How to Compile Aspects with Real-Time Java

Pengcheng Wu

College of Computer & Information Science
Northeastern University
Boston, Massachusetts 02115 USA
wupc@ccs.neu.edu

ABSTRACT

The Real-Time Specification for Java(RTSJ) uses a special memory model based on scoped memory areas to address the unpredictability of Java's Garbage Collection mechanism, which makes Java unsuitable to write real-time applications. It has been widely believed that Aspect-oriented Programming (AOP) is helpful for implementing distributed computing applications where a lot of crosscutting concerns exist, including real-time concerns. While it is tempting to use AspectJ with RTSJ programs, both of the AspectJ compilers cannot handle complication in the RTSJ setting correctly, since they didn't take into account the special memory model of the RTSJ. This paper reports our exploration in this area and proposes a compilation approach that takes into account of the memory model of the RTSJ.

1. INTRODUCTION

Although Java [1] has been successfully used for developing complex enterprise-level softwares, it has hardly been used for developing real time applications. Real time systems have stringent time constraints that Java is unsuitable to implement. One of the major obstacles is Java's Garbage Collection (GC) mechanism. While the GC provides important software engineering benefits by freeing developers from error-prone manual memory deallocation tasks, its unpredictable object allocation performance (due to the unpredictable behaviors of the garbage collector) makes it impossible to write real time programs.

The Real-Time Specification for Java (RTSJ) [9] was proposed and released to address this problem and it promises to make Java suitable to construct large scale real-time systems. One official reference implementation of the RTSJ has been provided by TimeSys Corp. [10], and several other open source implementations [8, 7] are being developed as well.

One of the most significant features offered by the RTSJ is a new memory management model based on scoped memory areas. A real-time thread can enter a scoped memory area. When it does so, all subsequent object allocation requests (using the **new** operator) until the thread exits from the scoped memory will allocate objects in the scoped memory area, which is not interfered by the GC, and the whole memory area will be freed once all real time threads have exited from it. Thus the time needed to allocate an object in a scoped memory is predictable. Scoped memory areas can be nested. To keep the safety of Java programming, some important object reference rules are set and enforced by RTSJ-compliant JVMs. For example, objects allocated in outer memory scopes must *not* refer to objects allocated in inner scopes to prevent dangling references.

On the other hand, Aspect-oriented Programming (AOP) [3] was proposed as a new programming paradigm to modularize crosscutting concerns and AspectJ [2, 6], as an AOP extension to Java, is the most widely used AOP language. It is widely believed that distributed system applications have many crosscutting concerns and thus are ideal working platforms for AOP techniques. One of the common concerns in distributed system applications is to implement real time requirements. So it would be tempting to use AspectJ on RTSJ systems to see how it could improve implementations of distributed applications.

However, current compilation approaches used by two major AspectJ compilers (one is Eclipse AspectJ team's AspectJ compiler [6] and the other one is the AspectBench Compiler for AspectJ or abc [5].) fail to work in the RTSJ settings, because neither of them took into account the RTSJ's special memory management schema.

This position paper reports our recent experience of using AspectJ in RTSJ programs and our exploration why the current compilation approaches fail to work with them. Based on the findings, we propose the correct compilation strategy with the RTSJ's memory management schema taken into account.

The rest of the paper is organized as follows: Section 2 provides an overview of the RTSJ's memory management model and show how the current AspectJ's compilation approaches fail in this setting; Section 3 proposes a new compilation approach to address those problems; Section 4 discusses future work.

2. ASPECTJ COMPILATION AND RTSJ MEMORY MODEL

2.1 RTSJ's Memory Model

To address the unpredicatability of Java's GC mechanism, the RTSJ extends the Java memory model by providing memory areas other than the heap. Memory areas are divided into three categories, i.e., `ImmortalMemory`, `HeapMemory` and `ScopedMemory` as shown in Figure 1.

`ImmortalMemory` is a singleton memory area and objects allocated in it have the same lifetime of the JVM, i.e., they are never reclaimed and the GC will never interfere with them. Objects allocated in `HeapMemory` are just like regular Java objects that are subject to GC's reclaim.

A `ScopedMemory` area provides guarantees on object allocation time. A real-time thread can enter a scoped memory and when it does so, all subsequent object allocation requests using the **new** operator until the thread exits from it will allocate objects in the scoped memory area. Objects allocated in a scoped memory area are not reclaimed by the GC, instead, the whole memory area will be freed once all real time threads have exited from it. Thus the time needed to allocate an object in a scoped memory is predictable. `LTMemory` and `VTMemory` provide linear time and a variable amount of time allocation respectively.

