Memory Management for Safety-Critical Java

Martin Schoeberl
Department of Informatics and Mathematical Modeling
Technical University of Denmark
masca@imm.dtu.dk

ABSTRACT

Safety-Critical Java (SCJ) is based on the Real-Time Specification for Java. To simplify the certification of Java programs, SCJ supports only a restricted scoped memory model. Individual threads share only immortal memory and the newly introduced mission memory. All other scoped memories are thread private. Furthermore, the notation of a maximum backing store requirement enables implementation of the scoped memories without fragmentation issues.

In this paper we explore the implications of this new scoped memory model and possible simplifications in the implementation. It is possible to unify the three memory area types and provide a single class to represent all three memory areas of SCJ. The knowledge of the maximum storage requirements allows using nested backing stores in the implementation of the memory area representation. The proposed design of an SCJ compliant scope implementation is evaluated on an embedded Java processor.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection)*

Keywords

Safety-critical Java, scoped memory

1. INTRODUCTION

Memory management for real-time and safety-critical systems is a critical issue. In a desktop application a memory leak might not be noticed as virtual memory gives the illusion of almost infinite main memory. We are also already used to programs that once in a while crash, which is often caused by a too early release of dynamic memory. However, in safety-critical systems both issues cannot be tolerated. Safety-critical applications often run for a very long time and a memory leak will lead to an error. Furthermore, real-time systems often do not use virtual memory due to the unpredictable timing a page swap introduces.

As a result, memory is used in a very conservative way in safety-critical systems. Most data structures are static and temporary data is allocated only on the runtime stack. Heap allocation with malloc is avoided, as the effects of heap fragmentation are hard to predict.

However, standard Java has no notion of static objects or arrays and no stack allocation. All data is allocated on the heap and memory is recycled with the help of a garbage collector (GC). To bridge the gap between the conservative memory management in real-time systems and the very dynamic memory management in Java, the Real-Time Specification for Java (RTSJ) [4] introduces new memory areas. Immortal memory is used for the statically allocated data and scoped memory to provide a similar facility as stack allocation.

Safety-Critical Java (SCJ) [6], as a subset of the RTSJ, provides similar, but restricted memory areas. SCJ defines immortal memory, mission memory, and private memory. Furthermore, the maximum size of a collection of memory areas (their *backing store* requirement) needs to be specified by the application. This size is needed for an implementation of scoped memories that avoids memory fragmentation.

Scoped memories can be nested. The object, which represents the memory area, is allocated in a scope memory and the nested, inner scope can be entered. Contrary to intuition, the memory for the nested scope (the backing store) is not allocated from the outer scope.

The SCJ inherits the same semantic for nested scopes. A size for a scope does not need to include any size of a nested scope. However, with the knowledge of the maximum backing store requirement, an implementation can actually use nested backing store allocation – without changing the application visible contract. The nested allocation can also include immortal memory and mission memory. Therefore, the implementation of all three different memory areas can be unified. In this paper we show how a single class can represent all memory types. The memory areas form a hierarchy with respect to lifetime, which can also be represented as nesting level. The initial mission memory is allocated in immortal memory, handler private scopes are allocated in mission memory, and nested private memories within their outer private memory.

The SCJ memory model is the result of discussions within the SCJ expert group. The original proposal [7] by Kelvin Nilsen suggested stack allocation of objects instead of using scoped memories. Stack allocation has been implemented in PERC Pico [1]. The difference between the SCJ memory model and the PERC Pico memory memory model is described in a recent paper [8].

The presented design of the unified memory area for SCJ might guide future implementations of it. As the SCJ specification is still in draft, implementations of it are still rare. A first prototype of a Level 0 implementation on top of OVM [2] and Fiji [9] is presented in [10]. A SCJ like implementation, called predictable Java profile [3], implements a different class hierarchy where RTSJ classes extend SCJ classes. In the mean time, the predictable Java project adapts to the SCJ defined class hierarchy. The upcoming version of PERC Pico [1] will be SCJ compliant. PERC Pico will use the C execution stack for the backing store of scoped memories.[1]

2. SAFETY-CRITICAL JAVA

The SCJ specification [6] is developed within the Java community process (JCP) under specification request number JSR 302. The SCJ specification is a subset of the RTSJ [4]. To cover different criticality levels, SCJ defines three different levels: Level 0 is a single-threaded cyclic executive, Level 1 a single mission with a preemptive scheduler, and Level 2 allows nested missions and usage of an adapted version of RTSJ's NoHeapRealtimeThread. To enforce that all tasks are either periodic or event triggered, the tasks are named handlers in SCJ.

