Supporting Multiprocessors in the Icecap Safety-Critical Java Run-Time Environment

Shuai Zhao
Department of Computer Science, University of York, UK
zs673@york.ac.uk

Andy Wellings
Department of Computer Science, University of York, UK
andy.wellings@york.ac.uk

Stephan Erbs Korsholm
VIA University College, Horsens, Denmark
sek@via.dk

ABSTRACT

The current version of the Safety Critical Java (SCJ) specification defines three compliance levels. Level 0 targets single processor programs while Level 1 and 2 can support multiprocessor platforms. Level 1 programs must be fully partitioned but Level 2 programs can also be more globally scheduled. As of yet, there is no official Reference Implementation for SCJ. However, the icecap project has produced a Safety-Critical Java Run-time Environment based on the Hardware-near Virtual Machine (HVM). This supports SCJ at all compliance levels and provides an implementation of the safety-critical Java (javax.safetycritical) package. This is still work-in-progress and lacks certain key features. Among these is the ability to support multiprocessor platforms. In this paper, we explore two possible options to adding multiprocessor support to this environment: the "green thread" and the "native thread" approaches. The "native thread" approach is adopted and the design and implementation of a revised icecap SCJ run-time environment is discussed.

Categories and Subject Descriptors: D.3.3 [Programming Languages], K.4.1 [Human Safety]

General Terms: Algorithms, Design, Experimentation.

Keywords: SCJ, Virtual Machine, Multiprocessor System

1. INTRODUCTION

The Safety-Critical Java specification [9] (SCJ) is a specification (JSR-302) created by the Open Group based on the Real-time Specification for Java [6] (RTSJ). In RTSJ, real-time features such as scheduling and memory management are defined to augment the Java run-time environment to make it more suitable for supporting real-time programs. SCJ can be viewed as a subset of RTSJ: it only inherits the necessary features from RTSJ (e.g. priority scheduler) and provides a restricted programming model. However, SCJ also introduces some new concepts such as missions, mission sequencers and compliance levels to facilitate the programming model ─ all of which can be implemented in the RTSJ. Hence, SCJ defines a run-time environment for execution of safety critical Java code. This includes a modified Java virtual machine and a set of base classes that implement the core SCJ programming model in the javax.safetycritical and javax.realtime packages.

JTRES '15, October 07 - 08, 2015, Paris, France

ACM 978-1-4503-3644-4/15/10…$15.00
DOI: http://dx.doi.org/10.1145/2822304.2822305

Each SCJ program consists of one or more missions [20]. A mission represents a specific activity and it encapsulates one or more schedulable objects (schedulables). In SCJ, schedulables are either asynchronous event handlers or no-heap real-time threads. Each mission has a lifecycle that contains three phases: initialization, execution and cleanup. During initialization, all the schedulables created by that mission are initialized and registered by the application. Then this mission enters into the execution phase. In this phase, the registered schedulables will be scheduled to execute by the SCJ run-time environment. If the mission is requested to terminate or all its schedulables finish their executions, the mission will be terminated and cleaned up. During the cleanup phase, memory used by the schedulables will be removed and the mission memory will be cleared.

To control the execution of more than one mission, the concept of a "mission sequencer" is introduced. A mission sequencer is a specialized event handler that creates and controls the sequential execution of a set of missions. The abstract method *getNextMission* provides an interface to application programs to create missions. During the execution of the mission sequencer, the method will be called and the returned mission is executed by the SCJ run-time environment. After the mission is finished and cleaned up, the sequencer will find and execute the next mission. When there are no more missions or the mission sequencer is signaled to terminate, the mission sequencer will wait until the current mission is finished and it will be cleaned up by the SCJ run-time environment.

Along with the introduction of missions and mission sequencers, SCJ makes several changes to the RTSJ memory model. SCJ requires no use of the heap space and removes the garbage collector provided in standard Java. It defines two new scoped memory areas called *mission* memory and *private* memory to facilitate its memory model. In SCJ, each schedulable has its own private memory for object allocation and each mission has an associated mission memory. These memory areas will be created at the point when the schedulable or mission is created and be removed after the component has been cleaned up. The private memory of a schedulable is also cleared of all objects at the end of each release of that schedulable. SCJ also supports the notion of inner private memory, which can be used by a schedulable to execute certain subtasks that wish to create objects that have a shorter lifetime.

SCJ defines three compliance levels. Level 0 is suitable for small and sequential safety critical systems. Level 0 supports cyclic scheduling and a single mission sequencer. In this level, only periodic event handlers are supported and they are executing sequentially. Level 1 provides a priority-based scheduling facility. Along with periodic event handlers, aperiodic and one-shot event handlers are also supported. As with Level 0, Level 1 supports a single mission sequencer, consequently missions are still executing sequentially. However, supporting a priority-based scheduler allows Level 1 to execute the event handlers

concurrently. In addition, synchronized methods are supported in this level.

Level 2 is the most complex level. Among its supported features, the nested mission sequencer is the most important one. In Level 2, the missions created by the initial mission sequencer, which is known as the outer-most mission sequencer, are called "top-level" missions. A top-level mission can create and maintain one or more inner mission sequencers, which can create child missions. As with top-level missions, child missions can also create inner sequencers. By doing so, concurrent execution of missions is supported. This feature makes the programming model suitable for developing large and complex safety critical systems. To enhance the concurrency model, Java's *Object.wait* and *Object.notify* methods are also supported. A restricted version of no heap real-time threads is also provided in order to support a thread-based concurrency model.

SCJ Level 0 supports single processor platforms. In Level 1 and 2, multiprocessor platforms are supported. In Level 1, schedulables are fully partitioned between the processors. Thus, a schedulable can only run (in a preemptive priority order) on the designated processor during its entire lifetime. In Level 2, a global preemptive priority scheduling approach is also available, which means that schedulables can migrate between the available processors during execution. Hybrid scheduling, where schedulables are globally scheduled1 between a subset of all the available processors, is also supported in Level 2.

As of yet, there is no official Reference Implementation for SCJ. However, the icecap project [22] has produced a Safety-Critical Java Run-time Environment (SCJ RTE) based of the Hardware-near Virtual Machine (HVM) [23]. This supports SCJ at all compliance levels and provides an implementation of the safety-critical Java (javax.safetycritical) package. Essentially, icecap provides a Java-to-C compilation system although it supports interpretation as well. One important feature provided by icecap is program specialization, which means that only the components in the JDK and other libraries that are being used by the program will be included into the executable file. This feature reduces the memory footprint of the application. In addition, a dependency analysis facility is supported by the icecap tool chain to check whether all the referenced Java classes are provided by the SCJ RTE. Thus, before compiling a Java program, dependency analysis is performed to ensure all the referenced classes are supported, and then only the components being used in the libraries will be included in the compilation and the final executable file.

