A Safe Mobile Code Representation and Run-time Architecture for High-Integrity Real-Time Java Programs

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

Abstract

Java is becoming increasingly popular in many application areas due to its rich programming semantics and portability. We believe that high-integrity real-time systems can also greatly benefit by adopting the Java technology, provided the unpredictable overheads and insecurity of the run-time system are conquered. We illustrate in this paper our on-going work on a safe mobile code representation based on SSA form, and a run-time system that will enable us to examine program code in terms of safety, WCET, and schedulability. Some miscellaneous techniques for detecting race conditions and allocating registers are discussed along with examples.

1. Introduction

High integrity systems are often characterised by their rigorous requirements for *Safety*, *Security* and *Timeliness*, and may have disastrous effects to the environment should they fail to perform accordingly. Such systems typically have a high production and maintenance cost due mainly to their use of tailored components including software. Here, upgrading subsystems and software portability can become a serious concern because of the customised nature of such systems. Hence, the use of mobile software and an appropriate run-time architecture may be a valid approach to tackling this problem.

Bearing this point in mind, Puschner and Wellings [12] proposed a profile for High-Integrity Real-Time Java programs, a subset of the Real-Time Specification for Java [5]. It is intended to facilitate the development of efficient and predictable systems.

Having considered the potential benefits of exploiting Java and profiles such as this, one may naively conclude that Java is now ideal and mature for developing high-integrity systems. However, as noted by many researchers the high-level binary (or JBC) significantly suffers from unpredictable overheads and possible security fears [2, 4, 14, 1], which is especially true when the Just-in-Time Compiler is used. This dilemma principally stems from the underlying stack model.

In this paper, we propose a new safe mobile code representation and run-time architecture in an attempt to overcome some of the problems mentioned above, and to encourage the use of Java, particularly the profile of [12] in building high-integrity real-time software. Our approach was initially inspired by SafeTSA [1] on the use of Static Single Assignment (SSA) forms [7] to safely represent mobile code. But, we focus more on temporal predictability and safety of Java programs, and develop some novel techniques for detecting race-conditions, checking data referential integrity and allocating registers.

Sections 2 and 3 briefly depict the architecture and the mobile representation with associated techniques. In section 4, we discuss some real-time issues. The subsequent sections are devoted to showing the implementation status, and related works. Finally, a conclusion is provided. A simple example showing our mobile representation is given at the first author's web page[1].

1.1. Summary of the profile

Burns et al. [6] suggest that a restricted programming model or profile can help produce efficient and predictable systems by removing language features with high overheads or complex semantics. Based on this idea, Puschner and Wellings [12] have defined a *Ravenscar*-like profile for the RTSJ, and the following is a brief summary of each of the key areas.

Threading model. There are two execution phases, i.e. *initialisation* and *mission* phases. In the initialisation phase, *all* necessary threads, event handlers, and memory objects are created in a non time-critical manner. No threads will be allowed to start until the top-priority thread with *main()* method finishes its execution. In the mission phase, threads may not change their own or other thread's priority except when forced by the underlying implementation of the priority ceiling protocol. Sporadic or event-triggered activities are implemented as event handlers, and only one handler is allowed per event.All periodic threads must be an instance of *NoHeapRealtimeThread* class and need to invoke *waitForNextPeriod* method to delay execution until the start of their next periods. Asynchronous Transfer of Control (ATC), overrun and deadline-miss handlers, and

[1] http://www.cs.york.ac.uk/~jagun/RTSS_WIP_Appendix/Traction.htm

delay statements are not supported by the profile; nor is dynamic class loading during the mission phase.

Concurrency. Synchronized methods and blocks are the key mechanism for mutual exclusion to shared resources in Java, and the priority ceiling protocol should be implemented in the run-time system in order to evade deadlocks. For similar reasons, *wait*, *notify*, and *notifyall* are not supported, avoiding any queue management.

