An Evaluation of Safety-Critical Java on a Java Processor

Juan Ricardo Rios
Department of Applied Mathematics and
Computer Science
Technical University of Denmark
Email: jrri@dtu.dk

Martin Schoeberl
Department of Applied Mathematics and
Computer Science
Technical University of Denmark
Email: masca@dtu.dk

***Abstract*—The safety-critical Java (SCJ) specification provides a restricted set of the Java language intended for applications that require certification. In order to test the specification, implementations are emerging and the need to evaluate those implementations in a systematic way is becoming important.**

In this paper we evaluate our SCJ implementation which is based on the Java Optimized Processor JOP and we measure different performance and timeliness criteria relevant to hard real-time systems. Our implementation targets Level 0 and Level 1 of the specification and to test it we use a series of micro benchmarks, an application-based benchmark, and a reduced set of a SCJ technology compatibility kit. We evaluate the accuracy of periods, linear-time memory allocation, aperiodic event handling, dispatch latency for interrupts, context switch preemption latency, and synchronization.

***Index Terms*—Real-time systems, Embedded systems, Java, Safety-critical systems, Safety-critical Java, Java processor**

I. Introduction

The safety-critical Java (SCJ) profile is being developed under the Java specification request 302 (JSR-302). It provides a smaller, tightly defined subset of the Java programming language intended for applications that need to be certified. SCJ uses a programming model that limits how a developer can structure an application. SCJ's programming model is more restricted in terms of concurrency, memory, synchronization, input and output, clocks and timers, and exception processing [10]. Three levels of compliance with different degrees of complexity are defined, namely Level 0 (L0), Level 1 (L1), and Level 2 (L2), L2 being the most complex.

To the best of our knowledge, currently there are only three implementations of SCJ that target embedded systems: (1) the oSCJ [12] implementation running on the Open Virtual Machine [2], (2) the Hardware-near Virtual Machine (HVM) [19] implementation, and (3) the Java Optimized Processor (JOP) [18] implementation. In addition, many use cases have been developed in order to either test an implementation [21] or to evaluate analysis tools [1]. It is therefore becoming important to evaluate SCJ implementations in a systematic way in order to measure different performance and timeliness criteria relevant to hard real-time systems.

In this paper we present an evaluation of our SCJ's implementation [18] running on top of the Java optimized processor (JOP) [15]. We evaluate two main categories: (1) performance and timeliness and (2) compliance to the SCJ specification. For the performance and timeliness category we used the miniCDj benchmark, an application-based benchmark, and we developed specific benchmarking methodologies were we evaluate (1) the accuracy of periods (i.e. release jitter), (2) linear-time memory allocation, (3) aperiodic event handling, (4) dispatch latency for interrupts, (4) context switch preemption latency, and (5) synchronization. In the compliance category we test how good our implementation adheres to the SCJ profile by using an early work on a technology compatibility kit (TCK) for SCJ developed at Purdue University. As our test platform we use an instance of JOP running in a Cyclone II FPGA at 60 MHz.

The remainder of this paper is organized as follows: Section II provides background on safety-critical Java. Section III presents work related to test and evaluation of embedded real-time and safety-critical Java implementations. Section IV provides a summary of our SCJ implementation [18]. Sections V and VI present the details and results of our evaluation. We conclude in Section VII.

II. Safety-Critical Java

SCJ's programming model is based on the execution of missions. Missions encapsulate modes of operation and separate the execution of time-critical from non time-critical operations. They consist of a bounded set of registered managed schedulable objects (MSO) executed either as a cyclic executive (L0) or under the control of a fixed-priority preemptive scheduler (L1, L2). MSOs are created with parameters that cannot be changed at run-time. Such parameters define a real-time priority, specify the nature of the MSO's execution (periodic or aperiodic), and binds the MSO to a memory area.

Depending on the required type of real-time activity, a MSO can be a *periodic event handler* (PEH) or an *aperiodic event handler* (AEH). The use of event handlers is preferred to the use of threads because all the activities for an individual release are encapsulated in a single method [23], namely the handler's `handleAsyncEvent()` method. Threads are only available at L2 in the form of *managed threads* that are a restricted version of RTSJ's `NoHeapRealTimeThread`. Managed threads are useful to implement activities with an execution pattern different from that of an aperiodic or periodic task, e.g. background activities that run at all times.

Memory management is based on memory areas free of garbage collector interactions. Depending on the intended life of the objects allocated in a memory area, there can be *immortal*, *mission*, or *private* memories. Immortal memory is used to store objects that will live for the whole execution of the application, mission memory holds objects that belong to a specific mission and private memory holds objects used only by a single MSO. From the point of view of a MSO, the different memory areas are logically organized as a linear stack where memory areas holding longer-lived objects are nested at deeper levels. Due to the nesting hierarchy and the different life times, objects cannot refer to each other in an unrestricted manner, as it is the case in standard Java. Objects allocated in a memory area can only store references to objects allocated either in the same or in an outer nested memory area. Objects allocated in immortal memory can be accessed by any MSO and objects in mission memory are accessible only to the MSOs belonging to the mission.