Currently the icecap SCJ RTE does not support multiprocessor platforms. This is because HVM supports a coroutine-like concurrency model [22], which is a single-threaded approach that regards all the schedulables as green threads. During runtime, HVM will switch between these green threads based on scheduling decisions from the SCJ scheduler. Applying the coroutine approach on uniprocessor platforms is efficient and convenient because the HVM does not need to rely on an operating system and can achieve a high portability. However, as a single-threaded program, the current HVM cannot be used when multiprocessor support is required.

This paper is divided into 7 Sections. Section 2 will describe the current software architecture of the icecap SCJ RTE. Then in Section 3 we will present two possible solutions to make the RTE capable of exploiting multiprocessor platforms. Section 4 will describe the revised software architecture based on a "native thread" approach. The modified RTE is evaluated in Section 5. Related work will be given in Section 6, and Section 7 will draw conclusions and present future work.

2. The ICECAP SCJ RTE

In order to promote portability, HVM is build on top of a Hardware Abstraction Layer (HAL) [22]. The SCJ base classes then sit on top of this and together they provide a full run-time environment for SCJ. With SCJ applications, these four components are embedded in a tool chain for compiling and executing SCJ programs. The whole architecture is shown in Figure 1 (taken from [13]). In this section, we will describe the facilities provided by the HAL and the implementation of the SCJ base classes that are relevant to this paper.

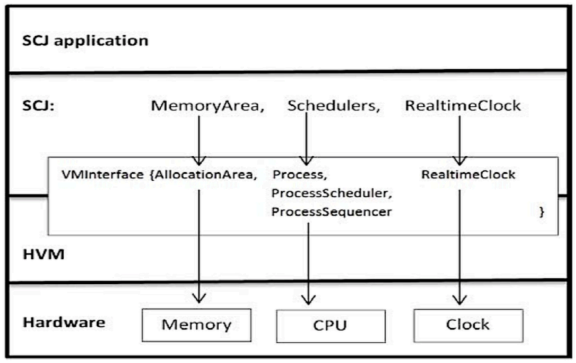

Figure 1: The SCJ and HVM architecture

2.1 Hardware Abstraction Layer

The HVM HAL provides the low-level primitives that are necessary to support the SCJ programming model: such as preemptive scheduling and memory management. One important feature of the HAL is that it is developed in Java [23]. This brings two advantages: high portability and program specialization. In this section, the major facilities provided by the HAL are described. In addition to these facilities, the HAL also provides the following primitives: (1) an interrupt handler to handle the hardware or clock interrupts; (2) hardware objects that are used to control IO devices; (3) a monitor interface to provide synchronization and (4) a real-time clock that binds to a simple hardware clock.

The Process Class

Concurrent execution of SCJ schedulables is facilitated by the *Process* class [23]. A *Process* can be viewed as a concurrent executor of a Java *Runnable* object that can be scheduled by the schedulers implemented in the SCJ base classes. Each *Process* object requires a Java integer array that acts as a stack for the execution of its corresponding *Runnable* object. The *Process* class supports a coroutine model where context switches between processes can only occur through an explicit call to a *transferTo* method by the SCJ RTE. When the *transferTo* method is called with the next process object, the HAL will (1) push the current state of the CPU to its stack; (2) save the stack pointer in the current process object; (3) get the new stack pointer and state from the stack of the next process. When the *transferTo* method returns, it will return to the next process instead of the process that made the original call.

Although the SCJ Run-time environment does not require standard Java threads to be supported, the icecap SCJ RTE does support them when running on top of an OS via calls to the pthread library. Hence, all the SCJ schedulables are run by a single OS thread and each Java thread has its own OS thread.

The Scheduler Interface

The *Scheduler* is an interface to the SCJ base classes for defining different application schedulers (SCJ supports a cyclic scheduler and a priority scheduler). In this interface, an abstract method *getNextProcess* is used to define the required scheduling behavior. At each rescheduling point, the scheduler will get the next process to execute by calling *getNextProcess* and then call the *transferTo* method to switch to that process. The HVM calls the scheduler from the clock interrupt handler at a regular interval. The interface also has methods to define the required behavior of *Object.wait* and *Object.notify*.

The Memory Class

The *Memory* class provides a low-level interface for implementing the SCJ memory management model. Each *Memory* object represents an area of memory and holds three integer variables: base, free and size to indicate the first free location, the current free location and the size of the memory. These are initialized on creation. When HVM starts, it creates a contiguous memory area and every object that is created from Java will be allocated from this memory. HVM does not support garbage collection.

To support the SCJ memory management model, more than one memory area can be created and memory areas created within other memory areas is supported. HVM holds a single global reference *currentMemoryArea* to track the current active memory area. When allocating new Java objects, the HVM allocator will atomically allocate the object into the current area. In addition, the current memory can be changed from Java code by calling the method *swichToArea*.

2.2 The SCJ RTE Base Classes

As we focus on adding multiprocessor support, this section only describes the related facilities in the icecap RTE, which includes the concurrency model, thread synchronization and communication. Further details on the entire SCJ implementation can be found in [14], [19] and [24].

2.2.1 Concurrency Model and Scheduling

This subsection describes two major components that use the HAL facilities to implement the SCJ concurrency model: the *SCJProcess* and the *PriorityScheduler* classes

The SCJ Process Class

An *SCJProcess* is a higher abstraction of a HAL *Process* and it is used to execute the SCJ schedulables concurrently. When each SCJ schedulable is created, a corresponding *SCJProcess* object is created by the base classes. Each *SCJProcess* object contains a HAL *Process* object, which is used to execute the run method of the schedulable; it supports several execution states (e.g. ready, executing and handled) to help the SCJ schedulers implement the required behaviors.

The PriorityScheduler Class

The SCJ base classes define two schedulers: a cyclic scheduler and a priority scheduler. They both indirectly implement the HAL *Scheduler* interface and support different SCJ compliance levels. We will focus on the priority scheduler, which is required to support multiprocessor platform.

The priority scheduler schedules SCJ schedulables according to their priority. In the icecap SCJ RTE, the priority scheduler maintains four queues: a run queue (RQ), a sleeping queue (SQ), a lock queue (LQ) and a wait queue (WQ). The RQ contains all the schedulables that are ready to execute while the SQ has all the schedulables that are waiting for their next releasing time. The LQ and WQ are for synchronization and communication, which will be described in Section 2.2.2. Figure 2 illustrates the scheduling algorithm of the icecap priority scheduler:

At each rescheduling point, the schedulables in the SQ that should be released at that moment (i.e. its next release time is less than the current time) will be moved to the RQ and wait to be scheduled. The priority scheduler will then get the highest priority schedulable in the RQ and make it the new current schedulable. The previous one (if it is not the next one) will be inserted into either the RQ or the SQ depending on its next release time and its execution state.

