Hardware Support for Safety-Critical Java Scope Checks

Juan Ricardo Rios
Department of Informatics and Mathematical Modeling
Technical University of Denmark
Lyngby, Denmark
Email: jrri@imm.dtu.dk

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

***Abstract*—Memory management in Safety-Critical Java (SCJ) is based on time bounded, non garbage collected scoped memory regions used to store temporary objects. Scoped memory regions may have different life times during the execution of a program and hence, to avoid leaving dangling pointers, it is necessary to check that reference assignments are performed only from objects in shorter lived scopes to objects in longer lived scopes (or between objects in the same scoped memory area). SCJ offers, compared to the RTSJ, a simplified memory model where only the *immortal* and *mission* memory scoped areas are shared between threads and any other scoped region is thread private. In this paper we present how, due to this simplified model, a single scope nesting level can be used to check the legality of every reference assignment. We also show that with simple hardware extensions a processor can see some improvement in terms of execution time for applications where cross-scope references are frequent. Our proposal was implemented and tested on the Java Optimized Processor (JOP).**

***Keywords*-Certification, Safety-Critical Java, Scoped Memory, Reference Assignment Checks, Java Optimized Processor**

I. Introduction

In order to take advantage of the benefits of an object oriented programming language such as Java into the context of real-time systems, the main sources of indeterminism introduced by standard Java implementations should be avoided. Operating-system scheduling, priority inversions, class operations (loading, initialization and compilation), the garbage collector (GC), and the use of shared resources affect the execution time of an application.

Automatic garbage collection is one of the primary sources of unpredictability because it can introduce inaccurate time bounds that tasks with hard deadlines may not be able to tolerate [1]. The Real-Time Specification for Java (RTSJ) [2] provides a framework intended to cope with such sources of indeterminism by introducing features like new types of threads and memory management models. RTSJ programs can avoid GC delays by using two memory regions which are not garbage collected: *Immortal Memory* and *Scoped Memory*. Objects allocated in Immortal Memory will be reclaimed only when the Java Virtual Machine (JVM) terminates. Scoped Memory regions are created and reclaimed at run time and objects allocated into them are collected when the scope is no longer active. A scoped memory region is no longer active when it has no schedulable objects executing in it.

Despite it's restricted memory model as well as other new characteristics, RTSJ was not intended to be used in safety-critical applications, as some features are too complex and thus hard to certify under standards such as the DO-178B. Safety-critical Java (SCJ) [3] on the other hand, built as a more restricted subset of RTSJ, introduces a programming and memory model aimed at simplifying the process of certification.

SCJ introduces the concept of *missions*. A mission consists of a bounded set of schedulable objects or handlers. SCJ defines three levels with varying complexity. They are referred in [3] as Level 0 (L0), Level 1 (L1) and Level 2 (L2). For a L0 application, a single mission with a cyclic executive is used (periodic handlers), L1 uses a single mission where handlers can be executed concurrently (periodic and aperiodic), and L2 extends the model by allowing nested missions.

In the memory model of SCJ, immortal memory is kept and two additional scoped memories are introduced: *mission memory*, shared between all schedulable objects whose lifetimes should cover the whole mission's life time and *private memories*, used to hold any object created by the schedulable objects. One important restriction in SCJ is that, in contrast to RTSJ, private memories can be entered only by a single schedulable object.

Even though the simplifications that SCJ provides to the memory model, the development of applications is not a trivial task. Programmers have to be aware of where objects are allocated [4], thus increasing the potential for memory leaks and dangling pointers. Referential integrity has also to be guaranteed by following a set of assignment rules, simpler than those for RTSJ. In this paper, we examine how, given the simplified memory model of SCJ, a single scope nesting level can be used to check the legality of every reference assignment. We also show that with simple hardware extensions we can check reference assignments without the overhead of a software based solution and improve the execution time of applications with frequent cross-scope references. Our proposal was implemented and

tested on the Java Optimized Processor (JOP) described in [5].

The rest of this paper is organized as follows. Section II presents previous work related to illegal assignment checks for scoped memory regions, both from software and hardware based solutions. Section III explains the simplifications of the memory model in SCJ; our concrete implementation is presented in Section IV. Evaluation results are presented in Section V and the paper is concluded by Section VI.

