Developing Safety Critical Java applications with oSCJ/L0

Ales Plsek, Lei Zhao, Veysel H. Sahin, Daniel Tang, Tomas Kalibera†, Jan Vitek
Purdue University †Charles University

ABSTRACT

We present oSCJ, an implementation of the draft of Safety Critical Java (SCJ) specification. SCJ is designed to make Java amenable to writing mission- and safety-critical software. It does this by defining a subset of the Real-time Specification for Java that trades expressiveness for verifiability. This paper gives a high-level description of our implementation of the first compliance level of the SCJ specification, a library called oSCJ, and reports on performance evaluation on the Ovm real-time Java virtual machine. We compare SCJ to C on both a real-time operating system on the LEON3 platform and Linux on a x86. Our results suggest that a high-degree of predictability and competitive performance can indeed be achieved.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*run-time environments*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*classes and objects*; D.4.7 [**Operating Systems**]: Organization and Design—*real-time systems and embedded systems*.

General Terms

Languages, Experimentation.

Keywords

Safety Critical Systems, Java virtual machine, Memory management.

1. INTRODUCTION

Software reliability is a central concern for mission- and safety-critical systems such as avionics control systems, medical instrumentation or space applications. A failure or malfunction may result in damage to equipment, serious injury or even death. Thus, safety-critical applications are required to follow an exceedingly rigorous development, validation, and certification process. Safety critical software is written to be verifiable, often in a restricted subset of a general purpose programming language such as C or Ada. However, with growing size of code bases – million line systems are not unusual – increasing software productivity is becoming a driving force behind the move to adopt technologies such as Java. The Java programming language and platform bring a number of features that facilitate the cost-effective development of large software systems.

To support safety-critical development in Java, the JSR-302 expert group was tasked with defining a new standard – the Safety Critical Java (SCJ) specification [5]. The SCJ specification is based on a subset of Real-Time Specification for Java (RTSJ) [2] and is designed to enable the creation of applications, infrastructures, and libraries that are amenable to certification under safety critical standards (such as DO-178B, Level A). The SCJ presents embedded developers with a simplified programming model centered around independent tasks called missions which have access to a restricted version of scoped memory. Other features of the Java platform, such as threading, reflection, and class loading, are restricted to reduce the code base and simplify certification. As mission- and safety-critical applications can have very different requirements, the SCJ specification offers three compliance levels with increasingly richer programming models allowing developers to tailor the platform to the needs of their application.

This paper introduces oSCJ, an open-source implementation of the first compliance level of the draft SCJ specification. We implemented SCJ as an independent Java library with a well defined interface to the underlying virtual machine. To run it efficiently we modified the Ovm [1] real-time JVM by removing features that were not required and adapting data structures and algorithms to the simpler SCJ model. A set of tools comes with oSCJ. In particular, a static checker for proving certain properties of SCJ applications [10] and a Technology Compatibility Kit [11]. For evaluation, we refactored the CDx benchmark [6] and ran it on the LEON3 architecture, a platform that is used by NASA and ESA, and the RTEMS real-time operating system. The performance evaluations shows that SCJ has performance competitive with C. The library, VM, and the benchmark are freely available under an open-source license[1].

The SCJ specification is still under development, the API and examples presented in this paper are subject to change.

JTRES'10 August 19-21, 2010 Prague, Czech Republic

[1]The oSCJ is available from `www.ovmj.net/oscj`.

2. THE SAFETY CRITICAL JAVA SPECIFICATION

The complexity of safety critical software varies between application; the SCJ specification lets developers tailor the capabilities of the platform to the needs of application through compliance levels. The first level, Level 0, provides a simple, frame-based cyclic executive model which is single threaded with a single mission. Level 1 extends this model with multithreading via aperiodic event handlers, multiple missions, and a fixed-priority preemptive scheduler. Level 2 lifts restrictions on threads and supports nested missions. In the remainder we will focus on Level 0.

2.1 Missions

SCJ programs are organized as *missions*, which are independent computational units with respect to lifetime and resources needed. Each mission is composed of a bounded number of *schedulable objects*. Missions are launched according to a pre-defined order. Figure 1 shows the three phases of a mission: *initialization*, *execution*, and *cleanup*. After a mission terminates, the next mission is released if there is one.