III. Related Work

Corsaro and Schmidt provide in [4] an evaluation of two real-time Java implementations. They compare the Timesys RTSJ reference implementation [22] with their own RTSJ implementation, namely jRate. In that work, Corsaro and Schmidt developed RTJPerf, a synthetic, workload-based benchmark to test time efficiency in an RTSJ compliant (for v1.0.1 of RTSJ at that time) JVM. They provide tests to measure linear time memory allocations, the delay to service asynchronous events, overhead on thread switching and preemption, and the accuracy of timers. A very similar study done by McEnery et al. in [11], provides an empirical evaluation of two main-stream, commercial implementations of RTSJ. They compare Sun's (now part of Oracle Inc.) Java RTS and Aicas' JamaicaVM. McEnery et al. assess the efficiency and predictability of the two commercial implementations by providing tests for memory allocation, thread management, synchronization, and asynchronous event handling.

From these two works we found that some tests, e.g. the linear-time memory tests, are relevant in the context of SCJ and can be adapted to our current evaluation. However, there are other tests that (1) are not applicable to SCJ e.g. the latency to dispatch unbounded asynchronous event handlers from the firing of an event, and (2) use features not allowed in SCJ e.g. RTSJ's `RealtimeThread` and self-suspending methods.

In [5], Doherty provides a benchmark to test RTSJ-based real-time Java implementations. His benchmark, SPECjbb2005rt, based on the SPECjbb2005 benchmark [20], is enhanced to provide throughput and response time metrics. However, SPECjbb2005rt focuses on soft real-time applications and does not make use of RTSJ's features intended for hard real-time systems such as `NoHeapRealtimeThread`s and does not implement tests that use immortal or scoped memory, which are integral parts of SCJ. Moreover, SPECjbb2005rt has not been made publicly available nor has it been adopted by the SPEC corporation. Nonetheless, Doherty's work emphasizes the need for standardized benchmarks that can be used to compare different implementations of (hard) real-time JVMs.

Instead of providing a collection of tests that independently evaluate embedded real-time Java features, in [6], Kalibera et al. present the CDx benchmark, an open source, application-based benchmark that can be adapted to run both on standard and RTSJ compliant VMs. CDx can be used for soft and hard real-time systems as it uses RTSJ's hard real-time features. CDx has one periodic thread used to detect potential aircraft collisions, based on simulated radar frames. The metrics that CDx provides are response time, computation time, and jitter for the collision detector periodic thread. A refactored version of CDx, miniCDj, that uses the SCJ API was developed and used in [12] to test Purdue's L0 SCJ implementation. The miniCDj benchmark was developed for v 0.76 of JSR-302 and the current version of JSR-302 is 0.94. Therefore, some modifications were necessary to run it in our implementation.

Any JSR requires a technology compatibility kit (TCK) and in [24], Zhao et al. describe their initial work towards developing a TCK for SCJ where focus was on functional and behavioral tests. The test cases were derived from SCJ's specification to evaluate features such as the mission life cycle, concurrency and scheduling, memory, clocks and timers, and exceptions. We will use a subset of the tests described in their work, as not all of them can be applied to our implementation e.g. tests that evaluate L2 features.

IV. Safety-Critical Java on JOP

For the evaluation we use the SCJ implementation [18] on top of the Java processor JOP [15]. We reuse the already available infrastructure for thread scheduling to implement the SCJ handlers. The original version of JOP uses a garbage-collected heap but for the SCJ implementation the garbage collector is substituted by an implementation of scoped memories. Our test platform is the Altera DE-2 70 evaluation board with a Cyclone II FPGA running at 60 MHz, external SRAM access latency of 3 cycles, and 4 kB method cache with 32 blocks.

A. Scope Memories

To allow a fragmentation free implementation of scoped memories, an SCJ application has to specify the sizes of immortal memory, mission memory, and all private memories per handler. Furthermore, the memory areas have a unique nesting relation. Mission memory is an inner memory of immortal, and private memories are inner memories of mission memory. According to the RTSJ notion of scoped memories, the object that represents the memory area is allocated in the outer memory. However, the size, given in the constructor of a scoped memory does not include the memory requirements of the nested scopes. Therefore, the actual memory, the backing store, is not nested in RTSJ style scopes.

However, with the information of the maximum sizes of nested scopes an *implementation* of the memory area can actually use real nesting of the backing store. We use this property in a single class that can be used to represent all three different SCJ memory types: immortal, mission, and private [16].

The class Memory is a system class and is used to implement ImmortalMemory, MissionMemory, and PrivateMemory. The SCJ memory classes still extend the RTSJ memory classes, but the actual implementation of the functionality is delegated to the Memory class. The user visible API of SCJ is unchanged.

The memory areas form a strict hierarchy with respect to lifetime, which can also be represented as nesting levels. These nesting levels are static. One can assign level 0 for immortal memory, level 1 for the initial mission memory, level 2 for the initial private memory, and so on. These levels, statically assigned to each memory area, can be used to simplify the reference assignment checks [13]. A reference field of an object at level n may only point to an object allocated at level $m \leq n$. Static fields are allocated in immortal memory and therefore, their level is 0.

