Predictable Memory Utilization in the Ravenscar-Java Profile

Jagun Kwon, Andy Wellings, and Steve King
Department of Computer Science
University of York, UK
{jagun, andy, king}@cs.york.ac.uk

Abstract

In this paper, we present the Ravenscar-Java profile from the perspective of memory utilization. This restricted programming model removes language features with high overheads and complex semantics, on which it is hard to perform various static analyses. Several classes in the RTSJ are refined, and a few new classes are added, which all result in a compact, yet powerful and predictable computational model for the development of software-intensive high integrity real-time systems. We provide rationales behind the decisions we have made on the use of memory areas and other language features that can have an effect on the predictability of memory utilization. After that, some analysis approaches are discussed in terms of how they can be developed and beneficially used.

1. Introduction

Memory is an important resource in any computer-based systems; improper use of it can cause system failure or malicious errors that can be hard to detect and correct. This is more so in high integrity systems or other embedded applications because, as a result of the complex computational models of today's programming languages, it has become more challenging to verify that a multithreaded program correctly utilizes its allocated memory space.

This paper reviews the Ravenscar-Java profile [21] in terms of how it facilitates predictable memory utilization, and gives rationales behind the decisions we have made. We then demonstrate some ways in which the profile eases various program analyses (e.g. memory consumption analysis) on Java programs. This will undoubtedly help produce safer Java software that should satisfy the demands of high integrity applications.

1.1. High Integrity Systems and Java

Computers are increasingly used in high integrity systems, where failure can cause loss of life, environmental harm, or significant financial penalties. Examples of such systems include space shuttles, nuclear power plants, automatic fund transfers and medical instruments. They typically have hard real-time requirements, implying any missed deadlines will have a direct impact on the safety of the systems. Within such systems, there has been a growing trend of using software, because it provides improved functionality, increased flexibility in design and implementation, reduced production cost, and enhanced management of complexity in application areas [23, 24, 27, 8].

Java, equipped with an automatic garbage collection mechanism, has proved to be an appropriate vehicle for a diverse range of applications thanks to its relatively simple linguistic semantics, the adoption of well-understood approaches to managing software complexity, and support for concurrency. Initially designed with embedded systems in mind, Java's main goal was to provide engineers with a reliable and cost-effective platform-independent environment.

However, despite all these valuable features, Java has been criticised for its unpredictable performance as well as some security concerns [2, 4, 1]. The automatic garbage collection and dynamic class loading mechanisms are often considered problematic, especially under time or performance-critical situations. Moreover, a number of security bugs in the Java virtual machine have been discovered since its first appearance, especially in the bytecode verifiers and Just-in-Time (JIT) compilers [13, 2]. These fears make the full Java language and its associated technology unsuitable for high integrity systems [22].

1.2. RTSJ and Ravenscar-Java Profile

In recent years, there has been a major international activity, initiated by Sun, to address the limitations of Java for real-time and embedded systems. The *Real-Time Specification for Java* (RTSJ) [7] attempts to minimise any modification to the original Java semantics and yet to define many additional classes that must be implemented in a supporting virtual machine. The goal is to provide a predictable and expressive real-time environment. This, however, ironically leads to a language and run-time system that are complex to implement and have high

overheads at run-time [1]. Software produced in this framework is also difficult to analyse with all the complex features, such as the asynchronous transfer of control (ATC), dynamic class loading, and scoped memory areas.

Following the philosophy of the Ravenscar profile for Ada [9], we have proposed a high integrity profile for real-time Java (called Ravenscar-Java [21]) along the lines of the set of software guidelines produced by the U.S. Nuclear Regulatory Commission (NRC) [15]. This restricted programming model (or a subset of Java and RTSJ) offers a more reliable and predictable programming environment by preventing or restricting the use of language features with high overheads and complex semantics. Hence, programs become more analysable in terms of timing and safety and, ultimately, become more dependable. The profile is intended for use within single processor systems.