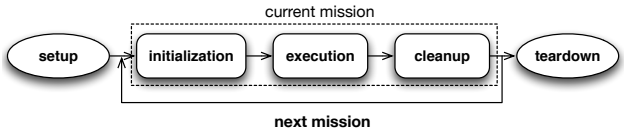

Figure 1: Mission Life Cycle.

The concept of schedulable object is inherited from RTSJ. A schedulable object contains both computation logic and some scheduling constraints, such as release time, deadline, priority, and so on. In the SCJ specification, schedulable objects have a dedicated thread and are restricted to periodic event handlers (PEH), aperiodic event handlers (AEH), and managed threads (MT) to simplify feasibility analysis.

At Level 0, the *cyclic executive model* defines a mission as a set of computations, each of which is executed periodically in a precise, clock-driven timeline, and processed repetitively throughout the mission. The only schedulable objects permitted at Level 0 are Periodic Event Handlers. All PEHs execute under control of a single underlying thread, so the implementation can safely ignore synchronization in the application. In this scenario, an operation which blocks will block the entire application. A Level 0 application runs on a single CPU.

Different parts of the SCJ APIs are available during the various phases of a mission. For instance, during the initialization phase, a private memory area is made available to the application for allocating temporary objects which will be reclaimed before any of the schedulable objects of the mission get to run. Other features of the platform are similarly restricted.

Figure 2 illustrates some of the core classes of the SCJ specification. At the beginning of program execution, the primordial thread – the first thread created by the system starts in immortal memory and creates a mission sequencer after executing the setup procedure. The mission sequencer holds references to all of the missions and repeatedly selects the next mission to launch. The mission sequencer and all mission instances live in immortal memory. Upon launching a mission, the mission memory is allocated with the desired size. A mission manager is then created, in mission memory, to control the mission's schedulable objects. The three phases of the mission are all executed in mission memory. All schedulable objects are created in the initialization phase of the mission; they are then started upon entering execution phase. A mission runs forever unless a termination request is sent explicitly by one of its schedulable objects. If a termination request is sent, the mission enters the cleanup phase. The cleanup phase includes storing or reporting the outcomes of a mission, as well as releasing all acquired resources. Once the mission is completed, all mission-specific objects are deallocated as the mission memory is reclaimed.

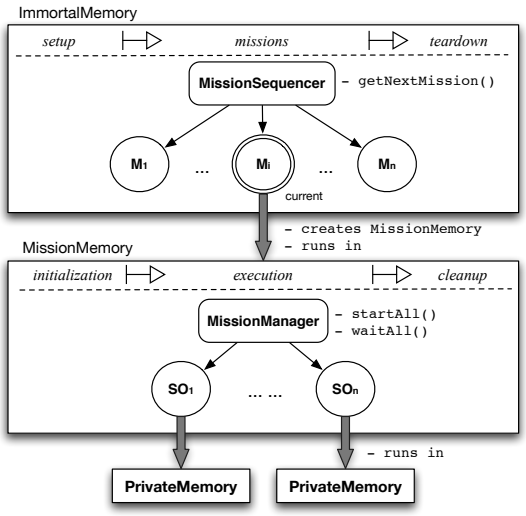

Figure 2: SCJ Mission classes.

2.2 Scoped Memory

The scope-based memory model introduced by RTSJ is retained in the SCJ specification. The main differences with the RTSJ are that the heap has been completely abandoned and that scoped memory has been further restricted to make certification easier. The SCJ specification has three types of memory areas: *immortal memory*, *mission memory*, and *private memory*. Immortal memory spans the lifetime of the virtual machine; therefore, only objects that should survive the entire program execution should be allocated there. The lifetime of the latter two memory areas is bounded. Each mission has a mission memory which is shared by the mission's schedulable objects and used to allocate data that must persist throughout the mission. Each schedulable has its own private memory area for the data that is needed for only a single activation of the schedulable. The `enterPrivateMemory(s, logic)` method creates a nested private

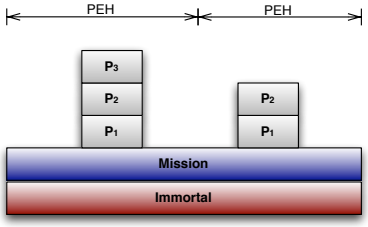

Figure 3: Memory structure of a SCJ program.

memory of size `s`, enters it, and execute the `logic` within it. A scope stack is a logical data structure that represent the scoped memory areas that a given schedulable object has entered implicitly (each schedulable object implicitly enters mission memory and its private memory), or by calling `enterPrivateMemory()` or `executeInArea()`. Figure 3 illustrates this with the stacks of schedulable objects — two periodic event handlers. The stacks grow logically from immortal memory up to private memories. Unlike the RTSJ where cactus stacks are allowed, the SCJ specification restricts navigation to linear sequences of scopes. This is achieved by removing the RTSJ's `enter()` method and replacing it with the more restricted `enterPrivateMemory()` which which only creates a subscope if there isn't one active already.