II. Related Work

The method we used to check for the legality of a reference assignment is based on the execution of write barriers whenever reference assignment instructions are executed [6], [7] .

In [6], as part of the write barrier, the scope stack of a thread is scanned to check that the memory area of the destination of a reference assignment is deeper nested than the memory area of the source of the reference. In this implementation, the time it takes to scan the stack is proportional to the number of nested scoped levels. To bound the execution time of the check, the authors limit the levels of nested scopes and in a latter work [7], the performance is improved with the aid of the write barrier support provided by the picoJava-II microprocessor as well as with specialized hardware. The scope stack is stored in an associative memory which allows a faster scanning of the hierarchy of nested scopes.

SCJ is very recent and the work presented in [8] describes a reference implementation of SCJ at Level 0. As part of this implementation, the write barrier works by locating and comparing the memory areas where objects are stored. The memory is organized as a continuous region that starts with the Immortal Memory then continues with Mission Memory followed by any private memory or stack of private memories. Lower memory addresses correspond to longer lived memory regions. The check is performed by determining the block of memory where both objects involved in a reference assignment are allocated. A second step might be needed to recover scope information of each object as both objects may reside in the same scope. We do not use the address value of each object itself because that would involve checking that the objects are allocated within a certain range, and that would be more time consuming as the boundaries of the memory region where the objects are allocated need to be known.

In [4], the authors give an analysis algorithm that statically guarantee that no illegal reference assignments can happen. The authors introduce the concept of Scoped Types as a way to encapsulate scoped objects. Scoped and Portal classes are defined and associated to their defining packages. Nested scopes are in turn associated with nested packages. Accessibility of a scoped class is restricted to instances of classes allocated in the same or nested scopes.

Verifying SCJ memory safety can be done statically by correctly using the annotation model defined by SCJ in [3]. For example, in [9], annotations are used as part of a two step verification process. In the first step, the scope tree is constructed and checked for errors and in the second step, the tree constructed in the previous step is used to check a set of rules for annotated and unannotated classes.

III. Scope Checks in Safety-Critical Java

SCJ uses the scoped memory model as introduced by RTSJ to allow creation and reclamation of objects during the mission phase without the help of a garbage collector. In SCJ, use of the heap has been abandoned and three different memory areas are defined: the immortal memory, the mission memory, and private scopes. Immortal and mission memory are shared between the concurrent handlers and private scopes are handler local. Any private scoped memory region can only be entered by a single handler and it can only be entered from the memory area where it was created. Objects allocated in immortal memory remain valid until the VM finishes, objects in mission memory remain valid for the duration of the mission and objects in private scopes are reclaimed when the handler finishes execution.

A. Memory Areas and the Scope Stack

The restricted scope model of SCJ significantly reduces the complexity of the assignment checks. In SCJ L0 and L1, where no nested missions exist, the hierarchy of nested scopes can grow only in a linear shape. This is because scopes are private to each schedulable object and the `enter()` method is not available, as SCJ applications have to be prevented from explicitly entering a memory area [10]. Restricting the use of the `enter()` method avoids breaking the linear scope hierarchy because this eliminates the creation of branches in the parenting relationship between scopes if it is used after the `executeInArea()` method.

Furthermore, the shared memory areas (immortal and mission memory) are at a fixed nesting level within this stack and it is not possible that two handlers see the same scope at different levels.

Thus, a unique nesting level can be assigned to each scope and the assignment check is reduced to a comparison between the nesting levels of the source and destination objects in a reference assignment. For a SCJ implementation, the following scope levels can be assigned to each memory area: immortal memory is level 0, mission memory level 1, the initial private scope of a handler level 2, and nested memories increase the level by 1.

B. Scope Checks

Our approach to detect illegal assignments is similar to the approach described in [7] that consists in taking actions upon the execution of instructions that store an object's reference pointer into another object's field or array element. To this

end, we need to know two things: the instructions which cause references between objects, and the location (scope level) of each object within the memory hierarchy (for SCJ, a linear stack).