With respect to memory areas, all three levels support immortal memory, mission memory, and thread private scopes. The only difference is that the backing store for private memories for all handlers in Level 0 can be reused.

2.1 Missions and Scheduling

SCJ defines the concept of a mission. A mission consists of a set of handlers (schedulable objects) and a mission memory.[2] The number of handlers is fixed. Handlers are either periodic or event triggered. The event can be triggered from the application or from an interrupt handler.

The mission memory can be used for data shared between tasks. A mission has three phases: initialization, execution, and cleanup. The mission memory is created by the SCJ implementation before executing the initialization phase. Within the initialization phase all shared data is allocated in mission memory and the handlers are created and registered. On the transition to the execution phase all handlers are *started*. During the execution phase no new handlers can be registered or started. In the execution phase temporary objects are allocated in handler private memory. Allocation in mission memory is not prohibited, but strongly discouraged. After the cleanup phase, the mission memory is cleared and a new mission can be started.

A SCJ application does not contain a main() method. Instead, it is represented by a class that extends Mission and implements the Safelet interface. How this *main* class is specified as the SCJ application is vendor specific. Figure 1 shows a minimal SCJ, Level 1 application – an SCJ *Hello World*. We can see that an application needs to define the maximum memory consumption of mission memory, immortal memory, and the maximum memory for all private memories a handler may use. This is the key element to avoid fragmentation and also to allow the nested implementation of memory areas, as described in this paper.

2.2 Memory Model

SCJ defines three memory areas: immortal memory, mission memory, and anonymous private scope memories. Immortal memory is like in the RTSJ for objects that live for the whole application, which might consist of several missions. Mission memory represents a scoped memory that exists for the lifetime of a mission and is the main memory for data exchange between handlers. Each handler has an initial private scope, which is entered on release by the infrastructure and cleaned up at the end of the release. The handler can enter nested private scopes. The private scopes are anonymous, as the application code has no access to a ScopedMemory object that *might* represent this private memory.

In contrast to the RTSJ, SCJ does not require a garbage collected heap. As there is no GC, the *original* heap of a JVM might just be considered as being the immortal memory; or immortal memory plus space for the scopes' backing stores.

The SCJ specification restricts the usage of scoped memory and defines the maximum size of backing store for each thread (handler). Therefore, management of the backing store can be implemented without memory fragmentation.

3. SCOPED MEMORY

The Java language and especially the library and coding style rely heavily on creation of temporary objects. In standard Java the GC is responsible to collect unused objects to free memory. Although real-time GC development advances, real-time programmers are still skeptic on relying on a GC. Therefore, the RTSJ introduced scoped memory for temporary allocated objects. Scoped memory is similar to stack allocation in C, except that it can be shared between threads.[3]

3.1 RTSJ

The RTSJ introduces, additionally to the Java heap, two memory areas: immortal memory and scoped memory. The intention of scoped memory is to allow memory management of dynamically allocated objects without a GC. A thread enters a scoped memory, creates temporary objects, and on exit of the scoped memory the storage of the temporary objects is reclaimed. With RTSJ several threads are allowed to enter a scoped memory concurrently. Only when the last thread exits the scoped memory, the storage is reclaimed. Scopes can also be nested and strict assignment rules have to be checked at runtime to avoid pointers from longer-lived objects to shorter-lived objects.

Scopes are represented by objects of class ScopedMemory. The memory that is used for the objects allocated in the scope is called *backing store*. The notion of the backing store is a little bit vague. Here is the description from RTSJ 1.0.2:

> When a ScopedMemory area is instantiated, the object itself is allocated from the current memory allocation context, but the memory space that object represents (it's [sic] backing store) is allocated from memory that is not otherwise directly visible to Java code; e.g., it might be allocated with the C malloc function. This backing store behaves effectively as if it were allocated when the associated scoped memory object is constructed and freed at that scoped memory object's finalization.

Intuitively one would assume that a nested scope will reserve its backing store from the outer scope and a scope would need to be sized for all allocated objects plus its inner scope memory requirement. However, this intuition is wrong. Backing stores for different scopes, independent of wether they are nested or not, are not related at all. This fact can be derived from "might be allocated with the C malloc function". This definition of the allocation and freeing of backing store easily leads to fragmentation of the memory from where backing stores are allocated.