Another simplification in the SCJ is that private memories are always accessed by a single schedulable object, and thus do not require synchronization on entry or exit. At Level 1 and 2, mission memory and immortal memory can be accessed concurrently by multiple threads.

For memory safety, every write to a field holding an object reference must be checked to prevent dangling references. An optional set of annotations can be used to prove that at compile-time that all stores are safe. When these annotations are used, an implementation is permitted to omit scope checks.

3. IMPLEMENTING SCJ

Development of a SCJ infrastructure is a non-trivial task. Figure 4 illustrates such an infrastructure with SCJ applications running on top of a SCJ library which, itself, communicates with a dedicated virtual machine. Most of the oSCJ functionality has been implemented in the library with a clear interface to the underlying virtual machine to ease the task of porting it across VMs. For the virtual machine, we have modified Ovm [1] to provide only the features required to be compliant with SCJ. The runtime support for our VM is lightweight. It contains two components: the memory manager and an OS abstraction layer for thread management. The runtime currently runs on POSIX-like platforms such as Linux, NetBSD, and on top of the RTEMS classic API.

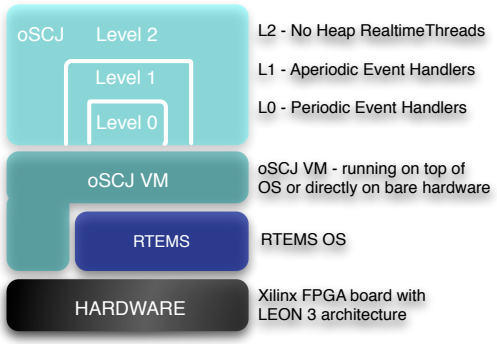

Figure 4: oSCJ Infrastructure.

Since we focus on Level 0, we reduced the number of VM threads to a single thread. This thread performs both the boot and initialization sequences of the VM and then starts executing the SCJ application. This required changing the type of Ovm's primordial thread to be instance of the `RealtimeThread` class. Furthermore, since there is only one thread, we have turned off synchronization.

3.1 Virtual Machine Interface

Our SCJ library is designed to be independent from the underlying VM. Therefore, we have designed an interface that serves as a unique connection point between the library and the VM. Figure 5 lists the methods in the VM interface. The methods

The main tasks delegated to the VM are responsible for creation and deletion of memory areas, switching of allocation context as well as other memory-related services. The `delayCurrentThreadAbsolute()` method is used by the `CyclicExecutive` class to wait for the start time of the next execution frame. The `getCurrentTime()` and `getClockResolution()` methods are dedicated to time related services.

```
interface VM_Interface {
  Opaque   makeExplicitArea(long size);
  void     destroyArea(Opaque area);
  Opaque   makeArea(MemoryArea ma, long size);
  Opaque   setCurrentArea(Opaque scope);
  Opaque   getCurrentArea();
  MemoryArea getAreaMirror(Opaque area);
  Opaque   getImmortalArea();
  Opaque   areaOf(Object ref);
  long     getScopeSize(Opaque scope);
  long     memoryConsumed(Opaque scope);
  long     memoryRemaining(Opaque scope);
  void     resetArea(Opaque area);
  long     sizeOf(Class clazz);
  long     sizeOfReferenceArray(int length);
  long     sizeOfPrimitiveArray(int length,
                                Class clazz);
  void     storeInOpaqueArray(Opaque[] arr,
                              int index, Opaque val);
  int      delayCurrentThreadAbsolute(long nanos);
  long     getCurrentTime();
  long     getClockResolution();
}
```

Figure 5: oSCJ VM Interface.

We have optimized the in-memory representation of objects such that each Java object has a two-word header. The first word is a pointer to type information for the object and the second is unused (we intend to store a scope descriptor in this field to speed up scope checks). Lock and GC fields are not needed, and, as object never move in SCJ, hash codes can be computed based on the objects' addresses.