B. Thread Scheduling

SCJ has the notion of handlers that are released either periodically or on an event. JOP's runtime infrastructure supports a simple form of real-time threads and event handlers. This infrastructure is reused to implement the SCJ handlers.

The thread scheduler in the JVM of JOP is invoked (1) on a timer interrupt, (2) when a thread finishes it's release (with the invocation of waitForNextPeriod()), and (3) when an event handler is fired. The scheduler performs three functions: (1) find the next thread to dispatch, (2) reprogram the timer interrupt, and (3) dispatch the thread. The thread with the highest priority, which is ready (it's release time is now or already passed), is selected for dispatch.

To find the time for the next timer interrupt, the priority ordered list of threads is searched. A thread shall only interrupt the currently dispatched thread, when its priority is higher than the dispatched thread. Therefore, only higher priority threads are searched. Within this set of threads, the thread with the *nearest* release time determines the next timer interrupt. That release time is used for the next scheduling interrupt.

The threads are collected in a priority ordered array. There is no explicit run queue or other queues. However, the run queue is implicit: each thread that has a release time now or in the past is in the run queue. The first thread found in the priority ordered array is dispatched. This gives short dispatch time for high priority threads and longer for lower priority threads, which is acceptable as high priority threads have shorter deadlines when priorities are assigned deadline monotonic. Threads check their deadlines and update their release time on a call to the waitForNextPeriod() method. The implementation of the SCJ handlers uses the original real-time threads from JOP (RtThread) [14].

C. Interrupts

SCJ (and the next version of RTSJ) has the notion of first-level interrupt handlers. An interrupt handler has to extend InterruptServiceRoutine and has to implement the interrupt handling code in the handle() method. The InterruptServiceRoutine object is used as lock to protect data structures that are used to communicate between the interrupt handler and the application threads or second level handlers. Interrupt priorities are higher than thread priorities. Executing a synchronized section with such an interrupt priority disables the scheduling interrupt and the interrupts at lower priority.

The notion of the interrupt handler in SCJ and the mapping of lock priorities to interrupt disabling are similar to the proposal in [17]. In our current implementation the JVM accepts a standard Runnable as an interrupt handler, which is registered by a system class. Even the thread scheduler is just a plain Runnable that is registered for the timer interrupt. Therefore, the SCJ implementation of interrupt handlers can keep the notion of the available Runnable and just invoke the handle() method of the interrupt handler. The hardware of JOP disables all interrupts, when an interrupt happens. By keeping them disabled during the handle() method and disabling them on all synchronized methods of the handler class we have a less responsive system, but are on the safe side for synchronization.

D. Code Size

The JOP build tool links only classes used by the application into the final application. Furthermore, small versions of the JDK (a CLDC 1.1 version and a subset of it) are available for JOP. Therefore, the overhead of the JVM is relative small. An SCJ Hello World example, all needed SCJ and JDK support classes, and the JMV Java classes result in an application binary of 102 KB.

V. Microbenchmarks

In this section we present the micro benchmarks developed to test our SCJ implementation. Each test is organized as an independent mission, where we use the mission memory to store test-specific configuration data.

A. Accuracy of Periods

The majority of the computational load in real-time systems comes from periodic activities (e.g. sampling sensor data and applying control laws) [3]. Support for periodic activities is provided in SCJ via the PeriodicEventHandler class where an application developer adds its own functionality by overriding the handleAsyncEvent() method and provides a *start* time and a *period*. The *start* value represents an offset measured from the start of the mission until the first release of the PEH (Φ in Figure 1).

Ideally, the j-th release of a PEH should happen at integer multiples of the PEH's period plus the initial offset, as shown in Figure 1. In practice however, due to e.g. executing higher priority threads at the release time of lower priority tasks, it may not be possible to completely eliminate the release delay. With a single task, one of the main contributions to the release jitter comes from the scheduling overhead.

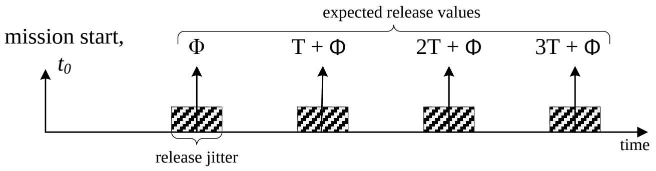


Fig. 1. Precision of periods

TABLE I
MEASURED PEH START TIME DEVIATIONS

Min (us)	Max (us)	Avg (us)	Stdev (us)
74	75	74.63	0.48

In this part of the evaluation we measured the accuracy of successive periodic releases. Our test setup uses a single PEH running at maximum priority, with a period $T = 150$ ms, and that is rescheduled by the SCJ framework to be release at the next ideal release time (see Figure 1) by calling `waitForNextPeriod()`. The measurement was done as follows:

1) We first obtain the mission start time, t_0
2) We get the actual release time of the *j*-th instance of a PEH, t_r^j, equal to the time at which its `handleAsyncEvent()` method is executed
3) We calculate the time interval between the actual j-th PEH release time and the start of the mission, $t_r^j - t_0$
4) We calculate the deviation of the start time for the j-th PEH release, Δ_j, as $\Delta_j = (t_r^j - t_0) - jT$

Table I shows the values of Δ_j for 1,000 releases of the single PEH. The release jitter is thus equal to the maximum deviation of the start time among all instances [3] and is equal to 1 us. We can see that there is almost no variation in the deviation of the start times, which is a desired characteristic for time predictable systems.