Memory management and raw memory access. The heap-based garbage collection mechanism of Java is not supported due to its long-debated unpredictability at run-time. Instead, only immortal memory and linear-time scoped memory are supported as defined in the RTSJ. Immortal memory is used by default to create objects during the initialisation phase, but is not allowed for further object creation afterwards. In addition to this, all other memory objects must only be created in the initialisation phase. The RTSJ classes for raw memory access are also supported, so that device drivers, memory-mapped I/O, and other low-level functions can be programmed.

Time and clocks. All the RTSJ classes for the representation of time and real-time clocks are included while the timer classes are not.

and method descriptions, so that it is easier to verify the *type safety* of code (which are considered to be sufficient for ensuring the minimum nontrivial level of program safety [11]). It is also more compact than low-level binary, meaning improved network utilization if used.

Having received the high-level code of a program, the Safety verifier automatically extends each instruction and operand(s) with pre- and post-conditions, forming a variant of the Hoare Triple [9]. For instance, the '*compute*' instruction will look like the following example.

<<*typematch*(x, 5) $\wedge$ *inscope*(x) $\wedge$ $x^{\#}10 + 5 = x^{\#}10 + 5$>>
// pre-conditions
compute ($x^{\#}11$, $x^{\#}10 + 5$) // instruction & parameters
<< $x^{\#}11 = x^{\#}10 + 5$>> // post-conditions

Explanation: *typematch()* and *inscope()* are internal functions of the Safety verifier; the first one evaluates true only when specified parameters have the same Java type, and the second returns true when parameters are in scope at this point of the program. This instruction simply adds 5 to the current value (of 10th definition) of variable *x*, and stores the sum in a new definition of *x*, thus is equivalent to '*x = x+5;*' in Java.

Example 2.1. Extended *compute* instruction in a variant form of the Hoare Triple

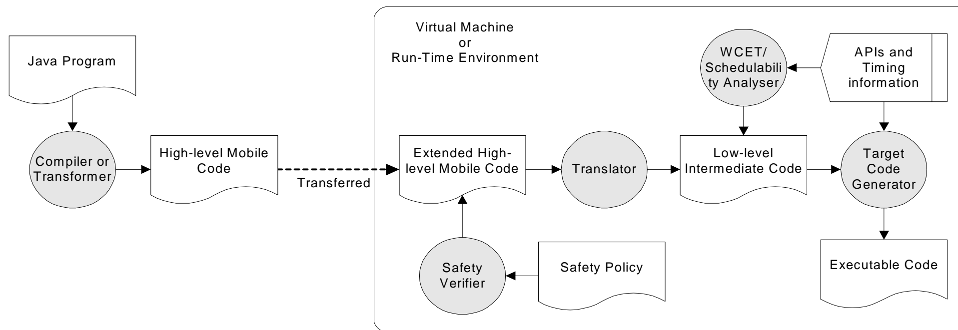

Figure 2.1. Proposed Architecture

2. Proposed architecture and mobile representation

As shown above in Figure 2.1, the proposed architecture consists of several tools that appear as shaded circles. Once a Java program is written, it is transformed into SSA forms and optimised (using SSA techniques for Common Sub-expression Elimination, Constant Propagation, and Dead Code Elimination) by a back-end of our compiler, which produces a high-level mobile representation that is, in fact, transferred to the *code-generating site* or virtual machine. This optimised representation is high-level in that it contains all the source-level semantics of Java programs including class

All such instructions and operands are then verified against their assumed roles expressed in the pre- and post-conditions by the Safety verifier, which at this stage checks for type mismatches, array bound errors, divide-by-zero and so on. Given that the code has passed this test, the verifier now moves on to examine the *data referential integrity*[2] of the code (explained in the next section).

After all the safety checks are performed, the code is now decomposed into our pre-defined low-level instructions that are, in turn, *basic blocks* of native instructions. Provided the *code-executing* system is known it is possible to find tight upper and lower bounds of

[2] *Referential integrity* means that all references in SSA forms are valid, so that there is no illegal use or definition of variables.

execution time of each low-level instruction, so that we can obtain the longest path (or WCET) in our program by incrementally adding all the bounds of instructions on each branch. This task is done by the WCET/Schedulability Analyser.

