Java for Real-Time Programming

Jan Lindström (jplindst@cs.helsinki.fi)

Helsinki 5th November 1998

University of Helsinki
Department of Computer Science
Seminar on Object Architectures, Fall 1998

1 Introduction

SUN Microsystems initially developed Java [1] [2, 1] as a tool to support internal development of small embedded systems. Java was found appropriate for development and distribution of Internet applications and it was released to the general public. One distinguishing charasteristic of Java is its execution model. Java programs are first translated into a fully portable standard bytecode representation. The bytecode is then available for execution in any environment that provides support for a Java virtual machine. A Java virtual machine is simply a system of software that understands and executes the standard bytecode representation. Initial Java implementations interpret bytecodes, but recently released implementations provide the ability to translate bytecodes to native machine code on the fly. Sun calls this just-in-time (JIT) compilation. Another technique for improving Java performance, known as ahead-of-time translation, is to translate bytecodes to machine code prior to deploying an application. JIT and ahead-of-time translation techniques offer the potential of providing performance that approximates the execution speed of C++.

Design of Java was originally motivated within Sun by the special needs of embedded system development. Sun is promoting the use of Java for embedded systems programming. Java, as it has been defined and implemented, lacks important capabilities necessary for the reliable development of portable real-time applications [9].

In this paper we present shortcomings of current Java [4, 5, 8, 9] implementations and a proposal for a real-time dialect of Java [6, 7, 10], which provides better support for real-time software development. This paper is organized as follows. In section 2 we briefly present real-time programming. In section 3 we present shortcomings of current Java implementations and section 4 presents proposed real-time diaclect of Java. Finally section 5 summarizes the paper.

2 Real-Time Programming

Computer programs that must execute within particular time constraints are said to be real-time programs or applications. Examples of real-time applications include voice synthesis, air traffic control, control of robots, telephone switching, and full-motion multimedia playback. Real-time tasks can be divided to four categories [4]:

- Periodic: executes regularly according to a precomputed schedule. Examples of periodic tasks include sampling of pen position, processing radar or sonar signal inputs, and playback of full-motion video.
- Sproradic: triggered by external events or conditions, with a predetermined maximum execution frequency. Examples of sproradic events include mouse clicks on particular buttons, and alarms raised by high temperature or pressure readings.

[1] Java is a trademark of Sun Microsystems, Inc.

- Spontaneous: optional real-time tasks that are executed in response to external events, if resources are available. Robot walking provides several examples of spontaneous tasks. Prior to each step, a number of independent real-time tasks must be scheduled, one to control the behavior of each joint of each leg that is to be moved during this step.
- Real-Time Threads: fair-share threads. These are tasks that run ”forever”, but which must make forward progress in proportion to the passage of time. Consider, for example, an application that is responsible for analyzing stock market trends.

Special implementation techniques are required of programmers who desire to enforce real-time constraints. In particular, programmers must determine the memory and CPU-time requirements of each real-time task independently, and then must analyze the collective workload of all the tasks that comprise the system workload. Writing real-time software consists tree main parts:

- Designers specify real-time constraints. By far, the most common requirement is for periodic servicing of hardware devices.
- Enginneers implement routines to provide the necessary services and analyze these routines to determine their worst-case execution times.
- Engineers analyze the combined workload to determine whether individual real-time tasks can be proven to not interfere with each other’s real-time requirements.

One common technique for analyzing system workloads is known as rate monotonic analysis [3]. With rate monotonic scheduling, developers assign task priority in order of decreasing execution frequency. Tasks that execute most frequently are assigned the highest priority. Mathematical analysis of worst-case senario demonstrates that all of the tasks will complete their execution prior to their next period of execution as long as the combined workload represents less than 69% of the CPU’s total capacity.

Developing real-time software is notoriously difficult and very costly. To exercise full control over real-time behavior require extensive machine-dependent analysis. As a result, implementation of real-time systems is mostly performed through a process of ”black magic”. Through human wizardy and experimentation, developers find the right combination of parameters that minimize costs while maximizing performance and compliance with real-time constraints. Many real-time programmers now do their development in C and C++. However, these languages do not provide mechanisms to allow programmers to describe real-time constraints. Consequently, programmers are required to enforce real-time requirements using combinations of compile-time analysis tools, pre-run-time measurements of application resource requirements, and run-time interaction with non-standard, real-time operating system services.

3 Shortcomings of Java

Development of real-time Java programs is especially difficult. The developer of a Java application has no idea how powerful the CPU and how much memory will be available in the Java virtual machine environment in which the application is to run [9]. In real-time systems, application developers take responsibility for ensuring that the software executes on schedule. In general, analyzing intertask timing dependencies is very difficult. And this analysis is especially difficult in the highly dynamic Java execution environment.