Furthermore, the backing store is only freed when the scoped memory object is released. The memory cannot be reused by other

[1]Private communication with Kelvin Nilsen.

[2]SCJ Level 2 also allows managed threads.

[3]In principle data on a C stack can also be shared between threads. However, this is considered bad programming style, whilst the RTSJ encourages scope sharing.

```
public class HelloSCJ extends Mission implements Safelet {

          static SimplePrintStream out;

          // Mission methods
          @Override
          protected void initialize () {

                    OutputStream os = null;
                    try {
                              os = Connector.openOutputStream("console:");
                    } catch (IOException e) {
                              throw new Error("No␣console␣available");
                    }
                    out = new SimplePrintStream(os);

                    PeriodicEventHandler peh = new PeriodicEventHandler(
                                        new PriorityParameters(11),
                                        new PeriodicParameters(new RelativeTime(0, 0), new RelativeTime(1000, 0)),
                                        new StorageParameters(10000, 1000, 1000)) {
                              int cnt;

                              public void handleAsyncEvent() {
                                        out. println ("Ping␣" + cnt);
                                        ++cnt;
                              }
                    };
                    peh.register ();
          }
          @Override
          public long missionMemorySize() { return 1000; }

          // Safelet methods
          @Override
          public MissionSequencer getSequencer() {
                    // we assume that this method is invoked only once
                    StorageParameters sp = new StorageParameters(20000, 0, 0);
                    return new LinearMissionSequencer(new PriorityParameters(13), sp, this);
          }
          @Override
          public long immortalMemorySize() { return 100; }
}
```

Figure 1: A safety-critical Java Hello World application

scopes, even when it is known that two scopes, allocated at the same scope levels, are never entered together.

3.2 SCJ

SCJ bases its memory classes on the RTSJ memory classes as they are defined by the upcoming version 1.1 of the RTSJ. In the following we show the public visible memory classes and methods from the packages javax.realtime and javax.safetycritical. Package private methods and methods available in a standard RTSJ implementation, but forbidden in SCJ, are omitted from the description.

3.2.1 javax.realtime

The interface AllocationContext defines the contract for all memory areas. Compared to the RTSJ, the enter() method is not available, as SCJ applications shall not explicitly enter a memory area.

```
public interface AllocationContext {

          public void executeInArea(Runnable logic);
          public long memoryConsumed();
          public long memoryRemaining();
          public Object newArray(Class type, int number)
                    throws IllegalArgumentException;
          public Object newInstance(Class type)
                    throws ExceptionInInitializerError ,
                    InstantiationException, InvocationTargetException;
          public long size ();
}
```

In RTSJ the interface ScopedAllocationContext defines methods to set and get a *portal*. As portals are not part of SCJ, this interface is empty.

```
public interface ScopedAllocationContext
        extends AllocationContext {
}
```

The class MemoryArea is the base class of all RTSJ memory areas (including heap memory). Within SCJ, immortal, mission, and private memories are based on MemoryArea.

```
public abstract class MemoryArea implements AllocationContext {

public static MemoryArea getMemoryArea(Object object) {...}
public void executeInArea(Runnable logic)
        throws InaccessibleAreaException {...}
public Object newInstance(Class type)
        throws IllegalArgumentException, InstantiationException,
        OutOfMemoryError, InaccessibleAreaException {...}
public Object newArray(Class type, int size) {...}
public Object newArrayInArea(Object o, Class t, int size) {...}
public abstract long memoryConsumed();
public abstract long memoryRemaining();
public abstract long size ();
}
```

The class that represents immortal memory is the unchanged version of the RTSJ class ImmortalMemory. However, we assume that the method enter() shall not be part of the SCJ version of ImmortalMemory.

```
public final class ImmortalMemory extends MemoryArea
{
        public static ImmortalMemory instance() {...}
        public void enter(Runnable logic) {...}
        public long memoryConsumed() {...}
        public long memoryRemaining() {...}
        public long size () {...}
}
```

Class ScopedMemory has no public visible methods.

```
public abstract class ScopedMemory
        extends MemoryArea implements ScopedAllocationContext {
}
```

The LTMemory class from the RTSJ is the base class for SCJ memory areas. However, as SCJ applications shall not explicitly enter a memory area, we assume that the enter() method should not be part of the SCJ API. We are further quite skeptic that method resize() should be part of the SCJ API. An indication that this is an error is that method resize() in the super class ScopedMemory is restricted for infrastructure code.