3.2 Memory Model

In our implementation, physical memory is organized in three levels as illustrated in Figure 6. The top level consist of immortal memory and all active missions. Each mission has a mission memory and what we call a backing store level. This level is made up of a stack of private memory areas. Finally, the scope level, describes the organization of the memory within a scope where objects are allocated contiguously.

At the top level, memory management is simple. Immortal memory is pre-allocated and is never de-allocated, and missions are added under the control of the mission se-

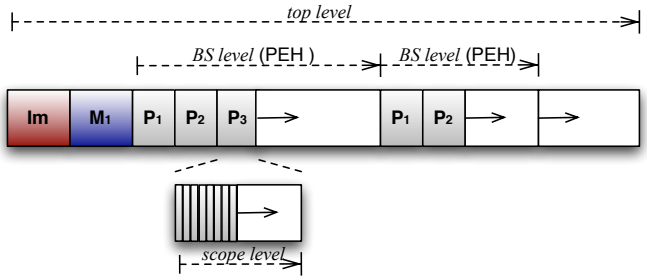

Figure 6: Physical memory is organized as three levels of stacks: the *top* level, *backing-store* level, and *scope* level.

quencer. At Level 0, there is at most a single mission active at any given time. The size of a mission is given by a storage configuration parameter that is attached to the mission. At the backing store level, schedulable objects are created at mission startup and space is reserved for each of them. Their private memories are managed using a stack discipline and are laid out contiguously in the space assigned their schedulable. Private memory areas are allocated when they are entered and reclaimed when their are exited. At scope level, objects are allocated in bump pointer fashion. Deallocation is implicit when the private memory is reclaimed.

This implementation has low-overhead. Allocation is constant time, as it boils down to bumping a pointer. Deallocation is linear in the size of the scope (due to the time spent zeroing memory on exit). The bookkeeping overhead for scope stacks is just two pointers (base and top). Moreover, there is no fragmentation due to scopes. SCJ requires the programmer to specify maximum memory requirements in advance, and our memory layout can fulfill any sequence of allocation requests that fits in the scopes. However, the raw size of memory allocated for each scope memory area is slightly bigger than the requested memory size as it is aligned by a memory page size, which depends on the architecture of the system. Lastly, none of the allocation operations requires synchronization in Level 0. (For other compliance levels, private memory are thread local and thus do not require synchronization).

The scope-stack that keeps track of entered memory areas is implemented as an array. A pointer into the array is referencing the currently active scope. As the list of entered memory areas grows, the array size is being dynamically adjusted.

3.3 Scope Checks

The VM implements scope checks to verify each memory access. Consider the assignment `x.f=y`. It is safe if the scope in which `y` is allocated is longer lived than the scope of `x`. Since our VM memory areas are continuously starting with immortal memory, continuing with mission memory, and the backing stores of schedulable objects, nested private memory areas are allocated with longer lived areas first. Memory areas allocated on lower addresses are longer lived than areas allocated on higher addresses. The code of the scope check is given in Figure 7. On the fast path, we check if the `x` and `y` are allocated in the same block of memory or if the block of memory where `x` resides is in a higher address space. In case this holds, the check succeeds since the region where `x` is allocated will be reclaimed earlier or at the same time as `y`. In case this does not hold, we enter the `storeCheckSlow`. In the slow path, we first check if `y` is in the immortal memory, which would cause the check to succeed. Then we check if `x` is in the immortal memory, since `y` is not in immortal, this would cause the check to fail. We special case the checking for the immortal memory, since its boundaries are not included in the boundaries of the scope memory backing store due to engineering reasons. Finally, we are safe to retrieve scope information of `x` and `y` from the array that keeps track of the living scope memories. If the variables are in the same scope, the assignment is legal, otherwise the scope check fails since `x` resides in a scope that lives longer than the scope of `y`.

```
void storeChecks(Oop x, Oop y) {
  int rx = x.asInt() >>> blockShift;
  int ry = y.asInt() >>> blockShift;
  if (rx >= ry) OK;
  else storeChecksSlow(rx, ry);
}

void storeChecksSlow(int rx, int ry) {
  if (ry < immortalEndIndex) OK;
  else if (rx < immortalEndIndex) FAIL;
  else {
      Area x_scope = scopeOwner[rx - scopeBase];
      Area y_scope = scopeOwner[ry - scopeBase];
      if (x_scope == y_scope) OK;
      else FAIL;
  }
}
```

Figure 7: Scope check implementation.