In our implementation, the values in Table I represent the interrupt dispatch latency plus one context switch delay. This is indeed the case, as it will be shown latter in sections V-D and V-E.

B. Linear-time Memory Allocation Time

SCJ requires that mission and private scoped memories be linear-time memory areas, i.e., memory regions where the allocation time is proportional to the size of the allocated data. Immortal memory however is not required to be of linear-time type [10]. If allocations in immortal memory are restricted to the initialization phase then whether or not the immortal memory is of linear-time type is not relevant. However, in mission phase, allocating data in immortal memory can affect the timeliness of the system as allocations may have variable execution times. In our implementation, all the SCJ memory areas are derived from a single system class, called `Memory`, that provides linear-time allocations

To test the linearity in memory allocation times ideally we would like to directly write a portion of the scoped memory we are interested in testing but in general there is no way to do that in Java. Another option is to allocate several objects for which we know their size, possibly obtained with the `SizeEstimator` class. However, `SizeEstimator` only provides a conservative estimate on the (implementation dependent) size of an object and we will also need to include the time to execute the object's constructor. We use a more VM independent approach which is to measure the time it takes to allocate a variable number of bytes of an integer array.

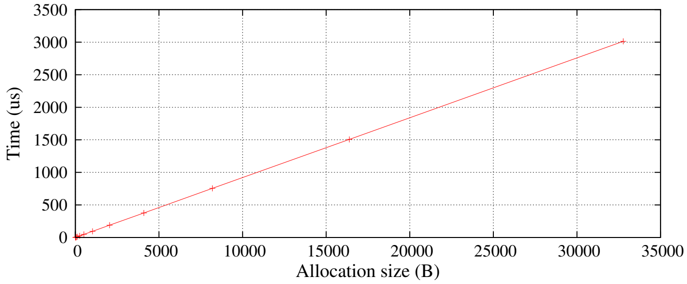


Fig. 2. Linear time memory allocation time

TABLE II
MEASURED SCOPED MEMORY ALLOCATION TIMES

Size (Bytes)	Min (us)	Max (us)	Average (us)	Stdev (us)
8	0	1	0.933	0.250
16	1	2	1.626	0.484
32	3	3	3.0	0.000
64	6	6	6.0	0.000
128	11	12	11.871	0.335
256	23	24	23.602	0.489
512	47	48	47.104	0.305
1,024	94	95	94.142	0.349
2,048	188	189	188.2	0.400
4,096	376	377	376.601	0.489
8,192	752	754	753.067	0.398
16,384	1505	1507	1506.025	0.584
32,768	3012	3013	3012.267	0.443

The results of 1,000 allocations of up to 32 KB is shown in Table II. Figure 2 shows a plot of the maximum allocation times registered where it can be seen the linearity in the allocation times. We also see in Table II that, even with a single thread, we registered maximum and a minimum values. We thought the variation was due to the idle tick of the scheduler but running the test with interrupts globally disabled still produces the variations. This issue remains to be investigated, but most likely is because allocating one word of memory takes less than 1 us, which is the accuracy of the timer used to take the measurements (this will also explain the value of zero in the first row of the table). Nonetheless, we see little variation in the data, a desirable characteristic in real-time systems where consistency is the main concern.

C. Aperiodic Event Handling

Real-time systems also need to respond to events (internal or external) that occur at random points in time. Handling this type of events has to be done as they occur, without disturbing the system's main application logic, which is usually executed by periodic activities [23]. Unanticipated events can be handled by *aperiodic* or *sporadic* tasks. An *aperiodic* task has either soft or no deadlines while *sporadic* tasks have hard deadlines and a minimum inter-arrival time [9].

SCJ does not provide support for sporadic tasks, as there is no detection of minimum inter-arrival time violations. Only aperiodic tasks can be implemented through the `AperiodicEventHandler` (AEH) or `AperiodicLongEventHandler` (ALEH) classes. Both AEHs and ALEHs must have a priority and, in the absence of shared resources, they will not interfere with the execution of higher priority PEHs. However, PEHs with lower priorities can miss their deadlines due to a higher priority AEH or ALEH with a large execution time. As opposed to RTSJ, where there is a configurable-size queue of events to service bursts of events, in SCJ the queue is of fixed size and equals to one with a queue overflow policy set to REPLACE, i.e. to overwrite pending event releases.