The computational model of the profile defines two execution phases, i.e. *initialisation* and *mission phase*. In the initialisation phase of an application, all necessary threads and memory objects are created by a special thread *Initializer*, whereas in the mission phase the application is executed and multithreading is allowed based on the imposed scheduling policy. There are several new classes that will enable safer construction of Java programs (for example, *Initializer*, *PeriodicThread*, and *SporadicEventHandler*), and the use of some existing classes in Java and RTSJ is restricted or simplified due to their problematic features in static analysis. For instance, the use of any class loader is not permitted in the mission phase, and the size of a scoped memory area, once set, cannot be altered. Further restrictions include the following (see [21] for details).

- No nested scoped memory areas are allowed,
- Priority Ceiling Emulation must be implemented and used for all synchronized methods/blocks,
- Processing groups, overrun and deadline-miss handlers are not supported,
- Asynchronous Transfer of Control (ATC) mechanism is not allowed,
- Object queues are not allowed (i.e. no *wait*, *notify*, and *notifyAll* operations),
- *continue* and *break* statements in loops are not permitted, and
- Expressions with possible side effects are not allowed.

1.3. Outline of the Paper

This paper is structured as follows: the next section describes the general computational model of the profile, while Section 3 deals with memory utilization issues in detail, giving rationales for the rules and guidelines proposed in the profile. Section 4 shows how various analysis techniques can be applied on Ravenscar-Java programs, followed by some related works in Section 5. Conclusions as well as future work are given in the final section.

2. Computational Model of the Profile

The key aim of the Ravenscar-Java profile is to develop a concurrent Java programming model that supports predictable and reliable execution of application programs, thus benefiting the construction of modern high integrity software. Particularly, we follow the philosophy of the Ravenscar profile [9] and emphasise the *reliability* attribute of the NRC guidelines[15]. This means that some language features with high overheads and complex semantics are removed for the sake of reliability, and programs are statically analysable in terms of functionality and timeliness before execution. Similarly, the Java virtual machine is also restricted to ensure predictability and efficiency. For example, a Ravenscar-Java VM (RVM) does not support garbage collection.

As in the RTSJ, the Ravenscar-Java profile allows concurrent execution of schedulable objects (threads and event handlers) based on pre-emptive priority-based scheduling. Schedulable objects have to be either periodic or sporadic with minimum inter-arrival times, and the priority ceiling protocol is required to be implemented in the runtime system. This profile facilitates the use of off-line schedulability analysis, which is associated with fixed priority scheduling (e.g. deadline monotonic or rate monotonic analysis [3, 25]).

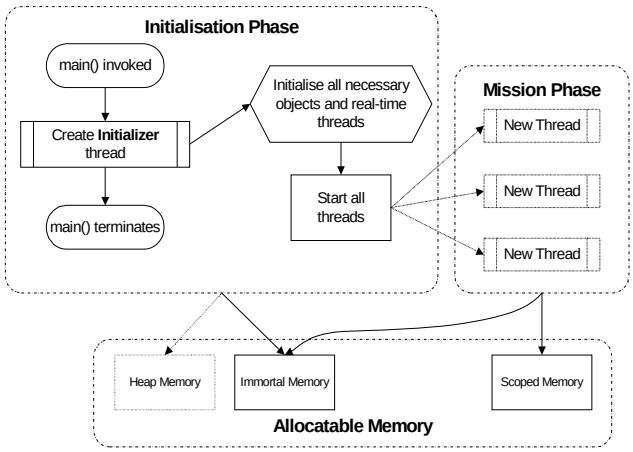

Figure 1. Two execution phases

[1] Sources of run-time overhead in RTSJ include interactions between the garbage collector and real-time threads, assignment rule/single-parent rule checks for objects in different memory areas, and asynchronous operations.

We assume two execution phases as suggested in [29], i.e. *initialisation* and *mission* phase, shown above in Figure 1. In the initialisation phase of an application (i.e. the *main*() method and one *RealtimeThread*), all non-time-critical activities and initialisations that are required before the mission phase are carried out. This includes initialisation of all real-time threads, memory objects, event handlers, events, and scheduling parameters[2]. In the mission phase, the application is executed and multithreading is allowed based on the imposed scheduling policy.