Field assignment instructions are the `putfield` and `putstatic` bytecodes. We also have to consider the `aastore` bytecode since an illegal assignment may result from storing a reference to an object into an array. When these bytecodes are executed, the stack has the following information [11]:

$$\texttt{putfield} : ...objectref, value$$

$$\texttt{putstatic} : ...value$$

$$\texttt{aastore} : ...arrayref, index, value$$

Note that for `putfield` and `putstatic`, *value* can be any data type allowed by the JVM that matches the type of the destination field pointed by *objectref*. Nevertheless, illegal assignments can be produced only when *value* contains a reference to another object. That means we do not need to check every field access.

At object creation, whenever the `new` and `newarray` bytecodes are executed, every object is associated with the level of the memory region where it was created (the current allocation context). This information can be stored in the objects auxiliary data (e.g., in the object header).

As mentioned before, compared to RTSJ, the memory model of SCJ allows for enforcing simpler assignment rules. In our case, depending on which of the previously mentioned bytecodes is executed, one of three simple rules have to be checked to detect illegal memory references. Recall that shorter lived scopes are deeper nested in the stack hierarchy, and so they are associated with higher level numbers. These rules are described below and summarized in Table I:

1) The `putfield` bytecode stores *value* into a field of an object whose reference is *objectref*. The reference check has to verify that the level of *value* is less than or equal to the level of *objectref*.
2) For `putstatic`, *value* cannot be in a scoped region, hence the level of *value* (pointer to the object) needs to be checked against level 0, as all static fields reside in immortal memory.
3) The reference check for `aastore` is similar to the test for `putfiled` but using the level of *arrayref* instead of the level of *objectref*.

Table I
REFERENCE ASSIGNMENT CHECK RULES

Bytecode	Scope check rule
`putfield`	Level(*value*) $\leq$ Level(*objectref*)
`putstatic`	Level(*value*) $= 0$
`aastore`	Level(*value*) $\leq$ Level(*arrayref*)

C. Header Based Scope Checks

Since the JVM specification does not require any particular internal structure for objects [11], the additional information needed can be stored as part of the header of the object itself. Following the idea exposed in the previous section, at creation time a particular field of the object's or array's header is associated with the scope level, and whenever a reference assignment instruction is executed, this value can be extracted from the object's header.

The advantage of this approach is that it can be used by any JVM implementation and is not restricted to be used just in JOP.

D. Pointer Based Scope Checks

In an embedded application the maximum memory is usually less than 2 GB and references are aligned to at least 32-bit word boundaries. Therefore, some bits of an object or array reference are unused and the scope level can be encoded in those unused bits. Having the scope level already in the reference saves at least two additional memory reads on the assignment check.

Similar to the object header based approach, the pointer based scope checks can be used in a software JVM when the maximum memory is restricted.

E. Hardware Based Scope Checks

The two methods described in the previous sections are suitable for any JVM, as the data structures required are not implementation specific. However, a hardware based implementation of the scope checks implies certain knowledge of the underlying construction of the VM and will be implementation specific (e.g. [7] for the picoJava-II microprocessor). For our JOP based implementation, we will use the pointer based scope check. Since JOP implements most of the VM in hardware, we have an easy access to the scope level information encoded in the reference to an object.

IV. IMPLEMENTATION

For an evaluation of the concept, we have implemented the software and hardware based scope checks in the Java processor JOP [5]. The hardware based scope checking is clearly only an option for a Java processor. However, the scope level encoding in the reference can also be used on a software JVM.

A. JOP Architecture

JOP is an implementation of the JVM in hardware. The JVM bytecodes are the instruction set of JOP. However, as some bytecodes are quite complex, the effective execution is in microcode. Bytecode instructions are translated to microcode sequences. For the affected bytecodes for scope