4. EVALUATION

We now set out to demonstrate that oSCJ can be competitive with hand-written C code in performance and predictability. To this end, we set up a representative workload on a realistic platform. For our workload, we selected the CDx [6] benchmark that we refactored to use the SCJ API. The refactored version is called miniCDj. We used a set of SCJ annotations to prove that the benchmark is allocation safe [10], this allowed us to disable scope checks in the VM during the benchmarking.

We compare miniCDj to CDc, a C implementation of the same [7] that matches closely the Java version, with one source file per Java class. Our experiments were run on a GR-XC3S-1500 LEON development board. The board's Xilinx Spartan3-1500 field programmable gate array was flashed with a LEON3 configuration, without a floating-point unit, running at 40MHz, with an 8MB flash PROM and 64MB of PC133 SDRAM split into two 32MB banks. The version of RTEMS is 4.9.3. Our second platform is an Intel Pentium 4 3.80GHz single core machine with 3GB of RAM, running Ubuntu Linux 9.04 with the 2.6.28-15-generic 32-bit SMP kernel.

4.1 Benchmark overview

The CDx benchmark suite [6] is an open-source family of benchmarks that can be used to measure performance of var-

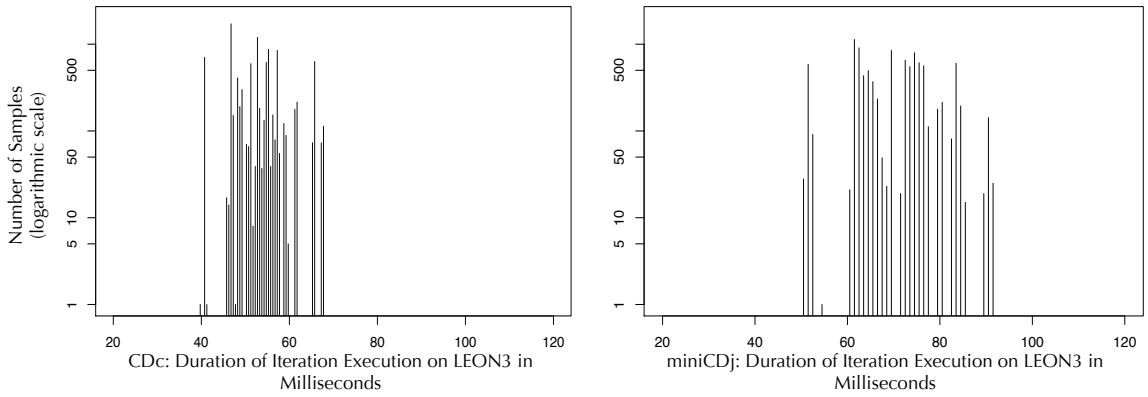

Figure 8: Histograms of execution times on LEON3. The observed worst-case for C is 34% faster Java.

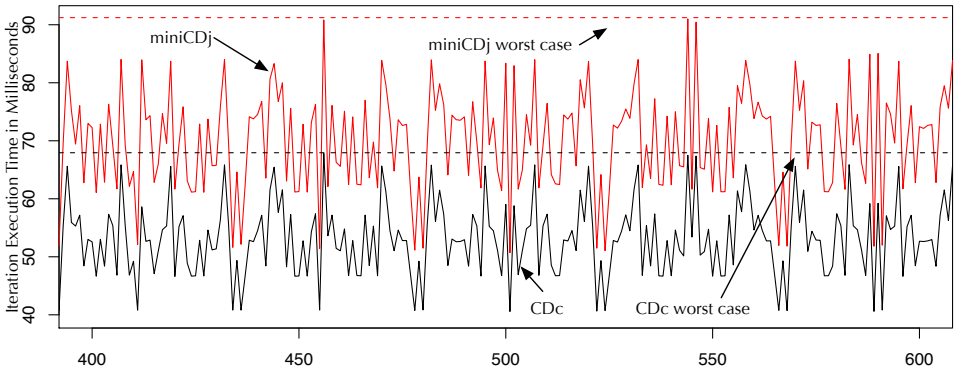

Figure 9: A detailed runtime comparison of for 200 iterations on LEON3.