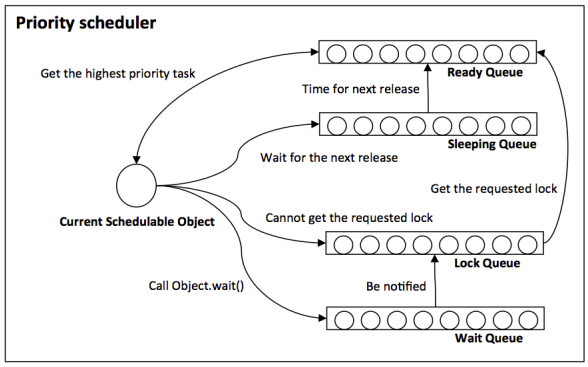

Figure 2: The structure of priority scheduler.

2.2.2 Synchronization and Communication

The *Lock* class implements the *Monitor* interface provided by the HAL. The icecap SCJ base classes provide their own locking and unlocking behaviors to coordinate with the priority scheduler. When a schedulable tries to enter into a synchronized section, it requests the lock. If the lock is available, then the schedulable can obtain the lock and keep executing. Otherwise, the schedulable will be placed in the LQ and the priority scheduler will select the next schedulable to execute. Once a lock is released, the icecap SCJ base classes will inspect the LQ and check whether there are any schedulables that are requesting the lock. If there is more than one schedulable, the schedulable with the highest priority will get the lock. Then this schedulable will be moved into the RQ and wait to be scheduled.

Each application scheduler defines its own behavior for *Object.wait* and *Object.notify* methods by overriding the corresponding method in the *Scheduler* class. The priority scheduler implements these methods using the LQ and WQ. If the current schedulable calls *Object.wait*, it will release the lock and be inserted into the WQ. The schedulable will remain in the WQ until it is notified by another schedulable. When a schedulable calls *Object.notify*, the highest priority schedulable that is waiting for the lock will be placed into the LQ and begins to compete for the lock. Once the schedulable gets the lock, it goes into the RQ and waits to be scheduled.

3. SUPPORTING MULTIPROCESSORS

Section 2 indicated that the underlying concurrency model supported by HVM is based on green threads and coroutines. This is a good model particularly for supporting bare board embedded platforms where there is no underling OS. However, most multiprocessor platforms will have a supported OS and there are OSs certified for safety critical systems (e.g. Integrity® -178B from Greenhills). Hence in this paper we assume the presence of an underlying OS.

There are essentially two approaches to modifying HVM to support multiprocessors based on *green* and *native* threads. With the green threads approach, HVM is modified to allow N threads of control where N is the number of CPUs. Each thread of control has an associated thread in the OS whose affinity is set to that of one of the processor. Each CPU then has its own scheduler that is responsible for determining the thread that is currently active. With native threads, the HVM is modified so that each schedulable is assigned its own OS thread. Scheduling and synchronization is performed by the OS, and SCJ scheduling allocation domains are mapped to OS affinities.

3.1 Green Threads

The green thread approach requires some modification to the current icecap SCJ RTE architecture. To achieve parallelism, a priority scheduler running as a native thread should be created for each processor. Among the schedulers, there can be a chief scheduler provided as part of the mission infrastructure that is responsible for associating the SCJ schedulables to one or more schedulers during the initialization phase of a mission. This approach can be viewed as multiple coroutines running in parallel. In addition, each running scheduler must hold a single reference to the current memory area instead of maintaining a global reference in the *Memory* class as there will be more than one threads that are running in parallel. Figure 3 illustrates the modified icecap architecture based on this approach.

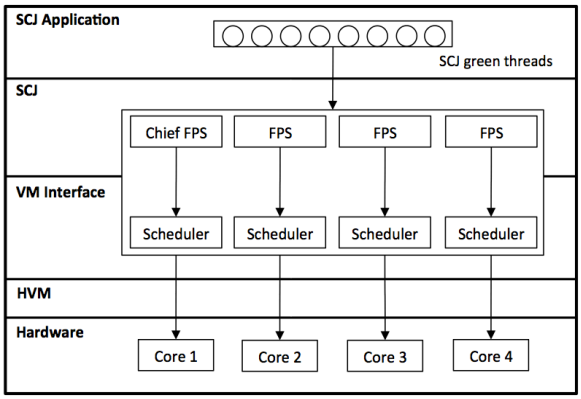

Figure 3: The structure of the green thread approach

For SCJ Level 1, the approach is straightforward to implement as Level 1 defines a fully partitioned scheduling approach (no thread migration). Before the start-up of SCJ programs, the HVM decides the number of schedulers needed based on the number of available processors, which should be provided either by the icecap tool chain (default) or the programmer (customized). A chief scheduler could be assigned to dispatch all the schedulables to each scheduler based on their scheduling allocation domains (affinity sets). After dispatching, the schedulers will start to execute in parallel. Once a scheduler has no threads to be scheduled, it will be stopped by the RTE. When all the threads are terminated, the infrastructure will clean up all the schedulers and then the mission is finished.

However, this approach becomes much more complicated on SCJ Level 2 as threads migration is allowed. To support Level 2 programs, a thread migration routine and a set of migration rules (i.e. when and where a thread should migrate) should be implemented. Then the chief scheduler will need to check the workload of each scheduler during each rescheduling point and migrate the green threads from a heavily loaded scheduler to other schedulers based on the migration rules. During migrating, the scheduler of the thread and target scheduler should be disabled to keep data consistency.

As an illustration, on a four-core platform the HVM will initialize four priority schedulers. When running a SCJ Level 2 program, the chief scheduler, say scheduler 0, will start to execute at the very beginning of the program and dispatch all the running SCJ schedulables to each scheduler (including itself). Then all the schedulers will be started to execute in parallel and schedule its own green threads to execute. In addition, the chief processor will need to check the workload of each scheduler during each interval and migrate schedulables from a heavy load scheduler to other schedulers.

This approach brings two major advantages: (1) it maintains the current HVM coroutine structure and (2) all dispatching is done inside the HVM so that the implementation can potentially be executed in a standalone embedded architecture. Further, it is easier to implement any SCJ-specific scheduling or synchronization semantics (e.g. ceiling locking that might not be available on the underlying OS). Thus, The SCJ RTE facilities described in Section 2.2 will need only minor modifications. In addition, the HVM keeps all the features and remains self-contained: it does not need any support from an operating system or any libraries.

Yet as schedulers are independent of each other (i.e. running in parallel on different cores), thread migration decisions will be difficult to implement and the migration rules are hard to define. In addition, frequent migrations could introduce a high run-time overhead, as the schedulers that are involved may have to be disabled during migrating.