For this part of the evaluation, we test that: (1) high priority PEHs do not miss their deadlines, (2) the event queue size is equal to one, and (3) the queue overflow policy is set to replace. Our setup is as follows: we generate two task sets of PEHs with a processor utilization of 69% and 88%. The 69% is chosen as it is the theoretical utilization limit to guarantee schedulability under rate monotonic [8] and the 88% value represents the average case bound for rate monotonic [7]. We then increase the total load of the system by generating a burst of events that are to be served by a single ALEH. We use ALEHs because we can piggy-back a payload of type `long` to the servicing of an event request that is used to identify which event is being serviced.

The events are generated using a PEH that fires the ALEH at times that follow a Poisson distribution with a mean arrival time of 200 ms ($\lambda = 5$). The service time for each event, i.e. the execution time of the ALEH, varies from 15 ms to 200 ms in order to increase the aperiodic load of the system. The aperiodic load is a function of the number of serviced events, N, and the ALEH execution time, C, and is calculated as $NC/\Delta T$, where ΔT is the total running time of the experiment, in this case 50,000 ms. The priority of the ALEH is chosen to be a value between the priorities of the PEHs.

Figures 3a and 3b show the results of our experiments. In the left side of Figure 3a we see the increase in the total aperiodic load and on the right side the total number of events serviced during the execution of the test. The total number of events released was of 247. Figure 3b shows the number of deadlines missed per PEH where the high priority PEHs are PEH0, PEH1, PEH2, and PEH3. We see that as the aperiodic load increases, fewer events can be serviced and that deadlines start to be missed as the aperiodic load reaches around 20% and 10% for the 69% and 88% periodic loads respectively. We also see that the high priority PEHs never miss a deadline and that deadlines are first missed by the lowest priority PEH, i.e. PEH9 in Figure 3b.

It might appear from Figure 3b that as the aperiodic load increases, low priority PEHs will start to miss fewer deadlines. This is however not the case and what really happens is that low priority PEHs will have very long response times thus completing fewer releases. This situation will continue up to the point were not even a single release will be completed e.g. see the bar located at 130 ms in Figure 3b. This is necessary to handle the overload situation caused by the aperiodic handler.

We checked that events that arrive too close overwrite each other and that only the last received event will be served. This was checked by using the `long` parameter of the released ALEH to identify which event is being serviced. In this way we see that a REPLACE policy for the overflowing of an event queue of size 1 is effectively implemented (in JOP this queue is a one-element array).

We ran a similar experiment with several ALEHs running at the lowest priority and the results showed that no PEH missed its deadline. The total aperiodic load stays always below 12% and 31%. In addition, as the number of available low priority ALEHs increases, more events can be serviced, giving an intuition on how to dimension the system in terms of the number of aperiodic event handlers required to serve aperiodic events with a known arrival pattern. Due to the space restrictions of this paper, the results are not shown.