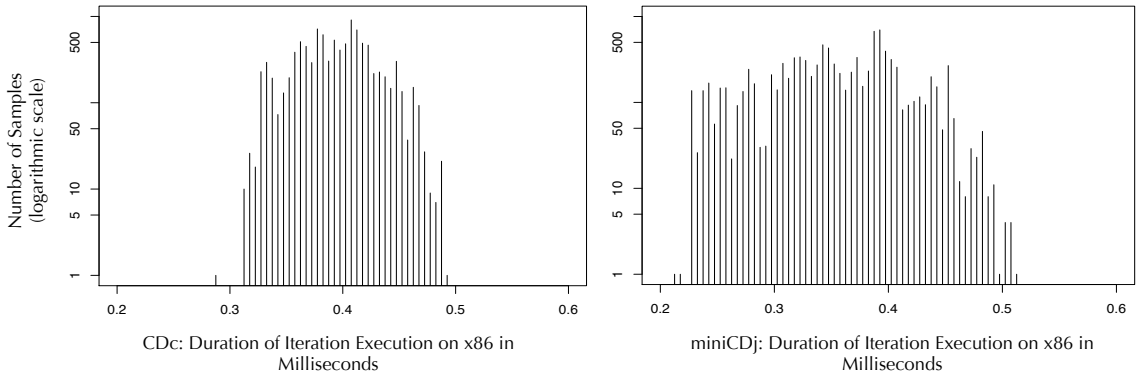

Figure 10: Histograms of execution times on X86. The observed worst case for C is 4% faster than Java.

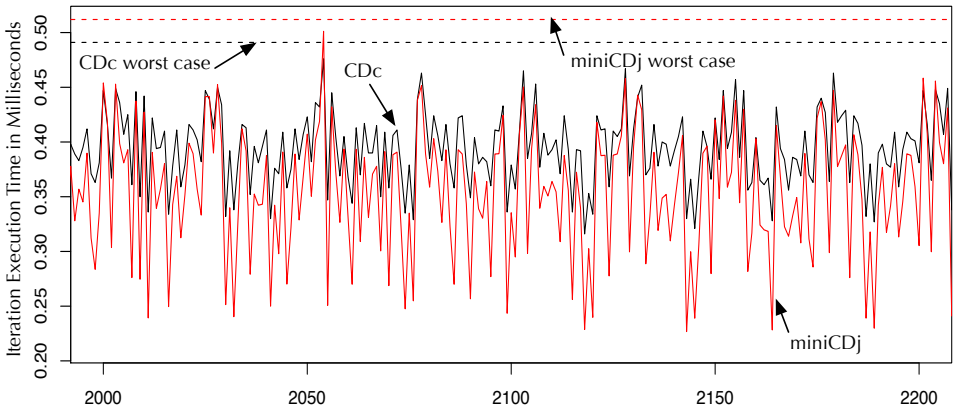

Figure 11: A detailed runtime comparison of CDc and miniCDj for 200 iterations on x86.

```
@SCJAllowed(value=LEVEL_0, members=true)
@Scope("immortal")
public class Level0Safelet extends CyclicExecutive {
  public Level0Safelet() { super(null); }

  public void setUp() {
    new ImmortalEntry().initialize();
    new Simulator().generate();
  }

  @RunsIn("cdx.Level0Safelet")
  protected void initialize() {
    new CollisionDetectorHandler();
  }

  public void tearDown() { dumpResults(); }

  public CyclicSchedule getSchedule(
                          PeriodicEventHandler[] handlers) {
    CyclicSchedule.Frame[] frames = new CyclicSchedule.Frame[1];
    frames[0] = new CyclicSchedule.Frame(
                  new RelativeTime(PERIOD, 0), handlers);
    return new CyclicSchedule(frames);
  }

  public long missionMemorySize() { return DETECTOR_SIZE; }
}
```

```
@SCJAllowed(value=LEVEL_0, members=true)
@Scope("cdx.Level0Safelet")
@RunsIn("cdx.CollisionDetectorHandler")
public class CollisionDetectorHandler
                    extends PeriodicEventHandler {

  private final TransientScopeEntry cd = new
    TransientScopeEntry(new StateTable(), VOXEL_SIZE);
  public boolean stop;

  public CollisionDetectorHandler() {
    super(null, null, null, TRANSIENT_SIZE);
  }

  public void handleEvent() {
    if (stop)
      Mission.getCurrentMission().requestSequenceTermination();
    else
      runDetectorInScope(cd);
  }

  public void runDetectorInScope(TransientScopeEntry cd) {
    RawFrame f = ImmortalEntry.frameBuffer.getFrame();
    cd.setFrame(f);
    cd.processFrame();
    ImmortalEntry.frames++;
    stop = (ImmortalEntry.frames == MAX_FRAMES);
  }
}
```