As a final step, the Target Code Generator takes as input the low-level instructions of the program to generate eventual binary code linked with all necessary kernel libraries.

3. Program safety checks

Blended with SSA form conventions, our instructions are high-level enough to represent the semantics of Java programs, and straightforward for safety checking at the code-generating site. Together with the instructions, there are two structural decorations for classes and methods expressed in a variant form of the Hoare Triple [9]. In order to make it straightforward to reason about pre- and post-conditions of delivered code the Safety verifier implements a few internal functions, and two of them are shown in Example 2.1.

3.1. Data referential integrity check

In SSA forms, every read or *use* of a variable must always refer to the *definition* (or assignment) that immediately precedes it, forming a chain of *definition-and-use* [3]. However, this data-flow chain may become rather complicated if control structures, such as *if* or *while* statements are used. In such cases, associated ø-function(s) need to be applied in order to attain an appropriate definition, and our special expression $ø_i$ ($x^{\#}n \rightarrow x^{\#}l$), which evaluates as $x^{\#}n$ once, then as $x^{\#}l$ afterwards, is used to avoid variable definitions with unnecessarily high version numbers in the beginning of a loop. Consequently, by imposing this rule and verifying the chain of version numbers, one can determine, to some extent, whether a stream of high-level instructions is tampered with, or illegally created.

3.2. Race condition detection

We have found an interesting property of SSA form that may enable us to detect race conditions in concurrent Java programs. Assume there is a tricky method that contains a subtle data race, as shown in Example 3.1 below.

A race condition can occur if more than one thread invoke *add_by* method and are interleaved between the two ill-adapted synchronized blocks. This sort of programming error often goes undetected by conventional checkers such as ESC/Java.

A program with a data race	Converted into SSA form

class Storage_A {
static int resource = 0;
...
void add_by (int by)
{
int tmp = 0;
synchronized (this) {
tmp = resource;
}
...
synchronized (this) {
resource = tmp + by;
}
}
}
	class Storage_A {
	static int $resource_0$ = 0
	...
	void add_by (int by_0)
	{
	int tmp_0 = 0
	synchronized (this) {
	tmp_1 = $resource_0$
	}
	...
	synchronized (this) {
	$resource_1$= tmp_1 + by_0
	}
	}
	}

Example 3.1. A Java program with ill-adapted synchronized blocks

For programs such as this, it is possible to automatically derive a post-condition or assertion from the SSA form of a method that manipulates an object or variable; such an assertion reveals the logical relationship between a shared object, input and the method. In this simple case it is

$$resource_1 = resource_0 + by_0.$$

This is obtained by repeatedly substituting all (*uses* of) variables on the right side of the last definition of the shared variable with appropriate *dominating* definitions. It may contain ø-functions and $ø_i$-expressions. We can then run two experimental threads or a code analyser (with suitable input if required) that exhaustively interleave each other by lock-step within the same method (or others if involved). Whenever a thread finishes executing the method, the manipulated resource can be compared with the derived assertion, so that if in any case the value of the resource does not comply with the assertion, then we can conclude that there is a possible race condition. This way of detecting race conditions does not depend on the locking discipline [13], meaning that even incorrectly used synchronized blocks can be considered.

4. Worst case execution time (WCET) and schedulability analysis

Once we acquire the low-level representation of a program along with the high-level structural information, it becomes easy to derive the WCET of each thread. We should then be able to conduct schedulability analysis, provided all the *jitters* of the underlying kernel library are known.

[3] Although in genuine SSA forms all dominating definitions can be referenced, we force this rule that only allows for any *use* to refer to an immediately preceding definition in order to make it harder to tamper with, i.e. the version number of any *use* must match with that of the most recent definition of the variable at a given point of a program.

4.1. Register allocation priority and cache memory management

In the SSA form of a program it is possible to determine which variables are *modified* more often than others by simply examining the definition numbers of variables. In other words, if a variable has a greater number of definitions than others, that variable should be given a higher priority to be allocated in a register, leading to a more efficient program. Moreover, this simple finding may possibly encourage the investigation of cache effects in performing WCET analysis, given that high-priority variables are cached in preference to others.