3. Memory Utilization in the Ravenscar-Java Profile

One of the main goals of the profile is to ensure that the utilization of memory space will always be predictable and statically bounded. To achieve this goal, a number of restrictions have been developed and placed on the use of the problematic features and classes of the language and RTSJ. In addition to this, we define several new classes that will smooth the way of coding high integrity real-time Java programs. Below we provide rationales and clarification to the restrictions and classes concerning memory utilization.

3.1. Two Execution Phases

By having two execution phases (i.e. initialisation and mission phase), we can be certain that application threads will not be interfered with by system-level activities, which include setting up memory areas and scheduling parameters, loading classes, and creating new threads. These activities not only hinder an accurate timing analysis of the application, but also make it difficult to perform a precise memory utilization analysis. This is because, in an ordinary Java program, classes may be loaded while the program is running and new memory area objects can be created, thus requesting more memory space from the underlying system at run-time. If such events are blended with complex program logic, it will become infeasible to obtain even the worst-case memory consumption bounds. Therefore, in the profile we define two execution phases with two different roles to play: the initialisation phase must create all required memory area objects, create all schedulable objects, and load all necessary classes, whereas the mission phase simply executes the schedulable objects according to the imposed scheduling policy (i.e. fixed priority scheduling in the profile).

[2] This includes loading all the classes needed in the application. In a JIT (Just-In-Time) compilation environment, all loaded classes will be compiled.

The newly defined *Initializer* class encapsulates the basic procedures of the initialisation phase of an application, which extends the *RealtimeThread* class of RTSJ, and runs with the highest priority, so that the *Initializer* thread will not be interrupted by any other thread. A typical application will be created by extending the *Initializer* class, as illustrated below.

```
import ravenscar.*;
public class MyApplication extends Initializer
{
  public void run()
  {
    // Logic for initialization: i.e.
    // Create memory areas (LTMemory areas)
    // Create periodic threads or sporadic event-
    // handlers
    // All objects required by application should
    // be created here or in the constructors of
    // objects created.
    // Start all threads.
  }

  public static void main (String [] args)
  {
    MyApplication myApp = new MyApplication();
    myApp.start();
  }
}
```

The mission phase begins as soon as the highest priority thread (i.e. *Initializer* thread) terminates. From this moment, the application threads will not be allowed to alter the properties of the memory areas, scheduling parameters, and so on. Threads may only utilise immortal and linear-time scoped memory areas in this phase, unless their logics require access to physical or raw memory areas[3].

3.2. Use of Memory Areas

Although real-time garbage collection has made significant advances in recent years [16, 32], there is still a reluctance to rely on it for high integrity systems. Instead, the profile only allows the immortal memory area and linear-time scoped memory areas. Any heap memory usage needed to start the main Java program is not collected and is considered to be part of immortal memory.

3.2.1. Immortal memory area. By definition, objects in an immortal memory area cannot be freed or moved, and all schedulable objects in an application share the same memory area [7][4]. Hence, in an attempt to prevent

[3] In this paper, we do not attempt to restrict the use of physical or raw memory other than what is implied by our restrictions on scoped memory areas. However, a potential implementation of a RVM might apply restrictions for security reasons.

[4] In practice, this may be too strong a definition. What is required is that the allocating and freeing of immortal memory is not subject to interference by the garbage collector [11]

memory exhaustion or corruption, objects (including memory area objects) that are needed for the lifetime of the application should be allocated in the area only in the initialisation phase.

3.2.2. Linear-time scoped memory areas. All memory area objects must be created during the initialisation phase (thus, in the immortal memory area), and schedulable objects during the mission phase should make use of their allocated linear-time scoped memory areas (*LTMemory* areas). Scoped memory areas will not be shared by more than one schedulable object as it is difficult to cost-effectively validate if a given scoped memory area is exploited correctly when different schedulable objects share it. In other words, one schedulable object must have only one active *LTMemory* area dedicated to it, and should use the immortal memory area when sharing data with another schedulable object. With this rule enforced, additional overheads of dynamic memory access checking can be potentially eliminated, and the virtual machine design and implementation will be significantly simplified.