Scoped memory areas can be nested. Each real-time thread is associated with a scope stack that defines its allocation context and the history of the scoped memory areas it has entered [4]. The RTSJ also provides APIs for programmers to explicitly specify in which memory area an object should be allocated (not just in the most recently entered scoped area). Due to the special characteristics of the RTSJ's memory model, to keep the safety of Java programming, some important object reference rules are set and enforced by RTSJ-compliant JVMs. One of the most important rules is that objects allocated in outer memory scopes must *not* refer to objects allocated in inner scopes to avoid dangling references. At run time, if such kind of references are ever detected by RTSJ-compliant JVMs, an `IllegalAssignmentException` will be thrown and the whole execution will be stopped. It is programmer's responsibility to make sure that object references obey those rules.

To give readers some intuitions, Listing 1 is a short RTSJ program for an aircraft detection system as presented in [11] . Class `App` implements a real time thread. It creates a scoped memory (line 9) and runs a task (implemented by class `Runner`) in the context of that memory area. The `Runner` then creates another memory area (line 16) and allocates a `Detector` object in the area referred by `mem`(line 17). Then the thread enters a loop in which the detector continuously receives position frames from aircrafts and stores those frames so that it can determine, for example, whether two aircrafts are too close each other. Note that the `run` method (line 24) of class `Detector` is called in the dynamic extent of the execution of method `cdmem.enter(...)`(line 19) and thus all of the **new** requests associated with that `run` method will allocate objects in the memory area referred by `cdmem`.

Listing 1: A RTSJ program

```
   class App extends RealtimeThread {
 2   public static void main(String[] args) throws Exception {
       MemoryArea mem = ImmortalMemory.instance();
 4     App app = (App) mem.newInstance(App.class);
       app.start();
 6   }

 8   public void run() {
       ScopedMemory mem = new LTMemory( ... );
10     mem.enter(new Runner());
     }
12 }

14 class Runner implements Runnable {
     public void run() {
16     LTMemory cdmem = new LTMemory( ... );
       Detector cd = new Detector( ... );
18     while(true)
         cdmem.enter(cd);
20   }
   }
22
   class Detector implements Runnable {
24   public void run() {
       Frame frame = receiveFrame();
26     //get a frame and stores it into a table
       ...
28   }
   }
```

2.2 AspectJ's Compilation Approaches Break RTSJ Memory Model

We want to deploy aspects to RTSJ programs to improve implementations of crosscutting concerns. However, the AspectJ 's compilation approaches (both the official AspectJ compiler and the AspectBench Compiler for AspectJ) do not take into account the RTSJ's special memory model and object reference rules, so the compiled code fail to run on RTSJ compliant JVMs. The following subsections present the cases where the compiled code may fail (they may not be all the cases, instead, they are just the cases we have explored).

2.2.1 Instance-based Aspect Instantiation

In the AspectJ language, aspect instantiation is always implicit. Although programmers can specify how aspect should be instantiated, they have no control when the instantiation should happen, neither can they explicitly instantiate aspects. When a programmer defines an aspect, she can specify how the aspect should be instantiated by using **perthis** , **pertarget**, **percflow** keywords or just without specifying anything, which indicates there will be only a singleton instance of the aspect during the program execution. We call **perthis** and **pertarget** *instance-based aspect instantiation*, because for each **this** (and **target** respectively) object of the corresponding join point (as specified by a pointcut designator(PCD)), there is a separate instance of that aspect associated with the object and the advice will be executed on the particular aspect instance corresponding to the **this** (or **target**) object of a join point. Interested readers are referred to the AspectJ language manual [6] for the details.

For an instance-based aspect, the AspectJ compilers gener-

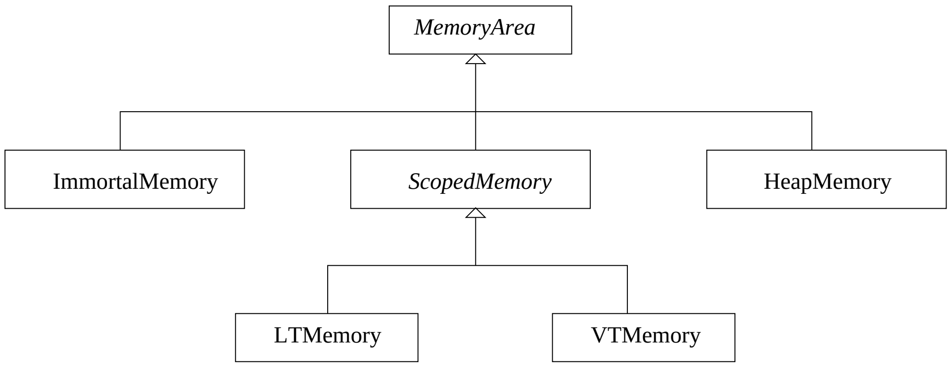

Figure 1: The RTSJ Memory Areas

ate code such that each of those object instances will maintain a reference to its corresponding aspect instance and thus the aspect instance looking up has little runtime overhead. And the aspect instantiation is a “lazy” procedure in that the instantiation (and the reference assignment) only occur when an join point matched with the PCD actually is reached at runtime, instead of whenever an object of such types is created. This approach avoids unnecessary aspect instantiations if no join point matched with the PCD is reached at runtime.