3.2 Native Threads

Instead of maintaining the coroutine model, this approach assumes an underlying real-time (safety-critical) operating system and provides a thick binding from schedulables to RTOS threads. By doing so, the SCJ schedulables will be mapped to native threads and be scheduled by the RTOS. However, applying this approach means that the HVM coroutine concurrent model has to be discarded. As a consequence, the icecap SCJ base classes described in Section 2.2 have to be re-implemented as they assume this model (the memory management model can be kept but the current memory tracking facility has to be changed).

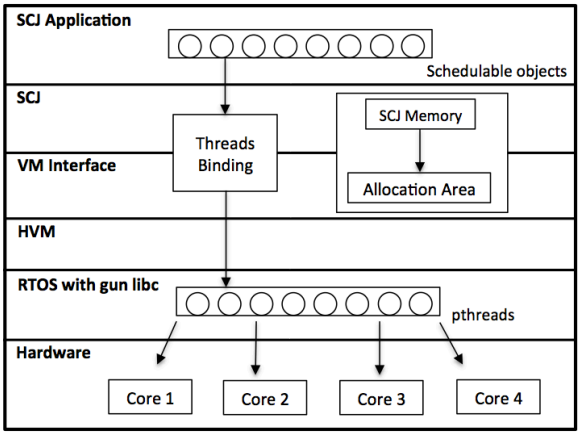

Figure 4: The structure of the native thread approach.

Based on these considerations, we choose POSIX threads (Pthread) to support this binding for two reasons: (1) the POSIX library provides sufficient interfaces that can implement the

required SCJ base classes; (2) the current HVM supports standard Java threads through the Pthread library and provides thread synchronization and communication through *pthread_mutex* and *pthread_cond* interfaces. These can be used directly. Our overall goal is to delegate as much work to the RTOS as possible as long as the SCJ semantics are maintained. We assume, as with the Linux POSIX binding, that all queues are ordered according to the selected scheduling policy, which for SCJ is FIFO within priority. The structure of this approach is shown in Figure 4.

To implement the new structure, the following components are needed:

Support for Schedulables: As described in Section 1, there are several kinds of schedulables and each schedulable has a different execution behavior (e.g. a periodic event handler runs periodically while the one-shot event handler only executes once per release). In the current icecap implementation, supports for these release profiles are intertwined with the coroutine implementation. Therefore, the new structure has to explicitly support the required execution profile for each type of schedulable. In addition, the new structure should be able to set the OS's priority for each schedulable.

An execution model of missions: Once a mission enters into its execution phase, all its schedulables (except aperiodic handlers) will start to execute. Again this is all intertwined with the coroutine model and needs to be re-designed.

A modified memory management model: As schedulables are now running in parallel, each object should hold a reference to its current memory area and update that reference each time when its current allocation context is changed. During allocation, the RTE should always allocate objects in the current memory area of the calling schedulable.

The facility for getting current mission and current schedulable: In the coroutine model, the priority scheduler holds a single reference to the current schedulable and updates the value at each rescheduling point. With the new RTE structure, the current schedulable returned must always be the calling schedulable (as more than one is now executing). Accordingly, if the current schedulable is not a mission sequencer, the current mission returned will always be the mission of the calling schedulable. Otherwise, the mission returned will be the current mission of the mission sequencer.

The scheduling allocation domain (affinity set) facility: The new RTE must support an affinity setting facility and an affinity inheritance model (a newly created SCJ schedulable will inherit the affinity set of the schedulable that creates it). In addition, it should support two ways to set affinity: the default setting (assigned by the RTE) and the customized setting (possibly assigned through a configuration file). For customized settings, an affinity set checking procedure is needed.

As most operating systems now support the POSIX library, this new structure is more portable to different RTOSs. In addition, the powerful interfaces provided in the POSIX library facilitate the implementation of much of the SCJ RTE. However, one disadvantage of this approach is that the new structure has to rely on an underlying RTOS for scheduling purpose, which makes the HVM more difficult to port to a standalone environment.

3.3 Discussion

As most multiprocessor platforms have an underlying operating system, the "native threads" approach becomes a more valid option. The reasons are: the underlying OS provides strong support for thread scheduling, which also includes a set of well-defined migration rules. The green thread approach has no direct support for thread migration. Based on these considerations, we choose the "native threads" approach. In next section, a detailed description of the design and implementation for "native threads" approach will be provided.

4. THE REVISED ICECAP RTE

In this section, we describe the revised icecap SCJ RTE based on the native thread approach. The section is divided into 5 sub-sections, each of which explains the implementation of one of the components described in Section 3.2.

4.1 SCJ Schedulables

To implement the new RTE, the schedulables are first mapped to native threads. The icecap SCJ RTE already provides an interface to pthreads in support of Java threads. Our modified RTE using this for the binding of a schedulable via a new class called *OSProcess*. In addition, an OS timer is used to support the behaviors of a periodic handler and one-shot handler.

OSProcess

The class *OSProcess* has the same function as that of the *SCJProcess* class: executing schedulables. In the new RTE, each schedulable (except the outer-most mission sequencer) has an OSProcess object as its executor. Standard Java threads are supported in HVM and implemented by pthreads, and these can be used directly to execute SCJ schedulables. As SCJ does not support heap memory, there are few differences between a schedulable and a Java thread. The most important here being the priority at which they run. Consequently, a Java thread is created in each OSProcess object and its run method is overridden to execute the corresponding run method of the schedulable. The outer-most mission sequencer has no corresponding OSProcess object and will be executed by the main thread once a SCJ program is started

To execute a schedulable, the RTE will call the *start* method of the corresponding Java thread, which is redefined by the HVM. When the *start* method is called, a function defined in HVM will be called to create and start a pthread though the *pthread_create()* function. The pthread will execute the run method of the Java thread with a stack supplied by HVM.

In addition, to facilitate the execution of SCJ schedulables, each OSProcess object has a structure that contains all the properties of the schedulable, which include the type of thread (e.g. periodic, aperiodic or one-shot), its priority, its start time, period, etc. Before starting a pthread, the attributes of the pthread is assigned by the RTE based on the properties of the corresponding schedulable.

Timer

A one-shot handler has a start time, which specifies a period of time that a handler has to wait after the schedulable is started. In addition, the start time can be reset by the *scheduleNextRelease* method. Periodic handlers have an interval between each release. To implement the executing behaviors of these handlers, the POSIX timer interface is used; this provides a set of functions to create and operate high-resolution timer objects. For each POSIX timer, a structure called *itimerspec* is provided to specify the period of the timer. The function *timer_settime* is called to specify the interval of a timer and the function arms the timer or disarms it (when the interval is 0). Once a timer is armed, the calling schedulable can wait for it to fire.