The size of all memory objects must be static and not be extended in the course of the program for the reasons mentioned above, i.e. the cost of extending the size at runtime[5]. Any other memory area objects defined in the RTSJ are disallowed, and the following simplified classes remain in the profile.

```
package ravenscar;

public abstract class MemoryArea
{
  protected MemoryArea(long sizeInBytes);
  protected MemoryArea(
             javax.realtime.SizeEstimator size);

  public void enter(java.lang.Runnable logic);
         // throws ScopedCycleException

  public static MemoryArea getMemoryArea(
                 java.lang.Object object);

  public long memoryConsumed();
  public long memoryRemaining();
  public java.lang.Object newArray(
        java.lang.Class type, int number)
         throws IllegalAccessException,
                InstantiationException;
         // throws OutOfMemoryError
  public java.lang.Object
               newInstance(java.lang.Class type)
         throws IllegalAccessException,
                InstantiationException;
         // throws OutOfMemoryError
  public java.lang.Object newInstance(
               java.lang.reflect.Constructor c,
               java.lang.Object[] args)
         throws IllegalAccessException,
                InstantiationException;
         // throws OutOfMemoryError;
  public long size();
}

public final class ImmortalMemory extends
                                      MemoryArea
{
  public static ImmortalMemory instance();
}

public abstract class ScopedMemory extends
                                      MemoryArea
{
  public ScopedMemory(long size);
  public ScopedMemory(SizeEstimator size);

  public void enter();
  public int getReferenceCount();
}

public class LTMemory extends ScopedMemory
{
  public LTMemory(long size);
  public LTMemory(SizeEstimator size);
}
```

Figure 4. Simplified memory area classes

To aid in the production of an efficient virtual machine and to simplify timing and memory usage analyses, access to *LTMemory* areas must not be nested and *LTMemory* areas must not be shared between *Schedulable* objects as mentioned earlier. Otherwise, the virtual machine will have to perform heavy assignment checks and enforce the single parent rule at run-time [7], which will cause a significant amount of overheads. This sort of runtime check is not desirable, and may have ambiguous time or memory requirements. With this restriction,

- any assignment to memory in an *LTmemory* area can always be allowed without checks; this is because the referenced object must always have the same scope or be in immortal memory, and
- any assignment to memory in immortal (or heap) must be checked to ensure the referenced object does not reside in an *LTmemory* area (as this could potentially result in a dangling reference); the goal is to perform static analysis to validate this rule before program execution (see section 4.3).

3.3. Other Restrictions

There are further restrictions along with those mentioned above, which include the following.

3.3.1. Dynamic class loading in the mission phase. Java classes and methods that can be used to load additional classes at run-time cannot be exploited in the mission phase since dynamic class loading is one of the main sources of delay and unpredictable use of memory. In particular, the classes listed below must not be used in the mission phase of any application.

[5] Having statically analysed the program, memory shortage should never occur. Therefore, some of the (unchecked) exceptions are not necessary (thus are commented out as shown below).

- `java.lang.ClassLoader`
- `java.lang.Class` (`forName()` methods)
- `java.net.URL.ClassLoader`
- `java.security.SecureClassLoader`

3.3.2. Simple finalizers. Object finalizers are invoked when the virtual machine detects that there are no more references to the objects. In the context of the scoped memory area, this process should occur when a memory scope is escaped (i.e. the *reference count* becomes zero), and all the finalizers of the objects in the scope should be invoked. Finalizers should not be associated with objects created in immortal memory because Ravenscar-Java applications are assumed not to terminate[6].