Note, there is a difference between the appendix of the specification [6] and the memory chapter. In the memory chapter LTMemory does not contain any public methods. At least the abstract methods from MemoryArea shall be implemented by this class.

```
public class LTMemory extends ScopedMemory {

        public void enter(Runnable logic) {...}
        public long memoryConsumed() {...}
        public long memoryRemaining() {...}
        public long size () {...}
        public void resize(long size) {...}
}
```

3.2.2 *javax.safetycritcal*

Class ManagedMemory is the base class for mission memory and private memory. Creation of the memory areas is performed by the SCJ implementation. A SCJ application cannot directly instantiate any scoped memory. If a handler wants to enter a nested private memory, it invokes enterPrivateMemory() and the nested scope is created by the infrastructure. This mechanism ensures that a private scope cannot be entered by more than one handler.

However, with getCurrentManagedMemory() a reference to the private memory can be obtained and shared via mission or immortal memory. Therefore, allocateInArea() has to check if the private memory belongs to the invoking thread.

Note, the method getMaxManagedMemorySize() should probably not be part of the public API. It is in the appendix of the specification, but not explained. It is assumed that the specification of ManagedMemory will change in the future and the following class definition might already be out of date when the paper is published.

```
public abstract class ManagedMemory extends LTMemory {

public static ManagedMemory getCurrentManagedMemory() {...}
public static long getMaxManagedMemorySize() {...}
public static void enterPrivateMemory(long sz, Runnable r) {...}
public long size () {...}
}
```

The class, which represents mission memory, has only two public methods. However, we assume that the public enter() method is an error in the specification. It should actually not be part of the SCJ API, as mission memory is entered by infrastructure code. We also do not see any good reason why this class shall support a toString() method. If both public methods are dropped, the MissionMemory class can be dropped from the public API, similar to the PrivateMemory class.

```
class MissionMemory extends ManagedMemory {
        public final void enter(Runnable logic) {...}
        public String toString () {...}
}
```

Class PrivateMemory represents the scoped memories that are used by the handler. However, the class has no public visible methods as this class shall not be used directly by an application. In that case it might be reasonable to drop this class from the public API and just explain the concept of private memories, which is behind the method enterPrivateMemory().

```
public class PrivateMemory extends ManagedMemory {
}
```

3.3 Discussion

Due to the inheritance of memory area classes from the RTSJ and specialization for mission and private memories in SCJ, the class hierarchy is quite large. As some features, e.g., portals and invocation of enter(), are not available within SCJ, some of the classes are basically empty.

One challenge an SCJ implementation faces is that the classes for the memory management live in different packages. Java does not have the *friend* concept of C++ for fine grain access restriction. Therefore, if a class in package javax.safetycritical needs to invoke a method from a superclass in javax.realtime this method has to be public.

As the allocation methods and executeInArea are concrete methods of MemoryArea, the *meat* of the implementation will be there. However, the enterPrivateMemory() method in ManagedMemory will need to call into MemoryArea for entering the memory area. The current approach within SCJ is that there are public methods in the RTSJ classes (e.g., enter()). These methods are marked with

