Overview of Real-Time Java Computing

Yu Sun
Department of Electrical and Computer Engineering, Southern Illinois University Carbondale, Carbondale, IL, USA
sunyu@engr.siu.edu

Wei Zhang
Department of Electrical and Computer Engineering, Virginia Commonwealth University, Richmond, VA, USA
wzhang4@vcu.edu

Abstract
This paper presents a complete survey of recent techniques that are applied in the field of real-time Java computing. It focuses on the issues that are especially important for hard real-time applications, which include time predictable garbage collection, worst-case execution time analysis of Java programs, real-time Java threads scheduling and compiler techniques designed for real-time purpose. It also evaluates experimental frameworks that can be used for researching real-time Java. This overview is expected to help researchers understand the state-of-the-art and advance the research in real-time Java computing.

Category: Embedded computing

Keywords: Performance; Reliability; Java; WCET analysis; Compiler; Hard real-time systems

I. INTRODUCTION
Real-time systems ranging from aircraft and nuclear power plant controllers to video games and speech recognition have become an increasingly important part of our society. The small stand-alone real-time applications of the past are giving way to a new type of networked real-time systems that are running on heterogeneous computing and networking environments; for example, integrated battleship management, VoIP-based telecommunications, stock arbitrage, and automotive systems [1]. The software development for real-time systems has also become more and more complex, due to the increase in the size, longevity, and required features of these systems. Following its successful use in Web applications, middleware, and enterprise software development, Java can offer significant advantages such as productivity, reliability, and portability for developing large and complex real-time software, and thus becomes an attractive option for real-time software design. A study by Nortel Networks indicated that using real-time Java doubled their productivity in developing telecom equipment and base stations [2].

For real-time systems, especially hard real-time and safety-critical systems, it is crucial to know the worst-case execution time (WCET) [3] to ensure that each task can meet its deadline. WCET should be computed based on static timing analysis, rather than measurement alone, because in general it is impossible to exhaustively measure all program paths in order to locate the longest execution time. The WCET of a real-time task however, is heavily dependent on the programming language used to implement this task, as well as the target processor and...
the compiler optimizations. Java, designed to run its byte code on Java Virtual Machine (JVM) for portability across different computing platforms, is a serious challenge for predictable execution and WCET analysis. Without solving this problem, Java cannot be safely used in hard real-time and safety-critical systems. Even for firm or soft real-time systems, unpredictable and varied execution time of Java computing may significantly compromise the quality of service.

Traditional JVM designs have mainly focused on improving the average-case performance, which may result in too much unpredictability with the use of multiple threads, dynamic compilation, automatic garbage collection, and so on. Since the first discussion of issues in design and implementation of real-time Java [4] was proposed in the 1990s, a considerable amount of remarkable research work has been done to adapt Java to real-time purpose. The Real-Time for Java Experts Group began to develop Real-Time Specification for Java (RTSJ) in March 1999 [5]. The RTSJ has been evaluated for uses in avionics and space systems by Boeing and Jet Propulsion Laboratory (JPL) [6, 7]. There are also a number of commercial implementations such as Sun Microsystems Mackinac, IBM WebSphere Real Time, TimeSys jTime, and Aicas JamaicaVM, as well as several open source implementations including OVM and jRate. A more recent effort regarding safety-critical Java software is the Safety-Critical Java (SCJ) Specification [8].

Due to the widespread use of real-time applications and the increasing use of Java in developing real-time software with hard deadlines, it becomes important for real-time researchers and developers to understand how to achieve time predictability in Java based computing. This paper presents an overview of the most recent and important studies in the area of real-time Java computing. More specifically, this paper will present related studies on real-time Java in the following topics:

- Real-time threads and scheduling
- Time predictability of Java code
- Bounded garbage collection
- Suitable compilation and optimization

The rest of this paper is organized by these four main issues. Sections II to V present the newest work on one of the above topics. Section VI describes a special way to implement real-time Java by hardware, that is, Java processor. Section VII discusses available tools and frameworks that may be used for real-time Java research and the last section gives the summary and conclusion.

II. REAL-TIME THREADS AND SCHEDULING

The RTSJ [5] by Java Expert Group firstly gives an abstract definition of real-time threads and scheduling. Two kinds of real-time threads are defined in addition to normal Java threads. One is real-time thread and another is no-heap real-time thread. NHRTs are not permitted to access the heap so that it can avoid the possible delay caused by garbage collection. On the other hand, RTs can access heap so that they have more flexibility in coding and are suitable for code with a higher tolerance for longer delays, such as soft real-time application.