The overheads of finalizers of *LTmemory* objects must be taken into account when performing schedulability analysis because the virtual machine can take some time to free up used memory areas. Further, finalizers should not attempt to acquire object locks that other threads or event handlers can use; there could be some possibilities of execution delay and even a deadlock when a thread holds a lock and attempts to enter a scoped memory area in which the finalizer of an object is attempting to acquire the same lock [5]. The thread will have to wait until all the finalizers of objects in the memory area run to completion while one of the finalizers is waiting for the lock to be available. On the whole, finalizers should be as compact as possible and must not block.

3.3.3. Method recursion. Recursive method calls (including mutually recursive calls, and loops in some cases) can dramatically consume available memory space at runtime, and an erroneous termination condition can cause unbounded recursion. Thus, programmers must avoid method recursion. However, this rule may be relaxed if the memory consumption for each method and termination conditions can be formally verified and bounded.

4. Analysis of Ravenscar-Java Programs

The inherent complexity in the verification of non-trivial software means that unsafe programs could be produced and used under critical situations. This is more so as today's programming models become more complex. Our Ravenscar-Java profile [21] has been developed with such concerns in mind, so that programs may become easily analysable, and the run-time platform will also be simpler to implement.

By statically analysing a program, we obtain a high degree of assurance that a program will behave according to its functional (and temporal) specification, and not exhibit any erroneous actions throughout its lifetime. Erroneous actions include data races, deadlocks, and memory overflows. However, in the context of real-time Java and the Ravenscar-Java profile, we also need to ensure that the rules defined in the profile and RTSJ are observed, for example:

- scoped memory areas are not nested,
- classes are not dynamically loaded in the mission phase,
- methods are not recursively invoked,
- periodic and sporadic threads are programmed using the given classes (i.e. *PeriodicThread* and *SporadicEventHandler*), and
- object queues are not used.

These rules are checked when programs are compiled and tested for conformance to the profile. This conformance test alone will remove many possible errors in the program. For example, deadlocks, and side effects in expressions can be prevented.

A conformance test can only be performed once the main application class has been defined. This allows the schedulable objects and all objects they access to be identified. The control and data flow graphs can then be generated. If two or more schedulable objects, for example, attempt to enter the same *LTmemory* area, an error can be flagged[7].

However, along the lines of the conformance test, a number of other useful memory analyses can be conducted on Ravenscar-Java programs, and we show some of the ways briefly in the following subsections, although there may be many other possibilities.

4.1. Verification of the Java Memory Model's Effect

As reported in [28] and [30], the Java memory model (JMM) in [14] is a weaker model of execution than ones supporting *sequential consistency*. It allows more behaviours than simple interleaving of the operations of the individual threads. Therefore, verification tools that simply examine Java source code or even bytecode can be prone to produce false results [30]. Because the semantics of the JMM can lead to different implementations, some virtual machines may support sequential consistency, while others may not for performance reasons. This does not match the Java's *write once, run everywhere* [8] philosophy.

[6] Any extension to Ravenscar-Java to allow termination should also address the issues of daemon schedulable objects, which currently are absent from the RTSJ definition.

[7] This may be overly restrictive as the profile requires that no two schedulable objects attempt to enter the same memory area at the same time.

[8] Programs may still run everywhere, but possibly with different or unsafe behaviours.

Roychoudhury and Mitra suggest that there are three approaches to tackle this problem [30], i.e.

1. Develop restricted fragments of Java programs for which the JMM guarantees sequential consistency,
2. Change the JMM altogether, or
3. Develop an executable formal description of the JMM and incorporate it into program verification.

They opt for the third approach because the first one can suffer from performance overheads when there are unnecessary synchronizations for every shared object, and foreign libraries may not follow such restricted fragments or rules. The second approach is seriously considered by the Java Community Process (JCP) 133 [20], but it is unrealistic to expect that all Java virtual machines will support one of the two proposed models soon.

However, we can still settle on the first approach, developing restricted fragments of Java programs for which the JMM guarantees sequential consistency, so that programs in the fragments will not have any unsafe effects of the problematic Java memory model. This approach may require a means to statically analyse Java bytecode to locate only necessary synchronizations, and libraries will still be considered because it can operate at the bytecode level.