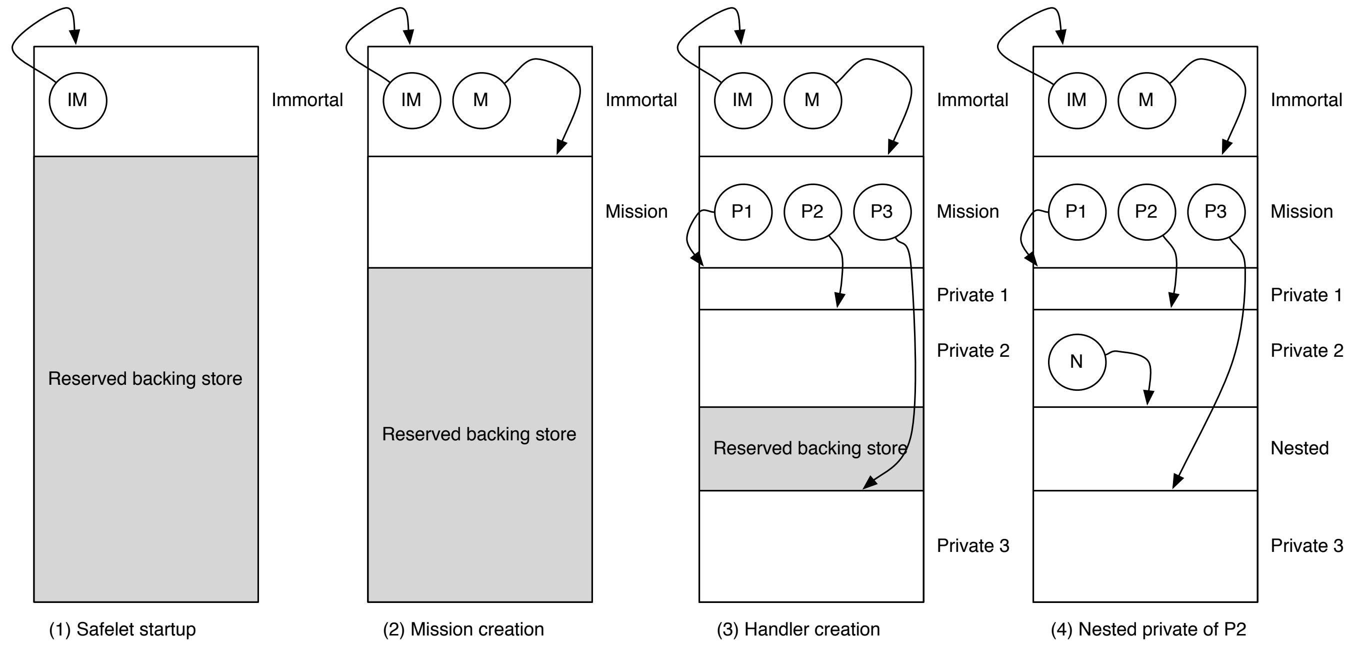

Figure 2: Memory layout with the single Memory class for all SCJ memory types

an annotation that application code is not allowed to invoke them. A checking tool will enforce these annotation rules.

An alternative would be to define just managed memory and immortal memory classes in javax.safetycritical that do *not* inherit from the RTSJ. In that case there is no package crossing and Java's protection scheme of public and package private methods is enough. There might be the counter argument that the reference implementation (RI) of SCJ shall execute on RTSJ. However, the SCJ memory area classes can simply use the RTSJ classes in the form of composition instead of inheritance [5].

With the current solution of using subsets of the RTSJ classes, the execution of the RI on a plain RTSJ JVM has following two issues: (1) the SCJ programmer does not see which classes and methods from the RTSJ are legal in an SCJ application and which not; (2) as a standard RTSJ is not SCJ aware their classes do not contain the annotations and the checker approach fails.

A clear separation of SCJ and RTSJ classes would actually simplify the implementation and the usage of the RI. A SCJ application is compiled against the javax.safetycritical classes of the RI, which enforces using only the SCJ API.[4] The RI itself is compiled against the javax.realtime classes.

4. MEMORY MANAGEMENT DESIGN

To allow a fragmentation free implementation of scoped memories, the maximum size of backing store requirements needs to be specified. The SCJ application has to specify the size of immortal memory, the size of the mission memory, and the size of 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 for all three memory types of SCJ. The class Memory is a system class and is used to implement ImmortalMemory, MissionMemory, and PrivateMemory.

Memory has two size parameters, one for the memory area it represents, and the second for the size of the nested memory areas. At JVM start the Memory object that represents immortal memory is created. At SCJ application start, when the immortal memory size is requested from the Safelet, the size parameter for the immortal memory is set and the rest of the available memory is reserved for the nested memories. In the next step, a mission memory object is created within immortal memory. This mission memory object consumes the entire remaining backing store, reserves mission memory size for itself, and the remaining storage for private memories.[5]

The individual handlers have a StorageParameter in their constructors. Besides stack sizes, this parameter contains the size of the initial private memory and the maximum backing store for the nested private memories of that handler.

With this information the first private memory object can be sized with its own size and the additionally required backing store for the nested private memories. A nested private memory consumes all the backing store of the outer memory, reserves memory

[4]The author has seen students struggling to build test cases for SCJ with a version of the RI on top of the RTSJ. They used RTSJ features without noticing that those features are not allowed in a *real* SCJ runtime.

[5]The size of all memory that is available for an SCJ JVM cannot be specified within a SCJ application. A SCJ JVM executing directly on the hardware might use all available physical memory. If the JVM is a process executing on an RTOS, the maximum memory requirement may be given as a command line parameter at the JVM start.

for the local allocation, and provides the backing store for the further nested private memories.