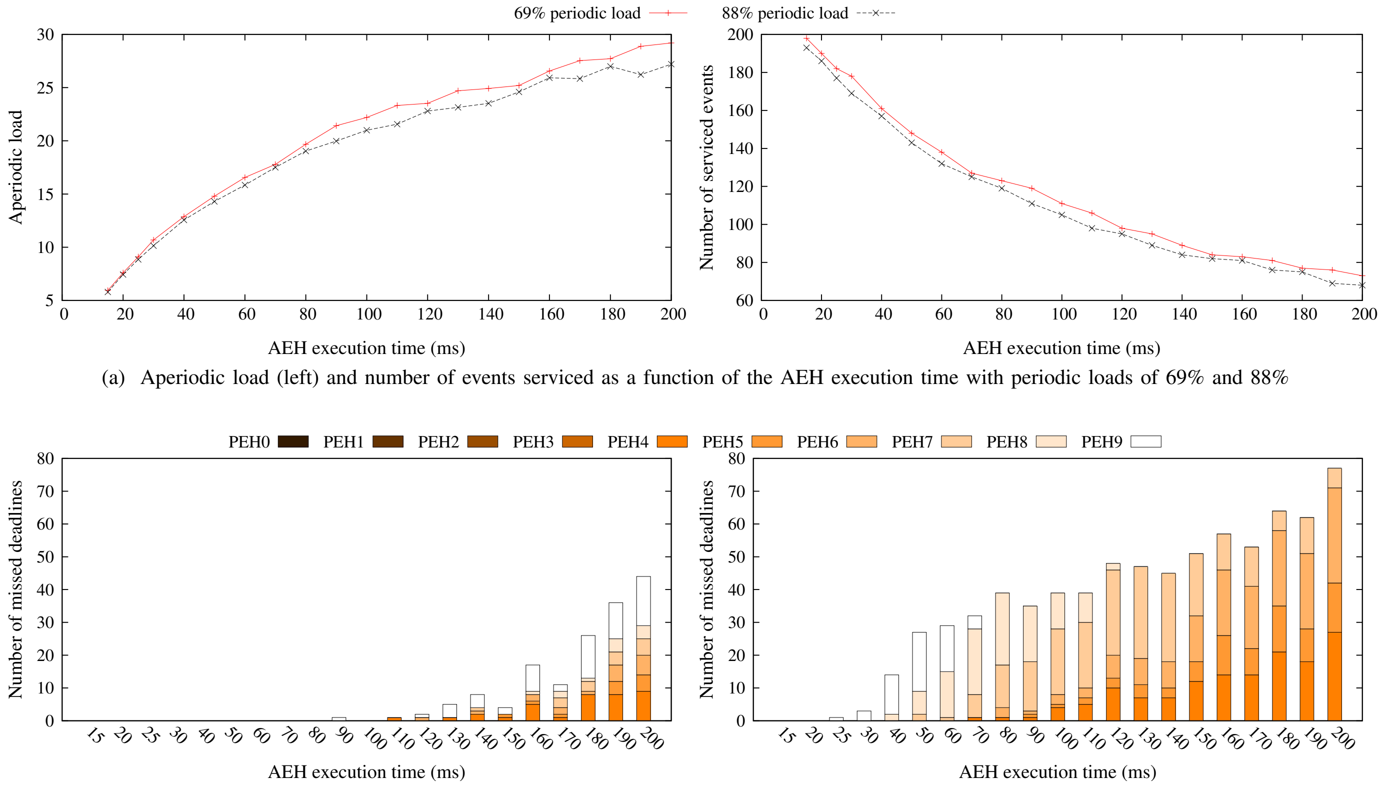
D. Dispatch Latency for Interrupts

SCJ supports the notion of first level interrupt handlers in Java thorough the `ManagedInterruptServiceRoutine` (MISR) class. Similar to other MSOs, MISR objects have to be registered to a specific mission and the interrupt service routine (ISR) code is implemented by overriding the `handle()` method of RTSJ's `InterruptServiceRoutine` class, which in SCJ is more restricted than its RTSJ counterpart. In general, in Java an ISR can be implemented as a handler or as an event [17]. The handler approach uses a method invoked by the hardware while the event approach uses a form of asynchronous event to fire an AEH or unblock a managed thread. The advantages and disadvantages of both methods are described in more detail in [17].

Interrupt handlers are executed at hardware priorities, i.e., priorities that are higher than those of ordinary schedulable objects, and can delay the completion of schedulable objects. It is therefore important that interrupts are dispatched as fast as possible. An ISR can be preempted (or interrupted) only by a higher priority interrupt, assuming that the ISR disables only lower or equal priority interrupts. Nested interrupts can make the system more responsive to external events but can increase the blocking times due to interrupts. In this part of our evaluation we test the interrupt dispatch latency.

To measure the interrupt dispatch latency we compute the time between the generation of an interrupt and its delivery.[1] How interrupts are generated is implementation dependent, making it difficult to measure the time of the interrupt generation in an way that can be ported between SCJ implementations. Measuring the time when the interrupt

[1] Generation is the hardware mechanism that makes the interrupt available to the Java program while delivery is the action that invokes an ISR [10]



(a) Aperiodic load (left) and number of events serviced as a function of the AEH execution time with periodic loads of 69% and 88%

(b) Number of missed deadlines as a function of the AEH execution time with periodic loads of 69% (left) and 88% (right)

Fig. 3. Total aperiodic load (top left), number of aperiodic events serviced (top right), and number of deadlines missed (bottom left and right) as a function of the event handler execution time for periodic loads of 69% and 88%.

TABLE III
INTERRUPT DISPATCH LATENCY

	Min (us)	Max (us)	Avg (us)	Stdev (us)
SCJ	25	26	25.23	0.422
Non-SCJ	4	5	4.6	0.491

TABLE IV
CONTEXT SWITCH LATENCY

Min (us)	Max (us)	Avg (us)	Stdev (us)
62	72	69.644	1.4361

is delivered is however non-implementation dependent and is equal to the time when the `handle()` method of the `InterruptServiceRoutine` is executed.

In our implementation we used JOP's system device to generate interrupts triggered by software using a PEH. The PEH is randomly selected at mission initialization from a group of PEHs with different priorities. The purpose is to show that indeed interrupt priorities are higher than MSOs priorities and that the dispatch latency is independent of the PEH's priority that generated the interrupt.

Table III compares the dispatch latencies for 1,000 interrupt occurrences when using SCJ's MISR and when using a plain runnable invoked by the hardware as the ISR. The overhead when using the SCJ implementation comes from invoking four methods as opposed to only one for the non-SCJ version. The additional methods are the `enter()` and its associated runnable's `run()` methods required to execute in the ISR's private memory, and the `InterruptServiceRoutine`'s `handle()` method.

E. Context Switch Latency

Context switch latency is a source of overhead that can affect the performance of the system. For this test, we create two PEHs, one with high priority, PEH$_H$, and the other with a lower priority, PEH$_L$. The start times of both PEHs are selected so as to make PEH$_L$ to begin execution before PEH$_H$. PEH$_L$ has a long period and constantly updates a variable with the current time, t_L. PEH$_H$ has a shorter period and reads the current time, t_H, as the first statement of its `handleAsyncEvent()` method. Because PEH$_H$ will always preempt PEH$_L$, the context switch latency will be equal to the difference between t_H and t_L. The results of our measurements, for 1,000 context switches, is shown in Table IV.