The RTSJ also provided minimum requirements for real-time Java scheduling. All implementation of RTSJ must provide a fixed-priority preemptive scheduler with no fewer than 28 unique priorities. RTSJ is open for extension of other scheduling algorithms, and the implementation relies on the support of real-time operating systems. Similarly to RTSJ definition, the Ravenscar-Java [9] defines its real-time threads and scheduler. Additionally, it derives two specific types of threads from RTSJ: periodic real-time threads and sporadic event handlers. This design is based on the behavior of real-time applications whereby most threads wake up and do their job after fixed time intervals.

III. TIME PREDICTABILITY OF JAVA CODE

It is necessary for real-time Java applications to know the WCET of Java programs. Since Java code is firstly compiled into Java byte code (JBC) and then executed on JVMs, it is quite natural to analyze the WCET of a Java program in two steps: JBC level and lower platform-dependent level.

A. WCET Analysis at JBC Level

There have been many research efforts to conduct WCET analysis for C programs. Lundqvist and Stenstrom [10] discovered timing anomalies in out-of-order superscalar processors. Li and Malik [11] and Li et al. [12] proposed the implicit path enumeration technique (IPET) to compute the worst-case path for deriving the WCET accurately. Recently, the timing analysis has been extended from single-core processors [13-19] to multicore processors [20-28]. A good summary of contributions in the area of WCET analysis can be found in [3].

Java itself is very friendly to JBC level WCET static analysis with its well-formed object-oriented structure and features. JBC stored in Java class files is easily read and analyzed by the WCET analyzer, which generates control flow graph (CFG) and basic block (BB) of given Java programs. With Java annotations which are supported by most modern Java compilers, additional information such as loop bounds can also be provided.

A series of works have been done in this area. Puschner and Bernat [29] described a general method that can be applied for the Java program using the integer linear pro-
programming (ILP) technique. Bernat et al. [30] provided an implementation of the WCET analyzer based on Java annotations. Bate et al. [31] modified the Kaffe [32] and Komodo [33] to support WCET of Java applications running on these two JVMs. Control flow and data flow are both considered in these studies. The works in [34, 35] designed an extension to bring loop bounds, timing modes, and dynamic dispatch semantics into the WCET analyzer. Harmon and Klefstad [36] then attempted to construct a standard of Java annotations for WCET analysis, based on all previous works. Hepp and Schoebelr [37] explored WCET-based optimizations for Java programs.

B. Platform-Dependent WCET Analysis for Java

The only gap between JBC WCET and reality is the low level, platform-dependent timing model of JVMs. The exact execution time of each JBC and combination of JBCs is needed to compute the final WCET of any Java program. Hu et al. [38] attempted to solve this problem by two methods: profiling based and benchmark based. The profiling method inserts instrument codes into the Java program and collects execution times of JBCs. As another way, by running specially designed benchmarks, the same job can be done on all kinds of platforms without changing the instrument codes. Bate et al. [39] studied the JBC execution overhead due to the JVMs and processor pipelines, as well as the effects on the WCET of each JBC. However, these works are all measurement based. There is still no way to statically analyze the low level WCET for a particular JVM and platform.

IV. REAL-TIME JAVA MEMORY MANAGEMENT

Java’s automatic memory management, garbage collection (GC), is a very good feature and brings great benefit to software development. However, most current garbage collectors are not time predictable. As a result, GC actually prevents Java programs from being adapted in the real-time area. It is totally unacceptable that a real-time task is interrupted by GC thread and does not know when it can be up and running again.

A. Bypassing Garbage Collection

The first idea is to remove the unpredictable GC from the real-time Java system. That is why the scoped memory and immortal memory section are defined in RTSJ [5]. In this case, the objects in real-time threads are managed by programmers instead of JVMs. The developers take care of the memory areas that are used for real-time tasks and leave the other part to GC. The GC thread holds lower priority than real-time threads and thus cannot interfere with them. Time predictability is guaranteed but the flexibility is lost in development. Most of the implementations of RTSJ support this kind of technique bypassing GC. Besides, researchers made an effort to improve the efficiency of this scoped memory. Beebee and Rinard [40] implemented the scoped memory model and evaluated its efficiency. Corsaro and Schmidt [41] presented another implementation of real-time JRate. Pizlo et al. [42] presented an informal introduction to the semantics of the scoped management rules of the RTSJ, and it also provided a group of design patterns which can be applied to achieve a higher efficiency of scoped memory. Andreae [43] provided a complete programming model of the scoped memory model to simplify the memory analysis that has to be done by developers. Bypassing GC is proved to be easy to apply to existing JVMs and brings small overhead to runtime memory management. However, it is limited because of the great difficulty it brought to the developers. Without the GC feature, programmers have to pay much more attention to the objects in real-time tasks to avoid memory error. Although scoped memory may be a good solution of small short-term projects, real-time GC is definitely necessary for future real-time Java systems.