Figure 2 shows how the memory data structure evolves over the lifetime of an SCJ application. Boxes represent memory areas; circles represent memory area objects. IM is immortal memory, M mission memory, Px are initial private memories for handlers, and N is a nested private memory of handler 2.

The first subfigure shows the memory division at Safelet startup. The immortal memory object IM is already created at JVM startup and originally all available memory is allocated for immortal memory. At creation of the SCJ application the immortal memory is resized and the remaining memory is reserved backing store for nested memories. Subfigure (2) shows the memory layout after creation of mission memory. The mission memory object resides in immortal memory; the space for the mission memory is allocated; and all remaining memory is allocated as reserved backing store for the mission memory. At handler creation, shown in subfigure (3), the handlers memory requests are allocated from the reserved backing store of the mission memory. In this example three handlers are shown with their initial private memories P1, P2, and P3. Only handler 2 has reservation for a nested private memory, which is part of its private memory P2. In the last subfigure (4) the memory is shown during a release of handler 2. Handler 2 has entered a nested private memory. The object for this nested memory is N and the backing store is allocated from the reserved backing store of private memory P2.

4.1 Implementation

We base our implementation on the Java processor JOP [11]. Implementing the SCJ compatible memory management is the first step towards a full SCJ implementation on JOP. As part of the JVM is implemented in Java, the low-level implementation of the memory class can also be implemented in Java. In a *normal* JVM this kind of system level code is usually programmed in C. The presented concept is independent of the implementation form of the JVM.

Figure 3 shows a sketch of the Memory class. We use plain integer fields to represent memory addresses. For example, startPtr contains the address of the start of the memory area. Allocation in the memory area is just an increment of the allocation pointer allocPtr; the end of the memory for the local allocations is stored in endLocalPtr. Following this local area is the backing store for the nested memories till endBsPtr. Nested memory areas are allocated by changing the backing store allocation pointer allocBsPtr.

Each memory area has a nesting level: starting with 0 for immortal memory, 1 for the first mission memory, and 2 and higher for the private memories. As only immortal memory and mission memory is shared between threads, the memory hierarchy, seen by a handler, is a single stack of memories. Therefore, pointer assignment checks can be simplified to just check the nesting level.

The first memory area that is created is the immortal memory and is special as the object that represents it is itself allocated in immortal memory. The creation of this special memory is performed as part of the JVM startup code by calling getImmortal. This singleton object is stored in the static reference immortal.

Each memory object has a reference to its parent, the outer scope. The memory object also contains a field for a nested memory area. An object for a nested private memory is allocated on the first invocation of enterPrivateMemory() from that level. To not leak memory area objects on entering nested private memories in a loop, the object for the nested memory is reused on a repeated enter. It is possible that a handler enters in one release several private memories with different sizes. Therefore, the memory object needs to be mutable to adjust it to different storage requirements.

```
public class Memory {

        /** Start address of memory area */
        int startPtr;
        /** Allocation pointer */
        int allocPtr;
        /** End of area for local allocations */
        int endLocalPtr;
        /** Allocation pointer for the nested backing store */
        int allocBsPtr;
        /** End of backing store */
        int endBsPtr;
        /** Parent scope */
        Memory parent;
        /** Nesting level */
        int level;
        /**
         * A reference for an inner memory that shall be reused
         * for enterPrivateMemory.
         */
        Memory inner;
        /**
         * The singleton reference for the immortal memory.
         */
        static Memory immortal;

        Memory() {}

        /**
         * Create a Scope object that represents immortal memory.
         */
        static Memory getImmortal(int start, int end) {...}

        /**
         * Create a scope of the specified size and
         * bsSize backing store.
         */
        Memory(int size, int bsSize) {...}
        /**
         * Create a scope and use all available backing store.
         */
        Memory(int size) {...}

        void enter(Runnable logic) {...}
        void executeInArea(Runnable logic) {...}

        /**
         * Return the memory region which we are currently in.
         */
        static Memory getCurrentMemory() {...}
        /**
         * This is SCJ style inner scopes for private memory.
         */
        void enterPrivateMemory(int size, Runnable logic) {...}
}
```

Figure 3: The Memory system class for the SCJ memory areas

4.2 Interaction with Handlers

Each SCJ handler, which is implemented by a periodic real-time thread on JOP, contains a reference to its current allocations context. That allocation context is represented by a Memory object. Object creation, which is also implemented in Java on JOP, looks up the thread's allocation context on the execution of bytecode new and the versions for array allocations. One optimization would be to store the current allocation context in a static variable in the thread data structure and change it on a thread switch.