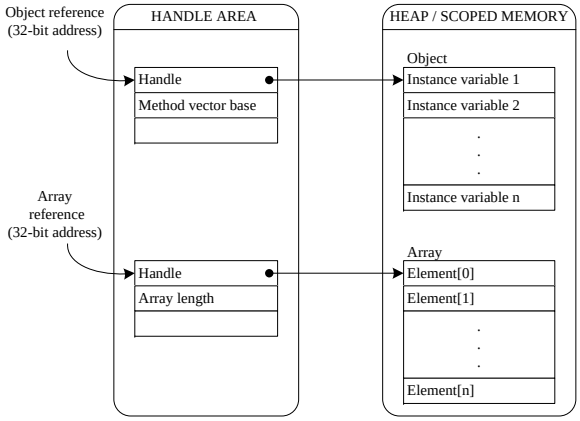

Figure 1. Object and Array format

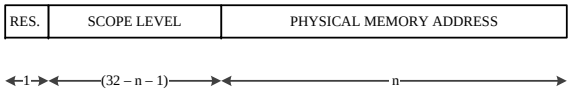

Figure 2. Object reference pointer

checks, most of the logic is actually in the memory management unit (MMU), which is triggered by short sequences of microcode instructions.

The object and array layout of JOP uses an indirection, called a handle. This indirection simplifies a compacting garbage collector, as only one word in memory needs to be updated on an object or array copy. Furthermore, the handle area also holds information, which is usually part of the object header, e.g., type information, GC information, size, etc. When an object is created, a reference to the object or array handler is obtained. The reference is pushed into the stack before an instruction can operate on it. Fig. 1 illustrates this concept.

Since SCJ does not allow the use of a GC, the use of the handler indirection for the method described in Section IV-B1 can introduce inefficiency in the field access due to the extra memory access through the handler indirection. Nevertheless, the layout is kept since JOP also allows the execution of non SCJ applications that need a GC.

JOP's application build tool provides an optimization that substitutes putfield bytecodes of references with the special bytecode versions `putfield_ref` and `putstatic_ref`. Therefore, scope checks are only executed in these special bytecodes and field access of primitive values have no additional overhead.

Addresses in JOP are 32-bit wide and memory is addressed as 32-bit data. This combination allows a maximum heap size of 16 GB of memory, which is more than what an embedded application will probably require. We can take advantage of this and use some of the upper bits of the address field, which is also the reference pointer to

```
private static void f_putfield_ref(int ref, int value,
                                   int index) {

  int ref_level = Native.rdMem(ref + GC.OFF_SCOPE);
  int val_level = Native.rdMem(value + GC.OFF_SCOPE);
  if (val_level > ref_level){
    IllegalAssignmentError();
  }
}
```

Figure 3. Handle based scope check in JOP

an object's handler (see Fig. 1), to store the scope level information of each object. We do this at creation time (`new` and `newarray`). The resulting reference pointer is shown in Fig. 2.

B. Software Based Scope Checks

Following the strategy described in Section III-B, we implemented two versions of the reference assignment in software. One version uses the handler of an object/array and the other uses the reference pointer of the object's handler. The following sections describe both implementations.

1) Using the Object Handle: The scope level can be stored in a field of the object's header. Within the SCJ implementation on JOP, the GC is disabled and therefore, we can reuse one of the handle fields used by the GC in the handle based object layout (see Figure 1) to store the scope level.

When performing a reference assignment check at run time, we can read the scope level information by reading from memory the location pointed by the handler's reference pointer plus an offset. This offset will be the position of the field in the handler that was used to store the scope level. A simple arithmetic comparison is then needed to check the validity of the assignment. Fig. 3 shows the scope check based on the scope level in the object handle.

2) Using the Object Reference: An additional software version of the reference assignment check was implemented with the aid of Java functions called every time the `putfield`, `putstatic` and `aastore` bytecodes are executed. These functions take the references to the objects (*objref/arrayref* and *value* in Table I) as arguments and test the legality of the assignment based on the scope level of the arguments. For this particular implementation, we used 7 of the 32 bits of the object's reference pointer to encode the scope level. This allows for the use of 128 levels of nested scopes. The code for this implementation is shown in Fig. 4.

C. Hardware Based Scope Checks

Since in JOP most of the JVM is implemented in hardware, all the information to perform the scope checks is now available directly in hardware. By adding the scope level value in the reference pointer, the level of *objectref/arrayref*

```
private static void f_putfield_ref(int ref, int value,
                                   int index) {
  if ((value >>> 25) > (ref >>> 25)) {
    IllegalAssignmentError();
  }
}
```

Figure 4. Reference based scope check in JOP

and *value* can be easily recovered by taking the upper bits from the internal registers that hold the values of the top of the stack (TOS) and the next of the stack (NOS) during the execution of the relevant bytecodes.