B. Time-Predictable Garbage Collection

Baker’s incremental copying collector [44] is the first idea of real-time GC. In this work, the memory mutator operation leads to the GC operation. Hence, GC is predictable and the worst case is that every read or allocation invokes a certain amount of collection operations. It is not very efficient yet provides a way to implement real-time GC. This so called work-based GC is then developed by many other following researchers, such as [45] and [46] in JamaicaVM. The basic idea is kept while overhead of allocation detection and unnecessary collection is significantly reduced. Even hardware can be used to assist GC efficiency and reduce WCET bounds, as presented in [47].

On the other hand, Bacon et al. [48] developed a time-based approach that invokes a collector at regular intervals. This kind of GC requires read/write barriers in order to maintain consistency, and the memory allocation must be time predictable. The quota of CPU time for GC can be dynamically computed and set at runtime based on the memory usage of real-time threads. This technique is then applied in IBM’s real-time JVM [49] named as Metronome [50, 51]. Henriksson [52] also proposed a time-based GC strategy which set the collector to active only when the processor is idle. The idea then is improved in [53]. The key issue of time-based real-time GC is how much time should be given to the collector. Schoebelr and Vitek [54] presented the algorithm to compute the GC quota and interval in their study. Cho et al. [55] used...
C. Concurrent Garbage Collection

Both time-based and work-based real-time GC have the same problem, that is, the processor is fully occupied by the collector thread at the GC stage. Thus thread switching has to be performed when a real-time task is presented. Fortunately, the improvement of multiprocessor technique brings a new approach that can avoid this problem. The concurrent GC runs on a different processor other than real-time tasks. As a result, the overhead of switching threads is reduced to a minimum. However, there are several serious challenges: synchronization between processors, access lock to memory objects, and so on.

The first multiprocessor concurrent GC implementation was presented by [56], which applied the algorithm in [57] on a 64-processor machine. It greatly reduced the pause time of GC to the millisecond level. After that, various kinds of concurrent GC algorithms and implementations were presented by researchers to further improve it. Xian and Xiong [58] showed that their technique can effectively reduce the memory amount used by concurrent real-time GC, which is relatively huge. Sapphire [59] implemented a copying collector for Java with low overhead and short pause time. Pizlo et al. [60] compared three lock-free concurrent GC algorithms: STOPLESS [61], CHICKEN, and CLOVER, which have the pause time as microsecond level. Although these algorithms are designed for and implemented by C#, it is easy to adapt them to Java since they are similar VM based language.

V. COMPILATION FOR REAL-TIME JAVA

Compilation and optimization are also critical issues of real-time Java. Just-in-time (JIT) compilation is already widely used in modern JVMs because of the performance boost it provides. Speculative optimizations are also an essential part now in many advanced JVMs. However, the JIT compiler and some optimizations bring unpredictability into Java applications, so that they cannot be simply used in real-time Java systems, where an old interpreter and ahead-of-time (AOT) compiler take their chances.

A. Interpretation

Interpretation is the most original way to execute JBCs. It reads the JBCs from classes and translates them into native code, and then executes them. It is slow but has excellent time predictability as the interpretation of each JBC can be easily measured. For this reason, most real-time JVMs, such as IBM WebSphere VM [49], Sun Java Real-Time System [62], and Open VM [63], provide the interpreter mode. However, the performance limits this mode to be used in modern real-time systems.

B. JIT Compilation

To improve interpreter performance, JVMs use the JIT compiler to compile the code sequence into native code on the fly before it executes. However, the JIT-only strategy introduces compilation overhead at runtime. To address this problem, a JVM can focus on optimizing only “hot-spots”, while the rest of the code can be either interpreted or compiled by a basic compiler without optimization. Examples of adaptive optimization systems include HotSpot virtual machine [64] from Sun (now Oracle), Jikes RVM [65] from IBM, and Open Runtime Platform (ORP) [66] from Intel Corporation. To identify hot-spots, researchers have proposed to use online hardware profiling mechanisms such as counters and samplings [67-71], or to use program instrumentation [72-78], combined instrumentation and sampling [79-81], or coupled offline and online profiling [82]. To further improve adaptive optimization, a number of techniques have been developed; for example, recompilation [83], deferred and partial compilation [84-86], and dynamic deoptimization [87]. The idea of dynamic and adaptive compilation has also been extended and studied in other contexts, among them hybrid JIT compilation [88], trace-based parallelization [89-93], and adaptive garbage collection [94].