Accordingly, each periodic handler and one-shot handler in the new structure contains a timer object that specifies the start time and/or period of that handler. For a periodic handler, its periodic timer will initially be armed when the run method of the schedulable is executed. As for one-shot handlers, its timer will be armed before executing the run method. In addition, when the *scheduleNextRelease* method is called with a valid start time, the current start time will be updated to the new one through calling the *timer_settime* function and the one-shot handler will be released again.

4.2 Execution Model for Missions

The mission execution model contains two facilities: an execution routine for schedulables and the termination facility of SCJ schedulables and missions.

Execution of Schedulables

A mission controls the execution of schedulables. The schedulables will be started when the mission enters into the execution phase and be cleaned up during the mission cleanup phase. Here, we use the term "start" to indicate the start-up of the SCJ schedulable executor (i.e. the pthreads) and the term "release" to represent the actual execution of schedulables (i.e. the execution of the *run* method of schedulables). In the SCJ specification, the lifecycle of each type of schedulables is well defined and faithfully implemented by the new structure as shown in Figure 5.

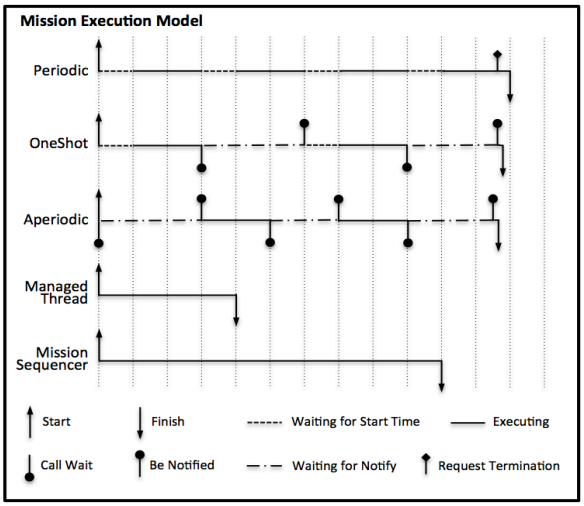

Figure 5: The execution model of missions

When a mission enters into the execution phase:

A periodic event handler will be released at its start time (i.e. either immediately or after a period of time). Then it will be released periodically based on a fixed interval i.e. its period.

A one-shot handler will be released at its start time. After this release, it will wait until the *scheduleNextRelease* method is invoked, which can assign a new start time to the handler. Calls to the *scheduleNextRelease* method once the one-shot handler has been started, result in the next fire time being altered.

An aperiodic handler will not be released unless its *release* method is invoked.

Managed threads and mission sequencers will be released immediately. However, they can only be released once during the entire lifecycle. Once the release is finished, they will wait to be cleaned up by the mission.

All synchronization required to implement the schedulable release patterns is achieved by using the pthread libraries *mutexes* and *condvars*. For instance, the *oneShot* handler in Figure 5 is started and then released after one unit of time, which is set by its start time. After this release, it calls the *pthread_cond_wait* method to wait for the next release signal. When its *scheduleNextRelease* method is invoked with the same start time (one unit) so that it is released again. As for an aperiodic handler, it will call the *pthread_cond_wait* method immediately once it is started. The aperiodic handler will be notified and then released when its release method is called (which will result in a call the *pthread_cond_signal*).

Termination of Schedulables and Missions

The mission termination facility is implemented mainly through the function *pthread_cancel* and *pthread_testcancel* provided by the pthread library. When the *requestTermination* method in the mission class is invoked, a termination request of that mission is pending. Every schedulable inside the mission will check whether there is a pending termination request after each release. If true, the thread will invoke the *pthread_cancel* function to send a cancellation request and then call the *pthread_testcancel* function to set a cancellation point, which will terminate the thread if a cancellation request exists.

When a mission starts to execute, it will maintain an integer value to indicate the current number of active schedulables. Once a mission (actually the mission sequencer) starts all its schedulables, it will call the wait method to wait for the termination of the schedulables. When the last schedulable of that mission is about to finish (i.e. the number of active schedulables equals to 1), it will call the signal method to wake up the mission sequencer so that the mission will enters into the cleanup phase, where the schedulables and the corresponding memory areas will be removed.

4.3 Memory Management Model

The current SCJ memory management model is implemented by both the HVM and the SCJ base classes. In the HVM, an allocator is provided to perform the memory allocation for new objects. Meanwhile, the SCJ classes implement the SCJ memory stack and provide a set of methods for entering or executing in a specific memory area. As described in Section 3.2, this memory model can be applied directly to the new structure with only a modification to the facility that tracks the current memory area, where each SCJ schedulable should hold a reference to its current memory area.

The "pthread-specific data" interface is used to implement this modification. Pthread-specific data is similar to Java thread-local data, where each thread can have its own value on a single variable. In the pthreads, each thread-specific data is linked to a *pthread_key* value, which is visible to all threads in that process. Although the same key is shared with all the threads, the values that are bound to the key can be different between threads. In addition, each individual thread can change or obtain its own value through the key by calling the *pthread_setspecific* and *pthread_getspecific* function respectively.

Accordingly, the current memory area of each thread is stored through a thread-specific data key and is maintained by each individual thread. The current memory area of a specific thread will be updated through the function *pthread_setspecific* each time the thread enters, executes in, or leaves a memory area. In addition, when the allocator in HVM is called, it will get the current memory area of the calling thread through the

pthread_getspecific function and allocate the object in that memory.

4.4 Tracking the Current Schedulable

In the revised structure, SCJ schedulables are bound to pthreads to achieve parallelism. However, as an executor, the pthread does not know which schedulable it is executing. Thus, a thread tracking facility is needed for obtaining the current running SCJ schedulable.

In the original coroutine structure, each schedulable is assigned with a unique index and a tracking facility is provided to get a specific schedulable based on its index value [24]. The index of each schedulable represents the position of the schedulable and its mission, which are stored in a Java array. This facility was designed for the priority scheduler so that it can get the next schedulable merely by an integer value with the time complexity of order 1. This tracking facility and the thread-specific data interface are utilized to implement the new schedulable tracking facility. Because each running pthread in our structure represents a schedulable, the current schedulable should always be the calling thread itself.

To get the calling schedulable, a thread-specific data key is provided for storing the index values of all threads. Then the index value will be passed from the SCJ classes to HVM by the OSProcess object when the start method is called (as described in Section 4.1) and this will be set through the thread-specific data key at the very beginning of the execution of a thread. Thus, upon a call to get the current schedulable, the index of the calling thread can be returned to the infrastructure by calling the *pthread_getspecific* function with the appropriate key value. After the index value is obtained by the SCJ classes, the tracking facility mentioned above will be used to get the schedulable with same time complexity. Thus, the current schedulable is obtained. The *getCurrentMission* method is also implemented with support from this tracking facility.