Reference assignment checks can now be efficiently performed since this is reduced to a simple arithmetic comparison between scope levels, which can be implemented in dedicated and simple hardware. In addition, because we have all the information available during bytecode execution, the check itself can be included as part of the bytecode execution performed in the Memory Management Unit (MMU). Recall that the check shall only be done when it is an assignment of a reference, so the hardware needs to know if it is a reference or a primitive data. That information is already available in the `putfield_ref`, `putstatic_ref` and `aastore` bytecodes and we only need a new microcode instruction to signal the memory controller that whenever the mentioned instructions are executed a reference assignment will take place and thus the levels need to be checked.

As the check is performed as part of the bytecode execution, there is only a minimal overhead of one clock cycle, which is the time needed to indicate a reference assignment bytecode to the MMU. There is also the extra cost of adding the scope level information at object creation time. Nevertheless, besides being an operation with low frequency of execution [12], it can be performed in constant time so it does not complicate WCET calculations.

To indicate that an illegal assignment has occurred, a flag is raised inside the memory controller which can be used to throw an `IllegalAssignmentError` as defined in [3].

For our pointer and hardware based solution, using some bits of the reference pointer does not affect the behavior of the system because internally, a memory address is trimmed to the width of the memory bus attached to the system. Therefore the scope level information is not used when accessing objects in memory. On the other hand, the 32 bits are fully stored in memory so we are saving the scope level together with the object's reference pointer.

The number of nested scopes that can be used will be limited by the number of bits we use to encode the scope level. Using more bits for the encoding of the level will result in a reduced amount of memory that can be addressed by the system.

Table II
EXECUTION TIME (IN CLOCK CYCLES) OF THE THREE BYTECODES IMPLEMENTING THE SCOPE CHECK METHODS DESCRIBED IN THIS PAPER.

Bytecode	**Execution Time**			
	Handle	*Reference*	*Hardware*	*None*
`putfield_ref`	185	169	13	12
`putstatic_ref`	163	157	8	7
`aastore`	179	163	15	14

V. EVALUATION

Our proposal was tested by implementing in JOP the three scope check methods described in this paper. The scope check was implemented as part of the special bytecode instructions `putfield_ref`, `putstatic_ref` mentioned in Section IV-A and on the `aastore` bytecode. According to [2], an `IllegalAssignmentError` should be thrown if there is an illegal reference assignment, hence taking the branch of a failed test is not considered as part of the overhead in any of the software checks.The cost in hardware of adding the scope checks is practically negligible as it adds around 32 look up tables and 10 registers when implemented in an Altera DE2-70 FPGA board. This is an increase in about 4% of the Memory Management Unit (MMU) where most of the logic is implemented.

In our implementation, scope checks are executed as part of the reference assignment bytecodes. Each bytecode has a different individual implementation and therefore a different individual execution time. Table II shows the execution times in clock cycles of each of the three bytecodes with the different options to perform the scope checks. The values were obtained using ModelSim by performing a hardware simulation of JOP running a micro benchmark with different types of reference assignments. From the table it can be seen that the hardware version is around 10 times faster than the two software versions.

To evaluate the improvement within an application, two benchmarks were used. The first is an application inspired in one of the examples presented in [13], where a probe containing certain sensors is used to scan the walls of a well. For our example, we have an application that periodically checks a set of artificial sensors simulated in hardware using hardware objects as described in [14]. The application creates an array of objects to hold the sensor's results in a scoped memory. It then enters a nested scope where the actual sensor objects are created, each sensor performs readings and basic calculations and stores back the results into the array of objects located in the parent scope. The second application is a scoped version of an N-body simulation (gravitational force). It uses a "brute-force" algorithm, where the resulting force on each body is computed as the result of the field interaction of each body.

Table III
SENSOR APPLICATION EXECUTION TIME INCLUDING THE THREE VERSIONS OF THE SCOPE CHECKS.