Figure 12: miniCDj code example showing `Level0Safelet` and `CollisionDetectorHandler` classes.

ious hard and soft real-time platforms[2]. A CDx benchmark consists of a periodic task that takes air traffic radar frames as input and predicts potential collisions. The benchmark itself measures the time between releases of the periodic task, as well as the time taken to compute potential collisions. For our evaluation, we used a pre-simulated formula to generate the radar frames to achieve a consistent workload across executions. The benchmark is floating-point intensive.

The benchmark is well-suited to the cyclic executive model, creating miniCDj was fairly straightforward. Data is recorded in immortal memory and is used for report generation after the mission terminates. The main computation is done in a periodic event handler, as the primary computation in CDx is periodic. However, since the algorithm relies on positions in the current and previous frame for each iteration, it is not enough to simply have a periodic event handler, whose memory is wiped after each iteration. A state table keeps track of the previous positions of each airplane. Since the state table is only relevant during the lifetime of the mission, placing it inside the mission memory is the ideal solution. miniCDj is approximately 3300 lines of code.

4.2 SCJ vs. C on LEON3

In our first experiment, we compared CDc with miniCDj on the LEON3 platform. The periodic task runs every 120 milliseconds with 6 airplanes and 10,000 iterations. Figure 8 gives the runtime performance of CDc and miniCDj. No deadlines were missed in any executions. On average, the execution time of one iteration in CDc is around 53 milliseconds, while for miniCDj it is around 69 milliseconds. The median execution times for CDc are only 28% smaller than the median for miniCDj. For real-time developers, the key metric of performance is the worst case. C is 34% faster than SCJ in the worst-case. Figure 11 shows a more detailed view of a subset of the iterations. There is a strong correlation of execution times between CDc and miniCDj. In this benchmark, Java is as predictable as C.

[2]See `www.ovmj.net/cdx` tags "miniCDj v1.1", "CDc v1.1".

4.3 SCJ vs. C on x86

In this second experiment, we compared miniCDj to CDc on x86. We configured the benchmark to run with a more intensive workload, using 60 planes with a 60 millisecond period for 10,000 iterations. The increased number of planes brings more detected collisions, which consequently poses higher demands on data structures, arithmetics, and memory allocation. The histogram in Figure 10 shows the frequency of execution times for SCJ and C. The data demonstrates that on average miniCDj is by 12% faster than CDc. Looking at the worst-case performance times, miniCDj has a 4% overhead over CDc. We can observe again that the results are highly correlated. In fact, for the most of the time, miniCDj performance times stay below those of CDc. It is unclear why Java would be faster than C. We believe that Ovm facilitates inlining by generating a single C file and removing most of the polymorphism around methods calls, but this should have the same effect on both platform. The x86 workload performs more allocation and CDc uses `malloc/free` which is more expensive than the bump-pointer allocation used for scoped memory in SCJ. Lastly, the SCJ performance on LEON3 may have been skewed by the fact that the Java code performs more floating point operations (the C code was optimized by hand) and the LEON3 does not have a hardware fp unit.

5. EXAMPLE

To illustrate the use of SCJ API we provide a code snippet of the miniCDj benchmark. Figure 12 shows two key classes of the benchmark : `Level0Safelet` and `CollisionDetectorHandler`. The `Level0Safelet` class extends the `CyclicExecutive` class and represents an instance of a Level 0 mission. It defines the specific actions that will be executed during the initialization (the `setUp()` method) and the

cleanup of the mission (the `cleanUp()` method). First, the oSCJ infrastructure asks for required mission space by calling `missionMemorySize()` method and creates a corresponding mission area. Furthermore, the `initialize()` method is executed to instantiate all the periodic event handlers that will be periodically executed during the mission. Therefore, the infrastructure instantiates the `CollisionDetectorHandler` class and creates a corresponding private memory - the size of the private memory is given in the constructor of the handler. Finally, the `getSchedule()` represents a Level 0 scheduler and defines the frequency of the handler execution.