4.5 Affinity Sets

Affinity sets are used by the SCJ to represent scheduling allocation domains. This in turn allows SCJ programs to execute with different scheduling schemes (i.e. partitioned, global or hybrid). Affinity sets are not supported by the current icecap project as its focus is on simple processor systems. The modified icecap SCJ RTE provides a default affinity setting and also supports customized affinity settings. The implementation of this feature includes a default affinity setting for all SCJ levels and a checking facility for customized settings. Note that currently there is no POSIX standard that addresses multiprocessor issues; consequently there is no standard API for affinity sets. Our revised implementation of the icecap SCJ RTE is targeted at Linux, so we use the Linux API here.

To implement the SCJ model, the *AffinitySet* class is implemented first. This class contains a set of processor ids that represents the available processors a schedulable can execute on. This class also maintains a static array that contains all the affinity sets created by the RTE. The static array will be obtained when the *getSchedulingAllocationDomains* method is called. Additional affinity sets can be created in Level 2 programs by calling the *generate* method, which will generate an affinity set that contains only one valid processor.

In addition, a reference of affinity set object is added to each schedulable and this will be initialized in the constructor. The affinity set of a newly created schedulable inherits the affinity set of the schedulable that creates it. However, the affinity set of a schedulable can be changed by calling the *setProcessorAffinity* method during the mission initialization phase.

When calling the start method of a SCJ schedulable, its affinity set (a Java array) will be mapped to a *cpu_set_t* structure in HVM and this will be set to the corresponding executor pthread by calling the *pthread_attr_setaffinity_np* function. Thus, the pthread will have the same affinity set as that of its corresponding schedulable.

Default Affinity Setting

The SCJ specification requires the SCJ RTE to provide a set of default affinity settings for each compliance level. In the revised icecap tool chain, before launching an SCJ program, the RTE will check whether a customized affinity setting is provided in the configuration file. The defaulting setting will be activated if no customized setting is provided.

Level 0 only directly supports uniprocessor platforms. The RTE will only create one affinity set with a single processor. Before the start-up of SCJ programs, the HVM will get the id of the processor that the HVM runs on by calling the *sched_getcpu* function and set the affinity of the thread to that processor. Then the processor id will be passed to the base classes for creating the affinity set in Java space. This affinity set will be assigned to every schedulable contained in the SCJ program.

For Level 1 programs, a fully partitioned scheduling scheme is applied. Thus, the number of affinity sets created by the RTE will be the number of available processors, which can be obtained by the *get_nprocs_conf* function. Each affinity set will contain one unique processor. By default, every schedulable will have the same affinity as that of the outer-most mission sequencer because of the affinity inheritance rule. However, in Level 1, the affinity of each individual schedulable can be changed through the *setProcessorAffinity* method.

SCJ Level 2 allows a global/hybrid scheduling approach, which means that the schedulables can execute on any available processors in the platform by default. As a default, the icecap SCJ RTE will only create one affinity set that contains all the available processors in the platform and set this affinity to each schedulable created. In addition, the *generate* method is supported in this level, which can create an affinity set with one valid processor. Thus, schedulables in Level 2 can be set to execute only on a specific processor by calling this method and the *setProcessorAffinity* method during the mission initialization phase.

Customized Affinity Setting

In addition to the default setting provided by the RTE, the affinity sets for all three levels can be defined by users. For instance, users can specify the processor to execute the SCJ Level 0 program. At Level 1 and 2, the processor or processors that will be used for execution can be specified. To support this feature, a *Configuration* class is added for receiving the customized setting and a checking model is added to ensure the setting is valid.

The *Configuration* class provides a two-dimensional integer array with a default value of null, where one dimension represents the processor or processors of an affinity set. To set the affinity, users should assign the desired value to this array. For instance, the values {{1}, {2}} indicates the SCJ program is fully partitioned (with two schedulable allocation domains) and will only execute on processor 1 and 2 while the value {{0, 1, 2}} means that the program is globally scheduled on processor 0, 1 and 2. Similarly, if the user requires hybrid scheduling, they will

set the array to, for example, {{0,1}, {2,3}}. This indicates that the application requires two scheduling allocation domains; the first contains the processors 0 and 1, and the second contains processors 2 and 3.

The RTE will check this array when starting an SCJ program. If its value is not null, the infrastructure will then enter into a checking routine to check whether the setting is valid. The checking routine mainly includes two parts:

- The processors assigned should be consistent with the scheduling of the compliance level. For instance, the value {{0}, {1}} in a level 0 program or the value {{0, 1, 2}} in a level 1 program will result in an exception being thrown as they violate the scheduling defined for that level.
- Each processor id assigned in the array should be a valid processor id and only be contained in one scheduling allocation domain. The RTE will get a list of all the available processor ids and then perform a comparison to check whether the assigned processor is valid. If an invalid processor id is detected, an exception will be raised.

5. EVALUATION

As a Java Specification Request, the SCJ should contain a technology compatible kit (TCK), which provides a test suit to verify whether a proposed SCJ implementation meets the requirements defined in the specification. However, currently the TCK is still under development and not available [17]. The only test available now is a set of test cases provided by the icecap team. To evaluate the revised SCJ RTE, new test cases are created to test individual modules. Then a regression test and an integration test are conducted to check whether new errors have been introduced to the existing HAL and SCJ RTE, and whether the new modules can work together properly. All of the test cases that being created or used to test against the RTE can be found at https://github.com/scj-devel/hvm-scj/tree/master/icecaptoolstest. The hardware specification of the machine that being used to perform the tests is described below:

- 2.4 GHz Intel Core i7 (4 cores)
- 8 GB 1600 MHz DDR3
- NVIDIA GeForce GT 650M 1024 MB

5.1 Module Testing

Eleven new SCJ programs have been created to test the revised RTE. These check the correctness of each module and whether the implementation meets the requirements in the specification. The execution of schedulable objects and missions, the multiprocessor memory model, thread synchronization, the wait and notify facility, nested mission sequencers and affinity set are tested.

5.1.1 Execution of SCJ Threads and Missions

The execution of schedulable objects is tested through a Level 2 program, which contains a periodic handler, an aperiodic handler, an oneShot handler and a managed thread. During execution, each schedulable object should print only one message to the console to indicate a successful release. The expected behaviors of these schedulable objects are described below:

- The periodic handler is released automatically so that its message should be printed regularly based on its period until the mission is terminated.
- The OneShot handler should print its message once when the mission starts, then its message is printed each time the *scheduleNextRelease* method is invoked with a valid start time.
- The aperiodic handler should print its message only when its *release* method is invoked.
- The message of the managed thread should be printed once during the entire lifetime of a mission.