The underlying assumption of this possible approach is that any reads and writes on a shared object in a method must be enclosed within the same synchronized block (or method) in order not to have any data races[9]. In other words, any syntactical gap between a read and write that are not covered by a single synchronized block will cause possible data races in a multithreaded environment because either a read or write action can be lost. This is true even when a shared object is indirectly read and updated using a local object because, for example, an interleaving of another thread that may update the shared object can occur in between the indirect read and a (synchronized) write in the method, resulting in a lost write. Thus, any indirect reads and writes should also be treated in a similar manner to direct ones on a shared object. The figure below shows an example case (see the *incBy1* method).

Another similar case is that even when both a read and write are synchronized, there still can be data races if the two blocks are guarded by two different synchronized blocks and can be interleaved by other threads in between (see the *incBy2* method in the figure).

An efficient algorithm can be developed that is capable of analysing all such conditions above, thus detecting problematic data races by tracing all shared objects and checking whether they are properly guarded by synchronized blocks or methods. The point-to and escape analysis [10, 31] can be used to trace escaping and possibly shared objects, as well as improving overall performance by allocating non-escaping objects in the stack of a method.

The Ravenscar-Java profile should simplify this task, since the algorithm will not have to deal with some of the language's complex features, such as the object queue.

Shared variable: **int** *Svar*

```
void incBy1(int val)
{
  int tmp1;
  int tmp2;

  tmp1 = Svar;

 //indirect read
  tmp2 = tmp1 + val;

 //synchronized write
  synchronized(this){
    Svar = tmp2;
  }
}
```

```
void incBy2(int val)
{
  int tmp;

 //synchronized read
  synchronized(this){
    tmp = Svar;
  }

 //synchronized write
  synchronized(this){
    Svar = tmp + val;
  }
}
```

Figure 4.1. Two methods illustrating possible data races

4.2. Memory Consumption Analysis

Overflow or shortage of memory space at run-time can be devastating in high integrity systems, but at the same time, oversupply of it will be costly. Considering the new memory areas introduced in the RTSJ, we need a different means of estimating the worst-case memory space that a program requires at run-time, so that only the required amount of memory for each area will be allocated. For this purpose the RTSJ defines the *SizeEstimator* class. However, the *getEstimate*() method of the class does not return the actual amount of memory that an object of a class and its methods dynamically use, but simply the total size of the class's fields. In this sense, the class is not readily usable in estimating the required memory size for an RTSJ application.

The Ravenscar-Java profile places some restrictions on the use of RTSJ's memory areas; for example, access to linear-time memory (*LTMemory*) areas must not be nested and such memory areas cannot be shared between *Schedulable* objects [21]. These restrictions should greatly ease the development of an algorithm that will inspect each thread's logic to discover all classes it instantiates. After that, by making use of control and data flow information extracted from the code (such as loop bounds), the algorithm will be able to calculate how many instances of each class are created by a thread. This information can then be used to produce a tight upper bound of the amount of memory that a thread utilizes at

[9] Essentially, data races are the most obvious outcome that the Java memory model could have on any multithreaded Java programs.

run-time by applying *reserve*() and *getEstimate*() methods of the *SizeEstimator* class at the target platform before the system's mission phase.

4.3. Dynamic Memory Access Check Analysis

This analysis is concerned with eliminating unpredictable runtime overheads caused by dynamic checks by the virtual machine. Because of the characteristics of the memory areas defined in the RTSJ, many assignment expressions are subject to a dynamic check, which will determine the legality of such assignment expressions. In other words, an object in a long-term memory area cannot not reference another object in a short-term memory area. However, dynamic memory access checks can be prevented by means of statically performing the point-to and escape analysis [10, 31].

Objects that escape from their original memory areas can first be identified using an escape analysis technique, followed by a simple means to resolve which direction the object is escaping in the memory area stack of a particular thread. If the escaping object is referenced by another object in a longer-term memory area, then the assignment at run-time will fail and an exception will be raised. A Ravenscar-compliant virtual machine will not need such dynamic checks.