Sensors	Reference count	Execution time (ms)			Improvement
		Handle	*Reference*	*Hardware*	
50	864	14.05	13.90	11.35	18.35%
100	1714	28.00	27.60	22.65	17.93%
150	2564	42.15	41.40	33.85	18.24%
200	3414	56.10	55.15	45.10	18.22%
250	4264	70.20	68.90	56.45	18.07%
				average	18.16%
				std. dev.	0.16%

Table IV
N-BODY SIMULATION APPLICATION EXECUTION TIME INCLUDING THE THREE VERSIONS OF THE SCOPE CHECKS.

Bodies	Reference count	Execution time (s)			Improvement
		Handle	*Reference*	*Hardware*	
2	619	1.635	1.635	1.634	0.10 %
3	1219	4.045	4.045	4.042	0.08 %
4	2019	7.519	7.516	7.511	0.07 %
5	3019	12.050	12.052	12.040	0.10 %
6	4219	17.648	17.648	17.634	0.08 %
				average	0.09 %
				std. dev.	0.01 %

This application was adjusted to have a number of cross-scope references proportional to the number of bodies and the total time steps used for the simulation.

The detailed results of the two benchmarks are summarized in Tables III and IV. The improvement gain on both tables is the difference in execution times of the hardware based implementation and the object reference software implementation (the reference based implementation is slightly faster than the handler based). Table III shows an improvement gain of around 18% for the Sensor application while Table IV shows that the improvement gain is roughly 0.09% for the N-Body simulation (see Section V-A). Fig. 5 presents a comparison of the overall execution time of both applications.

A. Discussion

The focus of this paper was to evaluate the benefits of providing hardware support for time critical operations such as reference scope checks in the context of Safety Critical Java. We found that our hardware implementation adds basically no timing overhead at a negligible hardware cost to this operation because the execution of the write barrier is part of the execution of the bytecode itself.

Nevertheless, cross-scope references between objects may not be so frequent in a real application and hence the hardware scope check will not make a major improvement in the execution time. In the two examples provided, the execution time was measured with different number of cross-scope references. For the Sensor application, increasing the number of sensors increases the number of references and in the N-Body simulation increasing the number of bodies and/or the time steps for the simulation increases the reference count. For the sensor application, it doesn't make too much sense to have hundreds of sensors since a real application most likely wont need that much. In the N-body simulation it does make sense to have many bodies and hence many reference assignments but their overhead is clouded by the execution time of the other operations.

In the Sensor application we can see an improvement gain of around 18% because it does not have heavy computations, it only reads data from the simulated sensors and then performs simple calculations on the sampled sensor data. With the N-Body simulation application the improvement is very small as most execution time is spent in floating point operations, which are slow on JOP.

It is also important to mention that the methods described in this paper were considered for SCJ L0 and L1 applications where the nested scope hierarchy can grow only in a linear shape. However, for an SCJ L2 application this hierarchy will have a tree shaped form because of the nested missions. Nevertheless, when the inner mission memory cannot be leaked to an outer mission handler, the level check is still an option for L2. In addition, one should be able to guarantee that parallel missions cannot reference one another.