5.1.2 Memory Model and Thread Tracking Facility

The modified memory management model and the thread tracking facility are tested through a Level 1 program with two periodic event handlers. For each handler, the *base* and *size* of its current memory area will be obtained at each release through a native method, which calls the *pthread_getspecific* function to get the current memory and then returns the *base* and *size* value. Then these two values are compared with the corresponding values of the handler's private memory. As each memory area has a unique pair of *base* and *size,* the corresponding values of these two memory (the private memory and the current memory) areas should always be the same in our case, which indicates that the modified memory model is working properly and meets the requirements i.e. support multi threads running in parallel.

Along with testing the memory model, the current schedulable object is obtained through calling the *getSchedulableObject* method by each handler on each release. Then the *index* of the current schedulable object is compared to the *index* value of that handler. As the current schedulable object returned should always be the calling thread, the *index* obtained through the *getSchedulableObject* method should always be the same as that of the calling schedulable object.

5.1.3 Thread Synchronization and Communication

As the wait and notify method interacts with locks, the SCJ synchronization and communication facilities can be tested together. A Level 2 program with two periodic event handlers is created to test these two facilities.

In the testing program, the first periodic event handler calls the wait method at the beginning of each release. Then the second periodic event handler calls notify to wake up the first handler so that the first handler can resume executing. The wait and notify method are invoked inside a synchronized method, which are protected by a SCJ monitor.

During execution, the first periodic handler prints a "before wait" message before the wait method is invoked and then prints "after wait" when it wakes up. The second handler prints "after notify" message after the notify method is called. Accordingly, the "after wait" message should always be printed after the "after notify" message is printed, which indicates a successful execution of the wait and notify method.

5.1.4 Affinity Set Model

The testing of the affinity set implementation covers the default affinity setting provided by the RTE and the customized affinity settings checking facility. In addition, this testing can also reveal whether the new structure supports multiprocessor platforms.

Default Affinity Setting

Two test cases are created for testing the default affinity settings of Level 1 and 2. As we use a four-core platform, the Level 1 testing contains four periodic event handlers and each handler runs on a unique processor. During execution, each handler gets the processor that it is executing on through the Linux *sched_getcpu* function. Then the handler compares this processor with the processor that was assigned by its affinity set. These two processors should always be the same. Otherwise, the testing program throws an exception.

As Level 2 supports global scheduling, the testing program mainly checks whether a schedulable can migrate to other processors. In the Level 2 testing program, five periodic handlers are created and each will perform a busy-waiting on each release. The properties of these handlers are shown in Table 1. As shown in this figure, the first four handlers are scheduled fully partitioned on each processor respectively with the same period and priority. However, their start times are different: handler 1 is the first that will be released immediately and handler 4 will be released after 30 milliseconds. As for handler 5, it will be released immediately and has a very short period (1 millisecond). In addition, it has a lower priority and can execute on all the processors in the platform.

Handler	Start	Period	Priority	Affinity
1	0	40	50	0
2	10	40	50	1
3	20	40	50	2
4	30	40	50	3
5	0	1	10	0, 1, 2, 3

Table 1: Handler properties in affinity testing program

Thus, the first four handlers will execute on their assigned processors at different times. Accordingly, handler 5 has to migrate between these processors to execute because it has a lower priority than the other handlers (cannot preempt) and a shorter period (released more frequently). During execution, handler 1 to 4 will print the message "h" to indicate their release, and handler 5 will print the current processor id that it is executing on. The processor id that should be printed should be one of the processors in its affinity set and should vary over time.

Customized Affinity Setting

The testing for the customized affinity setting mainly checks whether the customized setting can be activated correctly and whether the checking routine works properly through two test cases. The first program assigns various valid affinities through the configuration file. During execution, each handler will check whether its available processors are consistent with its customized affinity set by calling the Linux *sched_getcpu* and the *get_nprocs_conf* functions. In contrast, the second program assigns various invalid affinities that either contains an invalid processor id or is not consistent with the compliance level. Then the program checks whether the SCJ RTE can detect these invalid settings and respond with appropriate error messages.

5.2 Regression and Integration Testing

The test cases provided by the icecap project aims to test the entire tool chain described in Section 2, which includes the HVM, the HAL and the SCJ base classes. Among the test cases, several SCJ test programs are provided to verify the SCJ implementation of Level 1 and 2. Although this is far from a TCK for SCJ, these test programs are used to perform a regression test.

The regression test of the revised SCJ RTE mainly verifies that the revised architecture is consistent with the specification and that the implementation has not introduced errors to the existing HAL and the SCJ base classes. The test cases are either Level 1 or Level 2 application programs. Level 1 test programs mainly test against the thread synchronization i.e. SCJ monitors. Level 2 programs targets the features of nested mission sequencers and managed threads (including the waits and notify facilities).

The integration test aims to test whether there are any conflicts between the newly implemented facilities. The integration test is performed by a Level 2 program, which utilizes most of the features provided by the new structure together in one program. This program contains 3 inner mission sequencers. During execution, all the nested mission sequencers are executed by the top-level mission so that their inner missions are executed in parallel. Each inner mission contains two periodic event handlers that invoke the wait and notify methods respectively.

6. RELATED WORK

The Real-time Specification for Java (RTSJ) was the first Java Specification Request. RTSJ is considered not appropriate for developing safety-critical systems, as some of its features are quite complex. Accordingly, proposals for simplifying RTSJ have been created. Puschner and Wellings [21] created the first simplified version of RTSJ called Ravenscar Java [15] based on the concepts from the Ada Ravenscar tasking profile [4]. Ravenscar Java introduced the novel notion of mission and an initialization phase, where all schedulable objects are set up for execution. Later on, Søndergaard et al. created an implementation of Ravenscar Java [12], which targeted industrial applications on an aJ-100 processor, which can execute Java byte codes directly.

In 2008, the open group created the first draft of the Safety Critical Java Specification as another simplified RTSJ. Since then, international efforts have been made to create virtual machines that can execute Java safety-critical programs and SCJ implementations. Besides the HVM and its SCJ base classes provided by icecap, the virtual machines that can execute SCJ programs include JamaicaVM [11], Fiji VM [10], HVM_{TP} [16], PERC Pico [8, 18] and OVM [1].

The JamaicaVM is a hard real-time [3] Java virtual machine with a static compiler and a deterministic garbage collector. Under JamaicaVM, Java applications are translated into C code and then complied into machine code. As JamacicaVM is built for real-time Java programs, it supports SCJ programs as well.

The Fiji VM is design for general real-time Java programs but can execute SCJ program as well. It precompiles Java bytecode to C for better performance [10].

HVM_{TP} is a modified version of the icecap HVM. It mainly modifies the HVM byte code interpreter to obtain a better predictability and to assure each byte code can be executed within a constant time period.

PERC Pico was produced by Atego [8] according to an early notion of safety-critical Java programs. In PERC Pico, the memory model is implemented and checked through a Java metadata annotation approach, which makes the model different from standard SCJ. However, its newer version also provides SCJ compatibility [7].