The minimum value is obtained when t_L is updated just before the context switch while the maximum value is obtained when the context switch happens just before updating t_L.

F. Synchronization

Contention for shared resources among managed schedulable objects can result in priority inversions. To avoid priority

TABLE V
TASK SET FOR SYNCHRONIZATION TEST

Priority	Offset (ms)	Critical section (ms)	WCET (ms)
HIGH	400	100	300
MEDIUM	300	0	400
LOW	0	500	600

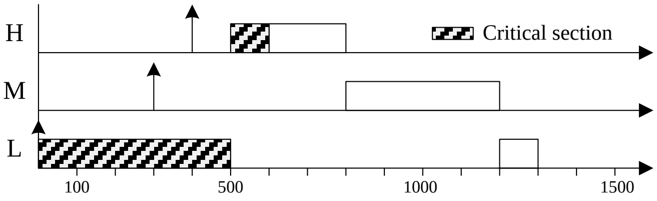


Fig. 4. Execution pattern of the task set of Table V

inversions, SCJ requires the use of the priority ceiling emulation (PCE) protocol to control access to shared resources.

To test a correct PCE implementation, we use the task set of Table V with three PEHs with low, medium, and high priorities. The offset column is the time of the PEH's first release, the critical section column is the time that the PEH requires a shared resource, and the WCET column is the total computation time. A shared object in mission memory is used by the low and high priority PEHs and its lock is acquired by the low priority PEH at time $t = 0$. At $t = 400$ the high priority PEH will try to lock it. With those values, we will force a potential priority inversion in the task set.

We expect that with a correct PCE implementation the execution pattern to be as shown in Figure 4. Measured release delay and response times for all PEHs is shown in TableVI.

JOP's implementation of the PCE protocol is done by globally disabling all interrupts within synchronized methods. As the scheduler is a first level interrupt handler fired by the programmable timer counter, with interrupts globally disabled there will be no scheduling decisions, i.e., once a PEH has acquired a lock it cannot be preempted or interrupted. The object's monitor ceiling is therefore effectively equal to the maximum priority of all the possible managed schedulable objects that could ever acquire the lock of the shared object.

G. Discussion

By organizing each micro benchmark as a single mission we are also testing compliance to the mission-based programming model of SCJ. We are implicitly testing the mission life cycle by: (1) changing missions and executing them in a predetermined sequence, (2) resizing the mission memory before mission initialization, (3) providing mission termination mechanisms, (4) using nested private memories at mission cleanup, and (5) delaying the start of tasks with the `start` parameter of the `ReleaseParameters` class.

TABLE VI
MEASURED TIMING VALUES OF SYNCHRONIZATION TASK SET

PEH	Release delay (ms)	Resp. Time (ms)
HIGH	501.840	403.565
MEDIUM	802.397	907.115
LOW	0	1308.939

In order to make our micro benchmarks portable across SCJ implementations we developed them using the available public SCJ API features. However, when we used SCJ's clock API for timing measurements an overhead is introduced due to additional method calls and normalization operations. Moreover, saving such measurements in *clock objects* allocated in shared memory areas increases the complexity. The use of the clock API is kept for portability but to increase the accuracy of our measurements we used JOP's microsecond counter. In this way our benchmarks are not inherently related to our particular SCJ implementation.

VI. SCJ's TCK and miniCDj

As SCJ is still in an early draft phase, currently there is no official TCK available that can be used to check our implementation against the specification. There is however a reduced SCJ TCK done at Purdue University that we can use to check the portions of the SCJ that the TCK exercises. As the TCK is organized as a collection of applications running on top of an SCJ implementation, it is not possible to exercise all the requirements of the specification. For example, to test that private scoped memories are not accessed by more than one thread, the TCK *explicitly creates* such memories from user space and enters them with the `enter()` method. Explicit creation of private scopes is not allowed in SCJ and enforced with a private constructor in the `PrivateMemory` class. Restricting the use of the `enter()` method is more difficult, as an implementation may be based on the RTSJ where such method is declared as public.

As a result, there are some tests that we cannot directly apply to our implementation. Moreover, as we are only targeting a L1 implementation, we skipped the L2 tests of the TCK. The tests that are exercised in our implementation by running Purdue's TCK are summarized in Table VII. The reader is referred [24] for the details of the tests.

To test our implementation with a complete application, we have used the miniCDj benchmark. We used six planes, 1,000 frames, and the period of the detector was set to 1,000 ms. No frame overruns were detected and values for the detector's execution time are shown in Table VIII.

The variation in the computation time is because collision detection depends on both, the current and the previous frames, thus generating a variable size space to search for potential collisions. The measured release delay is less than 3 microseconds thus it is not shown in Table VIII.