Java applications that will require more predictable real-time performance are already under development. These include real-time character animation, computer music synthesis, full-motion audio and video playback, and video conferencing. Sun is promoting the use of Java as a general-purpose programming language. They are marketing a new operating system called JavaOS [2] for use in small embedded systems. These small systems are to run only applications written in Java. JavaOS systems may also be employed in more traditional embedded environments including manufacturing automation, telephone switching, security systems, intelligent air conditioner control, and reactive robots. All of these require varying degrees of compliance with real-time constraints.

In its current form, Java is not appropriate for the development of real-time software. In order to offer real-time developers the promise of a portable real-time execution environment, certain aspects of the Java language specification that Sun currently describes as undefined need to be more rigorously specified.

In a real-time application, it is important that programmers are assured that memory will be available for allocation when new objects need to be allocated. It is also important that background garbage collection does not impose arbitrarily long delays at unpredictable times. This would interfere with the developer's ability to demonstrate compliance with real-time constraints. Current Java implementations do not address these issues [9]:

- To distinguish live objects from garbage, most Java implementations use a partially conservative scanning technique. Memory words containing values that represent legal memory addresses are assumed to represent pointers. Since these words may actually hold integer or floating-point values result is memory leak.
- Because of previous point, it is not possible to relocate live objects in order to defragment the memory heap. Over time, the cumulative effects of fragmentation may make it difficult to allocate large objects. This is especially troublesome for embedded systems that are expected to operate reliably for weeks at a time.
- In Sun's Java implementation, garbage collection is implemented as a low-priority background thread. If any application tasks are CPU-bound, the garbage collector is not scheduled for execution until the system runs out of memory. The first allocation request that

[2] JavaOS is a trademark of Sun Microsystems, Inc.

cannot be satisfied from the existing free pool triggers a stop-and-wait garbage collection of the entire system. This force all other tasks in the system to suspend until garbage collection completes.

- By design, the Java run-time environment does not allow applications to determine how much memory they require or how much total memory is available in the execution environment.
- Standard Java libraries provide no ability to request or enforce memory budgets.

The traditional Java environment does not provide any mechanisms to allow programmers to specify that tasks should execute at particular times. Applications can request to sleep for a specified amount of milliseconds before continuing their execution. However, there is no guarantee the task will be suspended no longer than the requested amount of time, and there is no guarantee the task will have the highest priority at the time it is made ready for execution. There is also no way for a given Java task to determine how many other tasks are running on the system, their relative priorities, and fraction of CPU time they consume. Thus, there is no way to assure that a paricular task will have sufficient CPU-time resources to execute within its real-time constraints.

Java uses monitors to protect critical sections of code from simultanous access by multiple tasks. Once a particular task has entered into the monitor corrensponding to a particular object, no other task can acces that object's monitor code until the first task has exited the monitor. In order to analyze the time required to perform certain actions, a real-time developers must know how long each task might have to wait for entry to monitors. Other problems with Java monitors include [9]:

- In the highly dynamic Java execution environment, the nuber and priorities of other tasks sharing access to a particular object are difficult to determine.
- Java imposes no restrictions on the complexity of code contained within a synchronized method. The code may comprise unbounded loops, dynamic method invocations, and nested entry in other monitors.
- Specification of Java's standard libraries fails to specify precisely the scheduling model, and fails to specify protocols for avoiding priority inversion [11].

Since the time and memory requirements of Java programs are not known until the Java bytecodes have been loaded into the environment in which they are to execute, mechanisms must be provided within the execution environment to analyze these resource requirements. For instance, Java's standard libraries fail to provide a protocol whereby newly loaded task would be able to determine how much CPU time they require to execute reliably on the host platform; they also lack mechanisms to enable applications to determine how much memory is required to represent particular objects in the local execution environment.

4 PERC Real-Time API

Java offers important high-level benefits to developers of traditional software. It would be desirable for real-time dialect of Java to improve upon the state of the practice in real-time development. Requirements for writing portable real-time applications include:

- Applications need to be able to determine their time and memory resource needs and intertask blocking times.
- API must allow applications to request dedicated resources.
- Virtual machine must commit resources to particular real-time workloads.
- API must allow applications to manage resources dedicated to their executions.

A complete description of the proposed real-time extensions is impractical here – what follows is only a high-level overview of the PERC [3] [10] real-time API, the implementation of which is currently in progress.

PERC is a superset of Java 1.1 in that it offers additional syntax and additional time- and memory-related semantics to the Java programmmer. However, it is a subset in that it forbids certain legal Java practices in cases when the use of these practices would interfere with the system's ability to support reliable complicance with real-time requirements.