4.3 Nested Missions

The proposed way to organize the memory areas is designed from an SCJ Level 1 point of view. However, it will also work for Level 2 implementations. After allocation of the first mission memory and the reservation for all handlers memory needs, some memory in the reserved backing store in the first mission must remain. A nested mission memory (plus the backing store for its handlers, plus the backing store for further nested missions) is allocated in the remaining backing store of the outer mission, similar to the backing stores for handlers of the outer mission are allocated.

To enable reservation of backing store for nested mission, the sizes of the missions is included in the storage parameters of mission sequencers. The mission sequencers are created at initialization of the outer mission. Therefore, all storage requirements are available at initialization time.

For a Level 1 implementation of SCJ, the storage parameter of a mission sequencer is redundant. The implementation can use all remaining memory after sizing of the immortal memory for mission memory and private memories. However, to enable reasoning about the memory requirements of an SCJ Level 1 application, the storage parameter for the mission sequencer is also used in Level 1. It has to be set to the maximum size of any sequenced mission, including backing store for the mission memory and the handlers.

4.4 Discussion

The current version of SCJ allows allocation of objects in immortal memory and mission memory also during the execution phase. However, when a periodic handler allocates objects on each release, this is a memory leak that will result on an out-of-memory exception.

Safety-critical programs are usually very conservative with statically allocated data and allocate it within the initialization phase. If SCJ would disallow allocation in mission or immortal memory during mission, the allocation operation can further be simplified. In that case only a single thread is allocating at any time in any memory area. Therefor, the synchronization within a new operation can be avoided.

5. CONCLUSION

In this paper we presented the current memory model of the safety-critical Java specification. The main differences to the RTSJ memory model are: single nesting of scopes, thread private memory areas, and that the application has to size the maximum backing store for all nested private memories. This model avoids fragmentation of backing store and allows for a simple implementation of the memory areas.

We also presented a unified memory class that can be used to represent immortal, mission, and private memory. This class contains, besides the backing store of the memory area it represents, also the backing store for nested memory areas. As scoped memories are not explicitly created in an SCJ application, the memory class contains already the reference for a nested scope. The object to represent a nested private memory is allocated on the first enter of a nested private memory, but reused when repeatedly entered during a single release of a handler. To allow different sized scopes in the same release, the size of the private memory needs to be mutable.

6. ACKNOWLEDGMENTS

I would like to thank the JSR 302 expert group, and especially Kelvin Nilsen, for the long (and sometimes intense) discussions of the memory model for SCJ at enjoyable expert group meetings. Nevertheless, this paper does not necessarily represent the opinion of the expert group and all mistakes remain in the responsibility of the author. I would also like to thank the anonymous reviewers for their very detailed reviews.

The author would like to thank Anders P. Ravn for discussions on the memory design for SCJ. This work is part of the project CJ4ES and received partial funding from the Danish Research Council for Technology and Production Sciences under contract 10-083159.

7. REFERENCES

[1] Aonix. Perc pico 1.1 user manual. http://research.aonix.com/jsc/pico-manual.4-19-08.pdf, April 2008.

[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. *Trans. on Embedded Computing Sys.*, 7(1):1–49, 2007.

[3] T. Bøgholm, R. R. Hansen, A. P. Ravn, B. Thomsen, and H. Søndergaard. A predictable java profile: rationale and implementations. In *JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 150–159, New York, NY, USA, 2009. ACM.

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

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

[6] 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, 2011.

[7] K. Nilsen. Draft safety critical Java standard. available from http://research.aonix.com/jsc/, April 2004.

[8] K. Nilsen. Harmonizing alternative approaches to safety-critical development with Java. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2011)*, 2011.

[9] F. Pizlo, L. Ziarek, and J. Vitek. Real time java on resource-constrained platforms with Fiji VM. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2009)*, pages 110–119, New York, NY, USA, 2009. ACM.

[10] A. Plsek, L. Zhao, V. H. Sahin, D. Tang, T. Kalibera, and J. Vitek. Developing safety critical Java applications with oSCJ/L0. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2010)*, pages 95–101, New York, NY, USA, 2010. ACM.

[11] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.