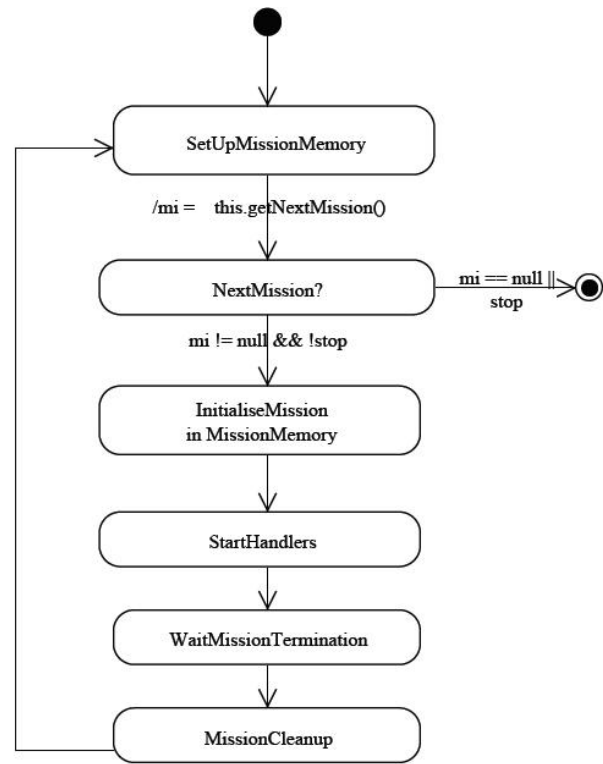


Figure 5: Mission sequencing.

debated features like long events, happenings, and the RTSJ interface to hardware are not implemented. Currently the priority scheduler does not support priority ceiling emulation. Also, the proposed standard library and I/O are not included. The implementation is for single core processors.

Because much of the implementation of a SCJ profile is done entirely in Java and looks similar in all implementations, only classes which use the *VMInterface* will be considered in the following, cf. Figure 2. The complete source code of the actual implementation, together with the generated Javadoc documentation, is available at the HVM homepage<sup>2</sup>.

#### 3.3 Implementing Scoped Memory

Scoped memory was already in RTSJ, and the simplifications that SCJ permits are explored thoroughly in the JOP implementation [22]. Thus there were no major design decisions. In the implementation, the ground work was done by the HVM as described in the previous section.

The SCJ *MemoryArea* class extends the *AllocationArea* class in *VMInterface*.

It has a singleton static *AllocationAreaStack* to keep track of the active memory area scopes. The method *pushAllocArea* pushes its argument on the stack and switches allocation area. Its converse *popAllocArea* is implemented correspondingly.

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
```

<sup>2</sup><http://www.icelab.dk/>

```

{
    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;
    ...
}

```

It also implements the SCJ `AllocationContext` interface: In particular the method that combines change of scope with executing a `Runnable` logic.

```

    public void executeInArea(Runnable logic)
        throws IllegalArgumentException {...}
}

```

In `executeInArea`, the `push-` and `popAllocArea` are used. Thus the machinery is ready for executing handlers of a mission entering and exiting mission scopes, handler scopes and nested private memories.

### 3.4 Implementing Missions with Scheduling

The key to understanding missions and scheduling is given by the state machine in Figure 5. Here it is helpful to think of the HVM as executing a primordial infrastructure mission with just one handler, a mission sequencer.

Immortal memory is already existing, it is the HVM's initial memory area. It is the mission memory of the primordial mission. Now the initial mission sequencer has to become active. Recall that it is a handler, so for this handler a private memory is allocated by the infrastructure. It is entered and the `handleAsyncEvent` method implements the logic of Figure 5.

However, when it comes to the states "StartHandlers" and "WaitMissionTermination", it acts differently for Level 0 with a *cyclic executive* approach and Level 1 with *fixed priority preemptive* scheduling.

For a Level 0 mission, it is very simple. The mission sequencer will execute the handlers in the order given by a table provided by the method `getSchedule` of the Level 0 mission. This is implemented in an infrastructure class `CyclicScheduler`. When mission termination is requested by one of the handlers, this implementation will exit at the beginning of a major cycle. The SCJ profile leaves it to the implementation to interpret the concept of mission termination. Also, both initialisation and finalisation of a mission does not have any timing constraints in the profile.

For a Level 1 mission, a `PriorityScheduler` is created. For each handler in the mission an infrastructure `ScjProcess`, which contains a HVM `Process` (see Section 2.1.2), is created.

```

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);
    }
    ...
}

```

At the end the `getProcessSequencer(...).start()` method is called, and the mission becomes active. Termination is observed by the `PriorityScheduler` if there are no more released processes. Control then returns to the original mission handler.

In order to do this, the abstract class `ProcessScheduler` in `VMInterface` is extended by an infrastructure class called `ScjProcessScheduler` which implements the `getNextProcess()` method, using priority queues for next releases and released processes. The context switch is done by the `ProcessSequencer` class which is instantiated by a scheduler.