However, in the settings of the RTSJ, the scoped memory area in which an object instance is created is not necessarily the same memory area in which the first join point matched with the PCD occurs. And in such a scenario, the aforementioned object reference rule of the RTSJ may be violated. For example, we want to deploy an aspect as defined in Listing 2 to the RTSJ base program as defined in Listing 1 so that the detector won't do busy polling about the positions of the aircrafts. Instead, it asks those positions periodicly, say every 2 seconds. Aspect `PeriodicalPoll` has to be declared as **perthis**, since there may be many `Detector` objects in the system and each detector has to maintain its own state about when the last polling was.

As expected, an `IllegalAssignmentException` is thrown at runtime (running on the RTSJ official JVM [10] and the code generated by both of the AspectJ compilers show the same behavior). The reason is that the `Detector` instance in this example is allocated (line 17 of Listing 1) in the scoped memory area referred by `mem`, while the `run` method on this detector is first executed in the context of the scoped memory area referred by `cdmem`, where the `PeriodicalPoll` instance corresponding to the `Detector` instance is created and associated with it, and memory area `mem` is an outer scope of memory area `cdmem`. Thus the object reference rule has been violated.

While it is reasonable for a programmer to obey the object reference rules in her own code (e.g, code to implement classes or advice), she has *no* way to fix the problem reported here, since aspect instantiation is implicit and beyond the control of her. It is not just a bug, instead, it is a systematic issue, since similar problems occur on other aspect constructs as well, as presented later. A different compilation approach must be proposed to take into account the RTSJ's memory model.

Listing 2: An aspect applied on the program

```
//Make a detector do periodic polling, instead of busy
//polling.
aspect PeriodicPoll perthis(p()) {
  Time lastTimePolled;
  pointcut p(): execution(* Detector.run(..));

  around(): p() {
    if(it has not yet been 2 seconds since last polling)
       getCurrentThread().yield(); //don't do polling
    else {
      //update the time
      lastTimePolled = System.getCurrentTime();
      proceed(); //do polling
    }
  }
}
```

2.2.2 CFLOW-based Aspect Instantiation

If we change the aspect declaration in Listing 2 to be a **percflow** aspect, the same exception will be thrown when the program is run on the RTSJ JVM. When an aspect is declared as **percflow**, each time the execution enters the dynamic extent of a join point matched with the PCD on which the **percflow** is defined, there is an instance of the aspect created and associated with that extent and the aspect program always operates on the innermost aspect instance. Again, interested readers are referred to the AspectJ language manual [6] for the details.

For a **percflow** aspect, the AspectJ compilers will generate code such that there is a global stack to simulate the execution's entering and leaving the extents of join points, and to store the aspect instances in the stack. The global stack is created in the class loading time, and thus is allocated in the heap memory, while the **percflow** aspect instances may be allocated in scoped memory areas, depending on the current thread's memory context. So the object reference rule of RTSJ may be violated.

2.2.3 CFLOW Pointcut with Bindings

When an aspect has a **cflow** pointcut with bindings, the program may also throw out an `IllegalAssignmentException` when running on the RTSJ JVM. The AspectJ compilers generate code using a similar stack based approach as for **percflow** aspects. And the reason for the exception is also similar.

2.2.4 *Singleton Aspect and Reflective Access to thisJoinPoint*

Singleton aspect instantiation and the reflective access to **thisJoinPoint** are other two cases where the code generated by the AspectJ compilers will create instances that have interactions with the base program. By analyzing the compilation approaches, we expect those two cases won't violate the rules of the RTSJ memory model. And this expectation has been consistent with our experiences of running AspectJ programs on the RTSJ JVM.

3. PROPOSED COMPILATION APPROACH

We propose a different compilation approach with the RTSJ's special memory model and object reference rule taken into account. We do a case by case explanation.

3.1 Instance-based Aspect Instantiation

There are several options to address the instance-based aspect instantiation problem. Let's list and discuss them here.