Similar to HVM, the OVM virtual machine uses ahead-of-time compilation; in this case the intermediate language is C++. The initial objective of OVM was to support RTSJ programs. However, it can execute SCJ programs as well. A Level 0 SCJ implementation is provided for OVM by Plsek et al [5].

7. CONCLUSION AND FUTURE WORK

In this paper, we introduced two options for adding multiprocessor support to the icecap SCJ RTE: the "green thread" approach and the "native thread" approach. We provide a detailed analysis towards each approach with their features, advantages and disadvantages. Then a prototype implementation based on the "native thread" approach is carried out with a description of each individual module. Finally, an evaluation is performed against the revised icecap SCJ RTE. Based on our evaluation, the revised

RTE successfully supports multiprocessor platforms with help from a Linux real-time operating system. The focus of the evaluation has been compliance rather than performance.

Although both RTSJ and SCJ target multiprocessor platforms, neither supports multiprocessor-aware resource control policies. Instead they rely on the use of the priority ceiling protocol. A review of multiprocessor resource sharing protocols in [2] shows that currently there is no agreed best practice for multiprocessor platform despite the fact that various protocols have been proposed: the reason is that these protocols either impose strong restrictions on nested resource access or carry high run-time overheads. A new protocol named "Multiprocessor resource sharing Protocol" [2] (MrsP) seems to be an effective choice. The most attractive feature of MrsP is that it is compatible with Response Time Analysis (RTA) [3]. In addition, MrsP carries a helping mechanism that improves the performance of spin locks (the spinning task will use its waste cycles to help other tasks) and nested resource access is allowed. According to [2], MrsP has been designed for fully partitioned systems for resources with short critical sections.

Our next objective is to implement MrsP in icecap SCJ RTE for both Level 1 and 2. For SCJ Level 1, we will mainly test the performance of this protocol as the paper has a clear description of its behavior under partitioned systems. However, [2] barely describes the behavior of MrsP under globally scheduled and hybrid system. Thus, for level 2, we will analyse whether MrsP remains compatible with RTA as well as its run-time performance. Resources with long critical sections will also be addressed.

8. REFERENCES

[1] A. Armbruster, J. Baker, A. Cunei, C. Flack, D. Holmes, F. Pizlo, E. Pla, M. Prochazka, and J. Vitek, A Real-time Java virtual machine with applications in avionics. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(1):5:1-5:49, Dec. 2007.

[2] A. Burns and A. Wellings, A schedulability compatible multiprocessor resource sharing protocol - MrsP, in ECRTS'13, 2013, pp. 282-291.

[3] A. Burns and A. J. Wellings. Real-Time Systems and Programming Languages (Fourth Edition). Addison-Wesley Longman, 2009.

[4] A. Burns, B. Dobbing, and G. Romanski. The Ravenscar tasking profile for high integrity real-time programs. *In Proceedings of the 1998 Ada-Europe International Conference on Reliable Software Technologies*, pages 263-275. Springer-Verlag, 1998.

[5] 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, ser. JTRES '10. New York, NY, USA: ACM, 2010, pp. 95-101.

[6] A. Wellings. Concurrent and Real-Time Programming in Java. John Wiley & Sons, 2004.

[7] Aonix. Perc Pico User Manual. http://research.aonix.com/ jsc/pico-manual.4-19-08.pdf April 2008.

[8] Atego, Atego PERC Pico – Products – Atego. http://www.atego.com/products/atego-perc-pico/, 2015.

[9] D. Locke, B. S. Andersen, B. Brosgol, M. Fulton, T. Henties, J. J. Hunt, J. O. Nielsen, K. Nilsen, M. Schoeberl, J. Tokar, J. Vitek, and A. Wellings. Safety-critical Java specification, version 0.95. December 2012.

[10] Pizlo F, Ziarek L, Blanton E, et al. High-level programming of embedded hard real-time devices[C]//Proceedings of the 5th European conference on Computer systems. ACM, 2010: 69-82.

[11] F. Siebert, "Hard real-time garbage collection in the Jamaica Virtual Machine," in Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99), Hong Kong, 1999.

[12] H. Sondergaard, B. Thomsen, and A. P. Ravn. A Ravenscar Java profile implementation. *In Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems (JTRES 2006)*, pages 38-47, Paris, France, October 2006. ACM Press.

[13] H. Søndergaard, S E. Korsholm, and A. P. Ravn. Safety-critical Java for low-end embedded platforms. In M. Schoberl and A. Wellings, editors, *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, ACM, 2012.

[14] J. Hunt and K. Nilsen. Safety-Critical Java: The mission approach. In M. T. Higuera-Toledano and A. J. Wellings, editors, *Distributed, Embedded and Real-time Java Systems*, pages 199–233. Springer US, 2012.

[15] J. Kwon, A. J. Wellings, S. King, Ravenscar - Java: A High Integrity Profile for Real-time Java. Proceedings of the Joint ACM Java Grande Conference. ACM press, New York, 2002

[16] K. S. Luckow, B. Thomsen, and S. E. Korsholm. HVMTP: A Time Predictable and Portable Java Virtual Machine for Hard Real-Time Embedded Systems. *In Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '14, pages: 107:107-107:116, New York, NY, USA, 2014. ACM.

[17] L. Zhao, D. Tang, and J. Vitek. A technology compatibility kit for safety critical Java. *In Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 160 –168. ACM, 2009.

[18] M. Richard-Foy, T. Schoofs, E. Jenn, L. Gauthier, K. Nilsen. Use of PERC Pico for Safety Critical Java, *Conference Proceedings: Embedded Real-Time Software and Systems*, Toulouse, France, May 2010.

[19] M. Schoeberl, H. Søndergaard, B. Thomsen, and A. P. Ravn. A profile for safety critical Java. In ISORC, pages 94–101. IEEE Computer Society, 2007.

[20] M. Schoeberl. Mission modes for safety critical Java. *In Proceedings of the 5th IFIP WG 10.2 international conference on Software technologies for embedded and ubiquitous systems*, SEUS'07, pages 105–113, Berlin, Heidelberg, 2007. Springer-Verlag.

[21] P. Puschner and A. J. Wellings. A Profile for High Integrity Real-Time Java Programs. *In 4th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, 2001.

[22] S. Korsholm, H. Søndergaard, and A. Ravn. A real-time java tool chain for resource constrained platforms. Concurrency and Computation: Practice & Experience, 2013:1--25, September 2013.

[23] S. Korsholm. Java for Cost Effective Embedded Real-Time Software. PhD thesis, 2012.

[24] S. Zhao, Implementing Level 2 of Safety-Critical Java, thesis, 2014, available from https://www.cs.york.ac.uk/library/proj_files/2014/MScComp/zs673/Final%20Project%20Report.pdf