The `CollisionDetectorHandler` class represents a `PeriodicEventHandler` dedicated to the mission and defines the code that will be executed each time the handler is scheduled to run. The handler is executed by the infrastructure code calling the `handleEvent()` method. Inside the `handleEvent()` method we delegate the functionality to the `runDetectorInScope()` method that first receives a data frame holding the current positions of the aircraft[3] and process these data by calling `cd.processFrame()`.

As the SCJ specification defines, a termination of the mission is explicitly under the control of the handler. In this case, once the number of processed frames reaches the `Constants.MAX_FRAMES` limit, the `stop` variable is set to `true`. This will cause the handler to call `Mission.getCurrentMission().requestSequenceTermination()` during the next execution of the `handleEvent()` and the mission will be terminated.

Finally, note that both the classes contain the SCJ annotations specifying their level and runtime allocation context, these annotations are used during the static analysis to prove allocation safety of the application.

6. RELATED WORK

The only other implementation of the SCJ specification is the reference implementation developed by the JSR-302 expert group. It is built on top of RTSJ and meant for prototyping purposes. SCJ itself has roots in previous safety-critical profiles for Java and Ada. Ravenscar Ada [3] inspired a restriction of the RTSJ proposed in [8]. That works was similar to SCJ Level 1. Another profile was proposed within the HIJA project [4]. Yet, another profile [9] argues against reusing RTSJ-based classes to avoid inheriting unsafe RTSJ features. Similar to SCJ, the authors propose a mission mode that permits recycling CPU time budgets for different phases of the application.

Fiji VM [7] is a new Java virtual machine dedicated to embedded, hard real-time devices that, similarly to the oSCJ VM, compiles Java code to C and supports a wide set of platforms, including x86 and the LEON3 architecture running the RTEMS operating system. The Aonix PERC Pico VM introduces stack-allocated scopes, an annotation system, and an integrated static analysis system to verify scope safety and analyze memory requirements.

7. CONCLUSION

This paper presented an implementation of the Safety Critical Java specification. The oSCJ library is isolated from the underlying virtual machine by a clearly defined interface and can thus be easily ported to different virtual machines. In order to perform some performance experiments we have modified the Ovm real-time Java virtual machine to implement the oSCJ interface. We have successfully deployed oSCJ on LEON3 and x86 and shown that it is capable of running a representative, safety critical workload on an embedded operating system and hardware platform, delivering predictability and throughput that is comparable to C code.

[3]Do not mistake with the execution *frames* from the `getSchedule()` method.

Acknowledgments. This work was partially supported by NSF grants CNS-0938256, CCF-0938255, CCF-0916310 and CCF-0916350. The authors thank Petr Maj, Filip Pizlo, and Ghaith Haddad and Gaisler Inc.

8. REFERENCES

[1] Austin Armbuster, Jason Baker, Antonio Cunei, David Holmes, Chapman Flack, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. A Real-time Java virtual machine with applications in avionics. *ACM Transactions in Embedded Computing Systems (TECS)*, 7(1), 2007.

[2] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, June 2000.

[3] Alan Burns. The Ravenscar Profile. *ACM SIGADA Ada Letters*, 19(4), 1999.

[4] HIJA. European High Integrity Java Project. `www.hija.info`, 2006.

[5] JSR 302. Safety critical Java technology, 2007.

[6] Tomas Kalibera, Jeff Hagelberg, Filip Pizlo, Ales Plsek, and Jan Vitek Ben Titzer and. CDx: A Family of Real-time Java Benchmarks. In *International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, 2009.

[7] Filip Pizlo, Lukasz Ziarek, Ethan Blanton, Petr Maj, and Jan Vitek. High-level programming of embedded hard real-time devices. In *EuroSys Conference*, 2010.

[8] Peter Puschner and Andy Wellings. A profile for high integrity real-time java programs. In *Proceedings of the 4th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, 2001.

[9] Martin Schoeberl, Hans Sondergaard, Bent Thomsen, and Anders P. Ravn. A Profile for Safety Critical Java. In *Int. Sym. on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, 2007.

[10] Daniel Tang, Ales Plsek, and Jan Vitek. Static Checking of Safety-Critical Java Annotations. In *International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, 2010.

[11] Lei Zhao, Daniel Tang, and Jan Vitek. A Technology Compatibility Kit for Safety Critical Java. In *International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, 2009.