- Always allocate instance-based aspect instances in the heap memory. The `IllegalAssignmentException` problem will go away with this approach, since objects allocated in scoped memory areas may have references to heap objects. However, this approach is contradictory to one of the original goals of the RTSJ, which is to remove the unpredicatabilities of Java's GC system. Allocate an aspect instance in the heap memory may trigger the GC thread and make the thread be suspended infinitely. Worse, the RTSJ supports a special yet very useful thread kind, `NoHeapRealtimeThread`, which disallows any access to heap objects. So the heap-allocated aspect instance approach cannot work with `NoHeapRealtimeThreads`.
- Allocate instance-based aspect instances in the immortal Memory. The `IllegalAssignmentException` problem will also go away with this approach, since objects allocated in `ImmortalMemory` have the lefetime as the JVM and objects allocated in scoped memory areas may have references to them. But we view `ImmortalMemory` precious resources (because the memory cannot be reclaimed, even when an aspect instance is no longer reachable), so this approach should at least be discouraged so that `ImmortalMemory` can be saved for necessary cases.
- Allocate instance-based aspect instances in the same memory area as the host objects. This approach will make the `IllegalAssignmentException` problem go away, while avoids all of the problems of the previous two approaches. In addition, this approach is feasible, since the RTSJ supports APIs to let the application allocate objects in any accessible memory area.

After analyzing all the possibilities, we propose to use option 3, i.e., allocate instance-based aspect instances in the same memory area as the host objects.

3.2 CFLOW-based Aspect Instantiation

Cflow-based aspect instantiation is more subtle to handle with than the instance-based aspect instantiation, due to the stack structure of a program execution. A simple approach would be to allocate the cflow-based aspect instances in the `ImmortalMemory` (the heap memory is definitely not an option, as discussed before.), but it is not an optimal solution since we want to save the `ImmortalMemory`.

With that in mind, we propose an approach that exploits the tree structure of scoped memory areas and the fact that there is a special communicating `portal` object associated with each scoped memory area. Each of such a `portal` object will maintain a map from threads to stacks, which are similar to the global stack used in the AspectJ compilers. Each of the stacks stores the aspect instances associated with the dynamic extent of the join point occurring in the current scoped memory area in the corresponding thread. When looking up a **percflow** aspect instance, the system will first look it up in the stack (corresponding to the current executing thread) stored in the `portal` object of the current scoped memory area; if it cannot find one, then it will climb up the scoped memory area tree hierarchy and in each of those scoped memories, look it up in the corresponding stack of the `portal` object until it finds one, or it has reached the root where we can determine there is no such an instance.

3.3 CFLOW Pointcut with Bindings

Our proposed approach for this case would be similar to the approach for the previous case. We will exploit the `portal` object and associate a stack to it and make use of the tree structure of the scoped memory areas.

4. FUTURE WORK

We plan to implement the proposed compilation approach in one of the AspectJ compilers and test the compiler on some real RTSJ benchmarks. In addition, based on the semantics of the RTSJ memory model and the semantics of the AspectJ (we need to give a new one to incorporate the instantiation respects and the cflow-related stuff), we are aiming to give a formal proof that under this new compilation approach, it is guaranteed that there will be no object reference violation due to the instantiations introduced by the compiler and the singleton aspect instantiation or the reflective access to **thisJoinPoint** won't violate those rules either.

5. CONCLUSION

This paper addresses the issues of compiling aspects in the settings of the Real-Time Specification for Java. We have identified and analyzed the cases where the current compilation approaches will fail due to the fact that the special memory model of the RTSJ are not taken into account. Based on the analysis, we propose a new compilation approach to address this problem.

6. ACKNOWLEDGEMENTS

We would like to thank Shriram Krishnamurthi, Karl Lieberherr and Mitchell Wand for their helpful discussion. We are grateful to anonymous reviewers for their valuable comments.

7. REFERENCES

[1] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification*. Addison-Wesley, 2000. Second edition.

[2] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mike Kersten, Jeffrey Palm, and William Griswold. An Overview of AspectJ. In Jorgen Knudsen, editor, *European Conference on Object-Oriented Programming*, pages 327–353, Budapest, 2001. Springer Verlag.

[3] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, pages 220–242. Springer Verlag, 1997.

[4] F. Pizlo, J. M. Fox, D. Holmes, and J. Vitek. Real-time java scoped memory: Design patterns and semantics. In *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2004.

[5] Programming Tools Group at Oxford University and the Sable Research Group at McGill University. The AspectBench Compiler for AspectJ. http://abc.comlab.ox.ac.uk/.

[6] AspectJ Team. AspectJ home page. http://www.eclipse.org/aspectj. Continuously updated.

[7] The jRate Team. The jRate Project. http://jrate.sourceforge.net/index.html.

[8] The Ovm Project Team. The Ovm Project. http://www.ovmj.org/.

[9] The Real-Time for Java Expert Group. The Real-Time Specification for Java. https://rtsj.dev.java.net/.

[10] TimeSys Corp. Reference Implementation for RTSJ. http://www.timesys.com.

[11] Tian Zhao, James Noble, and Jan Vitek. Scoped types for real-time java. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, 2004.