5. Implementation status

At the time of writing, we are still in a later design stage of the mobile representation and run-time system, investigating further advantages of our approach and more efficient techniques for safety checks and timing analysis. It is planned to use an open-source Java compiler, such as Kopi [10], to implement our SSA form-based optimiser and transformer, as well as an open-source real-time kernel that supports all the services required to implement the profile.

6. Related works

There are several mobile representations proposed over the past years, but almost none of them address temporal predictability. Java Bytecode is a relatively high-level stack-based representation. It is either interpreted by a VM or compiled by a JIT compiler into native code. Most of the necessary optimisations are done at the target for security reasons, so that the size of bytecode is not compact [4]. Slim Binaries are a tree-based mobile representation [8]. It makes use of techniques for code encoding, e.g. a semantic dictionary, and dynamic code generation, all resulting in very compact mobile code.

The ANDF [15] is another tree-based representation that was initially designed for the distribution of programs written in sequential programming languages. There have been some works to improve the format in terms of flow analyses, predictability, and verifiability, for example, the Safety-Critical ANDF (SC-ANDF) [3].

SafeTSA [1], like our approach, is a SSA form based mobile representation. High-level control structures and strict type information are embedded in code, enabling effective code optimisation and safety checks at the target.

7. Conclusion

This paper describes our on-going work on a new mobile code representation and run-time architecture that should hopefully facilitate developing safer and more predictable high-integrity systems. Our approach is based on the use of SSA forms. Some formal verification/reasoning techniques including the Hoare Triples are also (or are planned to be) applied to check the correctness of programs. However, the current safety policy is limited to type checking and data referential integrity testing, which raises some issues on application-dependent safety requirements. A more rigorous timing analysis is also still required to consider cache and pipeline effects. Further work is needed to implement the customised compiler and run-time system, and comparisons to other approaches should be made in due course.

8. References

[1] W. Amme, N. Dalton, M. Franz, and J. Von Ronne, *SafeTSA: A Type Safe and Referentially Secure Mobile-Code Representation Based on Static Single Assignment Form*, Accepted for the 2001 ACM SIGPLAN Conference on PLDI 2001.

[2] A. W. Appel, *Protection against untrusted code*, IBM DeveloperWorks, available at http://www-106.ibm.com/developerworks/library/untrusted-code/, written in 1999, last accessed in October 2001.

[3] N. Audsley, I. Bate, and A. Grigg, *Portable Code: Reducing the Cost of Obsolesence in Embedded Systems*, IEE Computing and Control, vol. 10, pp. 98-104, June 1999.

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

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

[6] 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.

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, *Efficiently computing Static Single Assignment Form and the Control Dependence Graph*, ACM Transactions on Programming Languages and Systems, 13(4):451-490, October 1991.

[8] M. Franz and T. Kistler, *Slim Binaries*, Communications of the ACM, 40(12), December 1997.

[9] C. A. R. Hoare, *An Axiomatic Basis for Computer Programming*, Communications of the ACM, 12(10), October 1969.

[10] *The Kopi Project*, http://www.dms.at/kopi, last accessed in October 2001.

[11] D. Kozen, *Language Based Security*, Technical Report TR99-1751, Cornell University, 1999.

[12] P. Puschner and A. J. Wellings, *A Profile for High-Integrity Real-Time Java Programs*, Proceedings of ISCRC 2001.

[13] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro and T. Anderson, *Eraser: A dynamic data race detector for multithreaded programs*, ACM Transactions on Computer Systems, 15(4):391-411, November 1997.

[14] R. Vallee-Rai and L. J. Hendren, *Jimple: Simplifying Java Bytecode for Analyses and Transformations*, Technical Report: 1998-4, Sable Research Group, McGill University, 1998.

[15] X/Open Company Ltd., *X/Open Preliminary Specification: Architecture Neutral Distribution Format*, X/Open Company Ltd., UK, 1996.