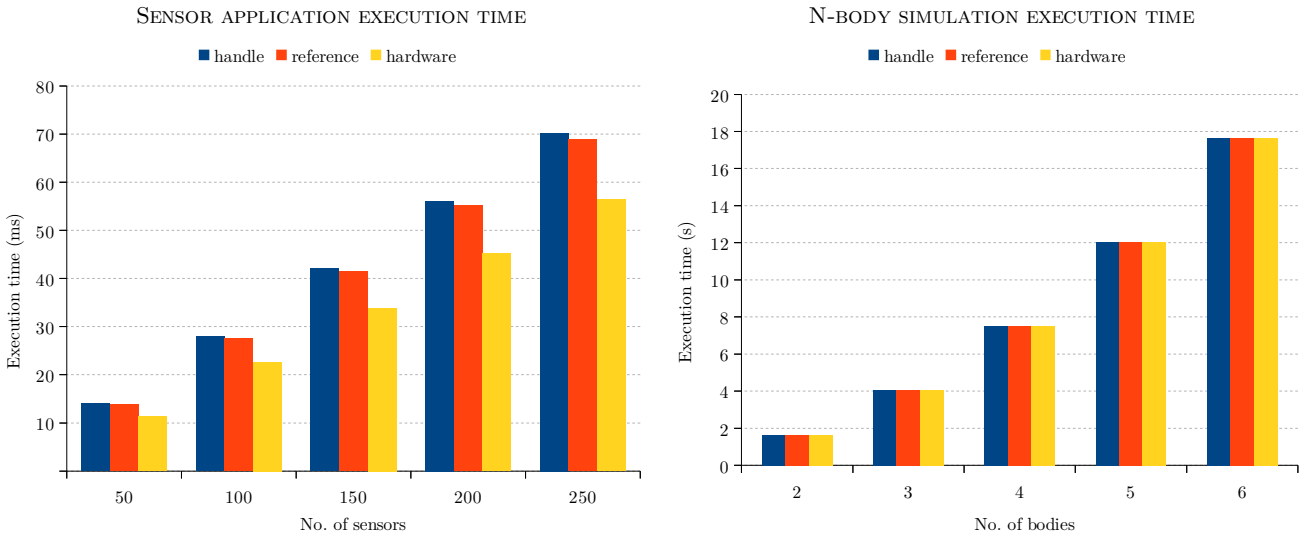

Figure 5. Execution times of the two applications used to test the implementation

VI. Conclusion

The scoped memory model in Safety-Critical Java is restricted to only allow sharing of objects between threads in immortal and mission memory. Dynamically created scopes during the mission phase are thread private. In SCJ L0 and L1, where no nested missions exist, the hierarchy of nested scopes can grow only in a linear shape thus allowing for simpler illegal reference assignment checks between memory regions.

In this paper we presented how, due to the linear hierarchy of scopes, assignment checks can be performed by simply comparing scope levels. For a restricted nesting level of scopes, these levels can be encoded in the object references. Moreover, on a hardware implementation of a JVM, this check can be done in the memory unit.

Our hardware implementation of the scope checks adds a minimal timing overhead (1 clock cycle) at a negligible hardware cost (4% increase in the MMU). The results show that those bytecodes execute about 10 times faster when checks are performed in hardware than in software. This improvement may not be dominant if cross-scope references between objects are not so frequent or if most of the time is spent in other operations (e.g. floating point).

Since the cost of having the scope checks in hardware is negligible and scope checks are mandatory, this implementation can be used to gain a small performance improvement.

VII. Acknowledgments

We would like to thank Anders Ravn for discussions on scope checks and the memory model of SCJ in general. 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.

References

[1] E. Bruno and G. Bollella, *Real-Time Java programming with Java RTS*, ser. Java (Prentice Hall). Prentice Hall, 2009.

[2] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull, "The real-time specification for Java 1.0.2."

[3] 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*, Java Community Process Std., 2011.

[4] T. Zhao, J. Noble, and J. Vitek, "Scoped types for real-time Java," in *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS'04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 241–251.

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

[6] M. T. Higuera-Toledano and M. A. de Miguel-Cabello, "Dynamic detection of access errors and illegal references in RTSJ," in *Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*, ser. RTAS '02. Washington, DC, USA: IEEE Computer Society, 2002.

[7] M. T. Higuera-toledano, "Hardware-based solution detecting illegal references in real-time java," in *Euromicro Conference on Real-Time Systems*, 2003, pp. 229–237.

[8] A. Plsek, L. Zhao, V. H. Sahin, D. Tang, T. Kalibera, and J. Vitek, "Developing safety critical Java applications with oscj/l0," in *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, ser. JTRES '10. New York, NY, USA: ACM, 2010, pp. 95–101.

[9] D. Tang, A. Plsek, and J. Vitek, "Static checking of safety critical Java annotations," in *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, ser. JTRES '10. New York, NY, USA: ACM, 2010, pp. 148–154.

[10] M. Schoeberl, "Memory management for safety-critical java," in *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, ser. JTRES '11. New York, NY, USA: ACM, 2011, pp. 47–53.

[11] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1999.

[12] M. Schoeberl, "Jop: A java optimized processor for embedded real-time systems," Ph.D. dissertation, Vienna University of Technology, 2005.

[13] J. Cooling, *Software Engineering for Real-Time Systems*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[14] M. Schoeberl, S. Korsholm, T. Kalibera, and A. P. Ravn, "A hardware abstraction layer in Java," *ACM Trans. Embed. Comput. Syst.*, vol. accepted 2009, 2011.