5. Related Works

There have been a few subsets or profiles for Java suggested in the literature[10]. None of them, however, is as complete or analytical as the Ravenscar-Java profile described in this paper and in [21].

Bentley [6] defines a sequential subset of Java after assessing the language. The subset consists of 21 rules that are effectively derived from [18], [26] and his assessment. All the rules are categorised into six groups, most of which are concerned with the use of some problematic features of the language itself. However, while this subset will undoubtedly help produce analysable and predictable *sequential* programs, it can be criticised for its restriction on multithreading, one of Java's inherent elements. Without the language-level support for multithreading and all the associated synchronisation mechanisms, Java may not be considered as a great evolution from its predecessors. In addition to this, the subset also fails to address issues on the object-oriented programming model of the language, as well as real-time issues.

Puschner and Wellings [29] suggest a *Ravenscar*-like profile for the Real-Time Specification for Java [7], and it is in fact a predecessor of the work presented in this paper. The profile is primarily focused on leaving out complex features of the RTSJ. However, little attention is paid to Java's sequential language constructs (unlike [6]) and object-orientation features that can be problematic in performing various static analyses. Furthermore, the profile is not consistent with the current version of the RTSJ.

A sub-committee has been formed within the Real-Time Java Working Group of the J Consortium to produce a high integrity profile based on the Real-Time Core Extensions [19]. The profile has not publicly been released yet, but according to Dobbing [12] it will resemble the Ravenscar profile for Ada95 [9]. It consists of four main themes: partitioning, memory management, concurrency, and error recovery. Like the one proposed in [29], this profile is mainly focused on sub-setting the Real-Time Core Extensions [19], but does not address issues on the use of problematic language constructs and object-orientation features of Java.

6. Conclusions and Future Work

In this paper we have presented the Ravenscar-Java profile from the perspective of memory utilization. This restricted programming model removes language features with high overheads and complex semantics, on which it is hard to perform various static analyses. Several classes in the RTSJ are refined, and a few new classes are added, which all result in a compact, yet powerful and predictable computational model for the development of software-intensive high integrity real-time systems.

We have given the rationales behind the decisions we have made on the use of memory areas and other language features that can have an effect on the predictability of memory utilization. Then, some of the analysis approaches are discussed, in terms of how they can be developed and beneficially used. We intend to implement the approaches in our integrated analysis tool [17] in the near future.

We believe that the profile is concise, yet expressive enough to use in high integrity real-time applications, especially with the restrictions placed on memory utilization that will significantly ease the analysis of Ravenscar-Java programs.

7. Acknowledgements

This work has been funded by the U.K. EPSRC under award number GR/M94113.

8. References

[1] W. Amme, N. Dalton, M. Franz, and J. Von Ronne, *SafeTSA: A Type Safe and Referentially Secure Mobile-Code*

[10] In fact, there are subsets of Java defined for other purposes than for use in high integrity systems. They are not considered in this paper.

Representation Based on Static Single Assignment Form, Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2001.

[2] Andrew W. Appel, *Protection against untrusted code: The JIT compiler security hole, and what you can do about it*, http://www-106.ibm.com/developerworks/library/untrusted-code/, as of January 2001.

[3] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, *Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling*, Software Engineering Journal, 8(5), 284-92, 1993.

[4] A. Azevedo, A. Nicolau, and J. Hummel, *Java Annotation-Aware Just-In-Time (AJIT) Compilation System*, ACM SIGPLAN Java Grande Conference, 1999.

[5] W. S. Beebee Jr., *Region-based Memory Management for Real-Time Java*, Master's thesis, Dept. of Electrical Engineering and Computer Science, MIT, 2001.

[6] S. Bentley, *The Utilisation of the Java Language in Safety Critical System Development*, MSc dissertation, Department of Computer Science, University of York, 1999.

[7] G. Bollella, et al, *The Real-Time Specification for Java*, Addison-Wesley, 2000.

[8] J. P. Bowen and M. G. Hinchey, *High Integrity System Specification and Design*, Springer-Verlag London, 1998.