To see if parts of our SCJ implementation may affect the execution times, we also run this test with scope checks enabled and disabled, and using both, JOP's microsecond counter and the SCJ clock API to drive the cyclic schedule. None of these showed to add considerable overhead. The main source of execution time comes from the large number of floating point operations that in JOP are done in software.

TABLE VII
TCK TESTS USED IN OUR IMPLEMENTATION

1	Each application uses a global mission scope to hold global objects used during a mission
2	During the initialization, all objects are allocated in mission memory by default
3	Object creation in mission memory or immortal memory during the mission phase is allowed
4	Each schedulable object has its private scoped memory
5	`PrivateMemory` is based on LTMemory
6	`MissionMemory` is ScopedMemory
7	Nested private memory is supported
8	Mission memory is resizable
9	`PriorityScheduler.instance().getMaxPriority()` is the default ceiling for locks
10	Nested calls from one synchronized method to another are allowed
11	The only scheduler is the default RTSJ preemptive priority-based scheduler with at least 28 priorities. There is no support for changing base priorities
12	Priority Ceiling Emulation is supported
13	Full preemptively scheduling shall be supported at Level 1
14	A preempted schedulable object must be placed at the front of the run queue for its active priority level

TABLE VIII
MINICDJ BENCHMARK EXECUTION TIME MEASUREMENTS

Min (ms)	Max (ms)	Avg (ms)	Stdev (ms)
422.94	744.05	568.03	75.72

VII. Conclusion

In this work we have presented an evaluation of a SCJ implementation on a Java processor. We have developed a series of micro benchmarks organized as independent missions in order to test timeliness and performance of the system. We have tested the accuracy of periods, linear-time memory allocations, aperiodic event handling, interrupt dispatch latency, context switch latency, and synchronization. In addition, we have used an application-oriented benchmark, the miniCDj benchmark, to test in a non-isolated way our implementation. We have also used a reduced version of Purdue's SCJ technology compatibility kit to show how well our implementation adheres to the SCJ specification.

Acknowledgments

This work is part of the project "Certifiable Java for Embedded Systems" (CJ4ES) and has received partial funding from the Danish Research Council for Technology and Production Sciences under contract 10-083159. We also thank the reviewers for their comments on improving the final version of this paper.

Source Access

Our evaluation files can be found at: https://github.com/jrri/jop in the java/target/src/paper/jopscjeval/ directory.

References

[1] hiJaC case studies. http://www.cs.york.ac.uk/circus/hijac/case.html.

[2] 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 Trans. Embed. Comput. Syst.*, 7(1):1–49, 2007.

[3] G. C. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Real-time systems series. Springer, New York, 3rd ed edition, 2011.

[4] A. Corsaro and D. C. Schmidt. Evaluating Real-Time Java features and performance for real-time embedded systems. In *Real-Time and Embedded Technology and Applications Symposium, 2002. Proceedings. Eighth IEEE*, page 90–100. IEEE, 2002.

[5] B. P. Doherty. A real-time benchmark for Java™. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, page 35–46. ACM, 2007.

[6] 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*, page 41–50. ACM, 2009.

[7] J. Lehoczky, Lui Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. *Real Time Systems Symposium, 1989., Proceedings.*, pages 166–171, Dec. 1989.

[8] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[9] J. W. S. Liu. *Real-Time systems*. Prentice Hall, Upper Saddle River, NJ, 2000.

[10] 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 Technology Specification, Public draft.* 2013. v 0.94.

[11] J. Mc Enery, D. Hickey, and M. Boubekeur. Empirical evaluation of two main-stream RTSJ implementations. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, page 47–54. ACM, 2007.

[12] 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*, page 95–101. ACM, 2010.

[13] J. R. Rios and M. Schoeberl. Hardware support for safety-critical Java scope checks. In *Proceedings of the 15th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2012)*, pages 31–38, Shenzhen, China, April 2012. IEEE.

[14] M. Schoeberl. Real-time scheduling on a Java processor. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2004)*, Gothenburg, Sweden, August 2004.

[15] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54(1):265–286, 2008.

[16] 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 2011)*, pages 47–53, York, UK, September 2011. ACM.

[17] M. Schoeberl, S. Korsholm, T. Kalibera, and A. P. Ravn. A Hardware Abstraction Layer in Java. *ACM Transactions on Embedded Computing Systems*, 10(4):1–40, Nov. 2011.

[18] M. Schoeberl and J. R. Rios. Safety-critical Java on a Java processor. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 54–61, Copenhagen, Denmark, 2012. ACM.

[19] H. Søndergaard, S. E. Korsholm, and A. P. Ravn. Safety-critical Java for low-end embedded platforms. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, page 44–53, 2012.

[20] SPEC. SPEC JBB2005.

[21] T. B. Strøm and M. Schoeberl. A desktop 3d printer in safety-critical Java. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, page 72–79, 2012.

[22] TimeSys. RTSJ Reference Implementation.

[23] A. Wellings and M. Kim. Asynchronous event handling and safety critical Java. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 53–62, Prague, Czech Republic, 2010. ACM.

[24] 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.