PERC source code is translated to standard Java byte codes by a special Java compiler named **percolate**. Besides translating standard Java source code, **percolate** recognizes and translates the special **timed** and **atomic** control structures that are part of the PERC real-time extensions. Further, **percolate** does additional analysis of the PERC source code and encode certain of the results of this analysis in the generated byte code files. The purpose of this analysis is to assist the worst-case execution time analyzer in efficiently identifying segments of code whose execution time must be analyzed. Further PERC includes [10]:

- Resource configuration. The configuration process consists of determining how much CPU time and memory is required to support the workload on the given platform.
- Resource negotiation. The real-time executive analyzes the resource request to determine if it has sufficient resources to satisfy the request.
- Reconfiguration. As new activities are introduced to the system or existing activities are removed, the system-wide allocation of resources may need to be revised.
- Resource budgeting. The PERC real-time API provides support for a variety of mechanisms to determine the resource requirements of the activity in the local execution environment.

[3] PERC is a trademark of NewMonics, Inc.

- The Embedded and RealTime Toolkits. The PERC real-time API implementation is divided into two packages named **com.newmonics.RealTime** and **com.newmonics.Embedded**. The **RealTime** package encapsulates abstractions relevant to reliable budgeting of time and memory. The **Embedded** package encapsulates abstractions relevant to programming of embedded computer systems.

PERC consists of a combination of special class libraries, standard protocols for communicating with these libraries, and the addition of two time-related control structures to the standard Java syntax. Following keywords are added to Java:

- **timed** identifies a segment of code whose execution must be time bounded. For example, the following describes a code segment that is allowed to execute no longer than 30 μs:

```
try {
    timed(Time.us(30)) {  // throws TimeOutException
        Method();
    }
} catch(TimeOutException t) {
    DoSomething(t); // Error
}
```

- **atomic** identifies a segment of code whose execution must be all or nothing. For example, the following describes a code segment where money is transferred between two accounts:

```
try {
    atomic{
        BeginTrans();
        Withdraw(FromAccountId,Sum);
        Deposit(ToAccountId,Sum);
        Commit(); // throws AbortException
    }
} catch(AbortException t) {
    DoSomething(t); // Error
}
```

PERC real-time API includes also extensions to units of time, asynchronous exceptions, restrictions on exception handling, and restrictions on use of Thread methods.

5 Summary

As discussed, real-time development using current state-of-art technologies is mostly ”black magic”. It is also tedious and costly. The resulting systems are inflexible and fragile. Further, development of real-time software is inherently non-portable. Programmers are forced to target particular operating systems and particular hardware architectures. Java offers important high-level benefits to developers of traditional software. It would be desirable for real-time dialect of Java to improve upon the state of the practice in real-time development. In its current form, Java is not appropriate for the development of real-time software.

References

[1] K. Arnold and J. Gosling. *The* $Java^{TM}$ *Programming Language. The* $Java^{TM}$ *Series.* Addison-Wesley Publishing Company Inc., 1996.

[2] Sun Microsystems Inc. The $Java^{TM}$ language environment: A white paper. Technical report, Sun Microsystems Inc., Mountain View, California, 1995.

[3] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[4] K. Nilsen. Issues in the design and implementation of real-time Java. *Java Developer's Journal*, 1(1):44–57, June 1995. http:// www. newmonics. com / WebRoot / technologies / java / RTJI.pdf.

[5] K. Nilsen. Embedded real-time development in the Java language. In *Proceedings of the Embedded Systems Conference*, pages 1–12, http:// www. newmonics. com / WebRoot / technologies / java / java-esce.pdf, April 1996.

[6] K. Nilsen. Real-time Java (v. 1.0). Tech. report, Iowa State University, Ames, Iowa, http:// www. newmonics. com / WebRoot / technologies / java / rtjava.api.pdf, January 1996.

[7] K. Nilsen. Real-time Java (v. 1.1). Tech. report, Iowa State University, Ames, Iowa, http:// www. newmonics. com / WebRoot / technologies / java / rtjava.api.v.1.1.pdf, February 1996.

[8] K. Nilsen. Invited node: Java for real-time. *Real-Time Systems Journal*, June 1997. http:// www. newmonics. com / WebRoot / technologies / java / java-rtsj.pdf.

[9] K. Nilsen. Adding real-time capabilities to Java. *Communication of ACM*, 41(6):49–56, June 1998.

[10] K. Nilsen and S. Lee. $PERC^{TM}$ real-time API (draft 1.3). Tech. report, NewMonics Inc., http:// www. newmonics. com / WebRoot / technologies / java / perc.api.pdf, July 1998.

[11] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, C-39(9):1175–1185, September 1990.