```

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() {...}
}

```

### 3.5 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 off-line 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 [19]. 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
<code>java.lang.*</code>	46	171
<code>java.util.*</code>	10	42
<code>javax.safetycritical.*</code>	46	185
<code>minicdj.*</code>	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 found in Figure 7 for a *mostly interpreted* and for a *compilation only* configuration.

miniCDj benchmark	ROM (bytes)
Mostly interpreted	94682
Compilation only	282166

Figure 7: HVM-SCJ ROM requirements

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 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 [19] 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 8 MB flash PROM and 64 MB 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	bytes
'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
<b>HVM infrastructure</b>	
Various	959
Class fields	557
<b>Total</b>	9907

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.

From this we can conclude that for the benchmarks tested, *KESO is approximately 2 times slower than C and the HVM is approximately 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 [17], and not clock cycles as in Figure 9. A more elaborate description of the methods used to obtain these numbers is available from the HVM website [14].

	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

	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

#### 4.3.1 Discussion

One of the reasons why the HVM is considerably slower than KESO (see Figure 9) is its support for full Java exception handling. KESO detects the occurrence of exceptions, but does not yet support that they are caught in outer scopes. Based on very rudimentary experiments not shown here we get a hint that the cost of exception support is an approx. 25% decrease in performance for the HVM. Also, as a consequence of supporting the hybrid execution style of mixed interpretation/compilation, the HVM passes the Java stack pointer to all method calls. This is a 4 byte entity that is carried across all method invocations. The cost of this is not exactly known, but may account for a significant part of the difference in execution efficiency.

On the other hand KESO supports garbage collection (GC), and though GC were not activated while running the benchmarks of Figure 9, the KESO VM pays a price in terms of execution efficiency for the presence of GC. Finally, KESO applies a range of optimizations not yet implemented in the HVM. E.g. 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.

So far, KESO holds the torch for being the most efficient VM for low-end embedded systems, and it is our goal to improve the HVM execution efficiency while maintaining integratability - a feature very important to the HVM.

## 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 [18] 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 integratability as the HVM: it produces sufficiently small executables and strips away unused code. The code produced by KESO requires the presence of certain OSEK [12] 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 integratable 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 support for a mixture of compact interpreted code and compiled code as well as support for integratability of Java into an existing C based build and execution environment.

In our experience from advocating the idea of using Java for low-end embedded systems to companies which already have a large amount of C based source code for their embedded devices, the feature of integratability is very important. We have found that the typical existing embedded developer will be reluctant to abandon his currently used tool-chain. In many cases the compiler, linker, and debugger are 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 and the build procedure supported by the IDE are hard to change. Embedded programming is notoriously difficult, and moving to a different kind of compiler or having to make significant changes to build procedures will most likely be tasks that developers avoid.

### 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 [25]. The core problem is actually that of doing type inference of virtual method calls. At virtual method calls, the HVM performs a *variable-type analysis* (VTA) as introduced by [29]. 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 [25]). 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 [23] 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 [30, 31].



We have wholeheartedly embraced the main concepts structuring a real-time application, although we have questioned details and design decisions [6, 28]. For the memory model we have learned much from the JOP implementation [22]. Further, we have learned from the early implementations [19, 1]. As a stepping stone toward the implementation presented here, we have done an implementation using RTSJ [27] 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 mission 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 enables 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].

## 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 Schoeberl for his comments on an early draft. We would also like to thank the reviewers for their constructive comments which have helped us clarify several points in this paper.

## 7. 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] O. Group. <http://portal.osek-vdx.org/files/pdf/specs/os223.pdf>. Visited May 2012.

- [13] 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.
- [14] S. E. Korsholm. HVM (Hardware near Virtual Machine). <http://www.icelab.dk/>, Visited June 2012.
- [15] K. Nilsen. Differentiating features of the PERC virtual machine. Available at: [http://www.aonix.com/pdf/PERCWhitePaper\\_e.pdf](http://www.aonix.com/pdf/PERCWhitePaper_e.pdf), 2009.
- [16] NOHAU. <http://www.nohau.se/iar>. Visited January 2012.
- [17] PAPI. Papi - the Performance Application Programming Interface. <http://icl.cs.utk.edu/papi/index.html>, 2012.
- [18] 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.
- [19] 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.
- [20] Polycom. <http://www.polycom.dk/>. Visited January 2012.
- [21] Polycom. The KIRK DECT application module 6.0. [http://www.polycom.eu/products/voice/wireless\\_solutions/dect\\_communications/modules/dect\\_krm\\_application.html](http://www.polycom.eu/products/voice/wireless_solutions/dect_communications/modules/dect_krm_application.html), 2012.
- [22] 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.
- [23] 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.
- [24] 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/serviceoriented Real-time distributed Computing (ISORC 2008)*. IEEE Computer Society, 2008.
- [25] O. Shivers. <http://www.ccs.neu.edu/home/shivers/citations.html#diss>. Visited August 2012.
- [26] 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.
- [27] H. Søndergaard and A. P. Ravn. Implementation of Predictable Java (PJ) and Safety Critical Java (SCJ). <http://it-engineering.dk/HS0/PJ/>, Visited June 2012.
- [28] 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.
- [29] 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.
- [30] TheOpenGroup. Jsr 302: Safety Critical Java Technology. <http://jcp.org/en/jsr/detail?id=302>, 2006.
- [31] TheOpenGroup. Safety-Critical Java Technology Specification. Draft Version 0.79, TheOpenGroup, May 2011.
- [32] N. Wirth. *Programming in Modula-2*. Springer Verlag, 1985.