[9] A. Burns, B. Dobbing, and G. Romanski, *The Ravenscar Tasking Profile for High Integrity Real-Time Programs*, In L. Asplund, editor, Proceedings of Ada-Europe 98, LNCS, Vol. 1411, pages 263-275, Berlin Heidelberg, Germany, Springer-Verlag 1998.

[10] J. D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff, *Escape Analysis for Java*, Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 1999.

[11] P. Dibble, Personal communications, October 2002.

[12] B. Dobbing, *The Ravenscar Profile for High Integrity Java Programs?*, ACM Ada Letters, Vol. 21, Issue. 1, March 2001.

[13] Li Gong, *Inside Java™ 2 Platform Security: Architecture, API Design, and Implementation*, Addison-Wesley, 1999.

[14] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, 2nd Edition, Addison Wesley, 2000.

[15] H. Hetcht, M. Hecht, S. Graff, et al, *Review Guidelines for Software Languages for Use in Nuclear Power Plant Systems*, NUREG/CR-6463, U.S. Nuclear Regulatory Commission, 1997, also available at http://fermi.sohar.com/J1030/index.htm, last accessed in January 2002.

[16] R. Henriksson, *Scheduling Garbage Collection in Embedded Systems*, Ph.D thesis, Department of Computer Science, Lund University, 1998.

[17] E. Y-S. Hu, J. Kwon and A. Wellings, *XRTJ: An Extensible Distributed High-Integrity Real-Time Java Environment*, To appear in the Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA), 2003.

[18] A. Hutcheon, B. Jepson, D. Jordan, and I. Wand, *A Study of High Integrity Ada: Language Review*, Technical Report SLS31c/73-1-D, Version 2, York Software Engineering, University of York, July 1992.

[19] J Consortium, *International J Consortium Specification: Real-Time Core Extensions*, Revision 1.0.14, www.j-consortium.org, September 2000.

[20] *JSR 133: Java™ Memory Model and Thread Specification Revision*, http://jcp.org/en/jsr/detail?id=133

[21] J. Kwon, A. Wellings, S. King, *Ravenscar-Java: A High Integrity Profile for Real-Time Java*, Proceedings of the Joint ACM Java Grande - ISCOPE 2002 Conference, Seattle, Washington, 2002.

[22] J. Kwon, A. Wellings, and S. King, *Assessment of the Java Programming Language for Use in High Integrity Systems*, Technical Report YCS 341, Department of Computer Science, University of York, 2002, available at http://www.cs.york.ac.uk/ftpdir/reports/YCS-2002-341.pdf.

[23] N. G. Leveson, *Software Safety: Why, What, and How*, Computing Surveys, Vol. 18, No. 2, ACM, June 1986.

[24] N. G. Leveson, *Software Safety in Embedded Computer Systems*, Communications of the ACM, Vol. 34, No. 2, February 1991.

[25] C. Liu and J. Layland, *Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment*, Journal of ACM, 20(1), 46-61, 1973.

[26] The Motor Industry Software Reliability Association, *Guidelines for the use of the C language in vehicle based software*, The Motor Industry Research Association (MIRA), 1998.

[27] D. L. Parnas, A. J. van Schouwen, and S. P. Kwan, *Evaluation of Safety-Critical Software*, Communications of the ACM, Vol. 33, No. 6, June 1990.

[28] W. Pugh, *Fixing the Java Memory Model*, Proceedings of Java Grande Conference, 1999.

[29] P. Puschner and A. J. Wellings, *A Profile for High Integrity Real-Time Java Programs*, In Proceedings of the 4th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC), 2001.

[30] A. Roychoudhury and T. Mitra, *Specifying Multithreaded Java Semantics for Program Verification*, Proceedings of the International Conference on Software Engineering – ICSE, 2002.

[31] A. Salcianu and M. Rinard, *Pointer and Escape Analysis for Multithreaded Programs*, Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2001.

[32] F. Siebert, *Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages*, aicas GmbH, available at http://www.aicas.com/books.html, 2002.