While the JIT compiler is useful for improving the average-case performance of non-real-time Java applications, for real-time systems, the JIT compiler has two main drawbacks. One is that it interrupts other threads from execution and the time it takes is unpredictable. Secondly, the speculative optimizations are not suitable for real-time applications. [95] compared AOT and JIT compilers and [96] introduced their implementation and evaluation in IBM WebSphere VM. However, the JIT compiler is still capable of soft real-time systems, as reported in [1]. A carefully managed priority is necessary and some optimizations must be turned off. In addition, Sun and Zhang [97, 98] explored multicore processors to improve time predictability of dynamic compilation.

C. AOT Compilation

The AOT compiler does most of the work before execution, with only the dynamic part left to runtime. The AOT compilation can provide higher performance than interpretation, while keeping good time predictability. However, the part that the AOT compiler cannot complete before execution is very crucial to the performance. For example, the AOT compiler has no idea of class references nor dynamically generated classes. It has to use a resolution thread to patch such information during runtime. Furthermore, none of the aggressive inlining can be performed by the AOT compiler due to lack of class references, which loses a great chance to improve perfor-
VI. JAVA PROCESSORS FOR REAL-TIME COMPUTING

Aside from these software solutions, the JOP [101] Java processor is also open-sourced, in VHDL format. So it is possible to combine it with some simulators such as SimpleScalar or Trimaran so as to build a multiprocessor system. There is some research work [110] focusing on multiprocessor, and I think this is a very promising research field.

VII. EXPERIMENTAL FRAMEWORKS

The research resources of real-time Java are quite limited. Only a few real-time JVMs are completed and fewer of them are under an open-source license.

IBM [49] and Sun Microsystem (now Oracle) [62] are two main companies who provide well-developed commercial real-time Java products. Evaluation or academic version of their real-time JVM can be obtained from the Internet. However, the source code is unavailable.

OVM [63] is a good choice. It is open-sourced, supports most of RTSJ’s features, and is still an active research project [6, 107] with some documents. It has been tested in our lab and appears to work well. There are two main problems with OVM: first, its JIT compiler is quite simple and incomplete (lack of dynamic class loading); second, OVM is obviously designed for a single processor. Extra work is needed if we want to do some research on multicore systems.

An alternative may be jRate [108], whose source code is also available. jRate is an extension to the GNU GCJ compiler and a group of runtime libraries. It implemented most features needed by RTSJ.

Another candidate is Jikes RVM [65]. Although Jikes RVM is not designed for real time purpose, it is the most completed open-source JVM. Actually the prototype of the Metronome garbage collector is implemented on Jikes RVM. So it is possible to develop real-time extension for Jikes RVM.

A Java library named Javolution [109] may be helpful. It is an extended library on Sun Java implementing RTSJ. It is open-sourced.

As can be seen in this overview, real-time Java computing is an active and promising research field. There
are many research challenges and opportunities as well. Based on this survey, further investigation is still needed in the following directions:

- Time-predictable dynamic compilation for real-time Java applications
- Real-time Java on multiprocessor (multiple uniprocessor OR uniprocessor+Java processor)
- Low level Java WCET analysis with architectural timing information (cache, branch prediction, etc.)
- Multiprocessor/multicore real-time GC algorithms

REFERENCES


51. D. F. Bacon, P. Cheng, and V. T. Rajan, “Controlling frag-
Overview of Real-Time Java Computing

Mellon University, Pittsburgh, PA, 1974.


Yu Sun

Yu Sun received his Ph.D. degree in Electrical and Computer Engineering in 2010 at Southern Illinois University Carbondale, Carbondale, IL, USA, and B.E. degree in Computer Science at Tsinghua University, Beijing, China in 2003. His research interests include compiler optimization, parallelization, embedded system and real-time system. He is currently working as a senior software engineer at MathWorks, Inc.

Wei Zhang

Dr. Wei Zhang is an associate professor in Electrical and Computer Engineering of Virginia Commonwealth University. Dr. Wei Zhang received his Ph.D. from the Pennsylvania State University in 2003. From August 2003 to July 2010, Dr. Zhang worked as an assistant professor and then as an associate professor at Southern Illinois University Carbondale. His research interests are in embedded and real-time computing systems, computer architecture, compiler, and low-power systems. Dr. Zhang has received the 2009 SIUC Excellence through Commitment Outstanding Scholar Award for the College of Engineering, and 2007 IBM Real-time Innovation Award. His research has been supported by NSF, IBM, Intel, Motorola and Altera. He is a senior member of the IEEE. He has served as a member of the organizing or program committees for several IEEE/ACM international conferences and workshops.