July 19, 1996

Issues in the Design and Implementation of Real-Time Java[1]

Kelvin Nilsen

This report, first distributed as a Postscript document available for anonymous ftp downloading on Nov. 15, 1995, serves as a foundation for discussion regarding standardization of Java extensions designed to support development of reliable real-time software. This revision of the document includes limited discussion regarding issues that have been raised since the original document was first published.

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

Executive Summary

Current difficulties in developing and maintaining software within limited budgets and challenging development-schedule constraints result in part from the shortcomings of existing programming languages and development environments. In the context of distributed programming for the World Wide Web, Sun Microsystems has recently introduced the Java programming language environment. Since formal announcement of Java in May of 1995, a number of companies have already signed agreements to license Sun's technology. Among these new partners is Netscape and Oracle.

Java was originally designed by Sun to facilitate the development of embedded system software [1], but has been initially positioned as a language for Web programming because of its ability to simplify the development of flexible, portable, distributed applications with high-level graphical user interfaces [2]. Java is derived from C and C++, but the language has been restricted to eliminate many of the most costly common programming errors. Java shows promise of greatly improving developer productivity in the targeted domains.

Since many embedded computer systems must comply with real-time constraints, the question might be raised: "Can Java support the development of reliable hard and soft real-time applications?" This report suggests that Java, as it has been announced and distributed to date, is not appropriate for development of real-time software. However, by combining certain Java programming conventions with special implementation techniques, it is possible to support varying degrees of real-time reliability, ranging from 100% guaranteed compliance with hard-real-time constraints to 100% best-effort (unguaranteed) compliance with soft-real-time constraints. Users who require hard-real-time performance will, of necessity, need to pay much more for their execution hardware in order to prove that worst-case needs will always be satisfied. The same real-time

software that has been designed to run with 100% reliability on special Real-Time Java virtual machines will run reasonably well on less expensive Java virtual machines that are not capable of guaranteeing compliance with hard-real-time constraints. The benefits of a Real-Time Java standard include lower development and maintenance costs, quicker time to market, increased portability, enhanced network connectivity, improved reliability, and increased functionality of real-time systems.

One of the most exciting potential benefits of the Real-Time Java execution model is the support it provides for development and reuse of portable real-time software. Not only does Java enable the creation of real-time software modules to be shrink wrapped for use on a variety of different Java execution platforms in cooperation with arbitrary mixes of other concurrently executing real-time activities, but it also allows reliable integration of software written in multiple different languages. For example, work is currently under way to retarget an Ada compiler to generate Java virtual machine code. Furthermore, the Java real-time execution model allows integration of non-real-time COTS[1] software components as optionally executed components of real-time tasks. This makes available to developers of Real-Time Java applications large libraries of highly functional reusable Java object definitions.

The purpose of this paper is to raise important issues, suggest general solutions to the system-level problems, and point out low-level design issues that require further design refinement. This report places emphasis on refinement of design objectives and narrowing of implementation choices. It considers discussion of language syntax and class interface designs to be premature at this time.

Discussion Topics

Though this report takes the form of a proposal and argues informally for the adoption of certain standard practices, the main purpose of distributing the report in this form at this time is to encourage discussion and refinements, and to recruit support for the total effort. We would hope to be able to involve a broad interdisciplinary assortment of experts in these discussions. Some of the particular questions we intend to address include:

- Multimedia Developers: Are these abstractions useful? Do they offer important capabilities not provided by “unadorned” Java? Is more than what has been proposed necessary?
- Real-Time Developers: Are the proposed abstractions useful? Do they meet your needs for developing flexible and portable real-time software? Or should standardization of additional capabilities be attempted? Are the benefits to be provided by a “Real-Time Java” programming environment sufficient to justify the anticipated costs?
- Java Implementors: Do you feel that the proposed methodologies could be implemented (to the desired degree of compliance with real-time constraints) within the context of your current implementation efforts? Does this report adequately forewarn Real-Time Java developers regarding the variety of different operating conditions under which their “portable real-time” software might be expected to operate?

1. COTS stands for commercial off-the-shelf.

- Java Standardization Bodies: Is there any hope of standardizing a Real-Time Java specification such as has been suggested in this report? How would we go about pursuing this possibility? Which aspects of the proposed approach do you find most objectionable? What sort of compromises might be more tolerable?

At the same time we are striving to better understand how best to define the problem that must be solved, we are also involved in the formation of a new corporation[1] which intends to tackle parts of this large problem as its first commercial products. With this in mind, we seek interaction with possible business partners, investors, and future customers.

To become involved in ongoing discussion and refinement of the Real-Time Java standard, subscribe to the real-time-java mailing list by sending the message body SUBSCRIBE in an email message to real-time-java-request@iastate.edu.

Introduction

Java's Origins

As reported in [1], Java "originated as part of a research project to develop advanced software for a wide variety of networked devices and embedded systems." The research project initially chose to use C++ for development. But subsequently, the developers encountered so many difficulties with C++ that they decided it would be best to design an "entirely new language environment." Java offers a number of important improvements over developing software in currently popular languages such as C and C++:

- Java borrows the familiar syntax of C and C++. Like C++, Java is object-oriented, but it is much simpler than C++ because Java's designers intentionally discarded redundant language features that were present primarily to support backward compatibility with legacy code. An additional benefit of its simplicity is the small size of its run-time system. Sun reports that the basic interpreter is about 40 Kbytes, and that basic libraries and thread support add approximately 175 Kbytes [3].
- The development cycle is much faster because Java supports both interpreted and just-in-time compiled implementations. During development and rapid prototyping, developers save time by using the interpreter.
- Application software is more portable because the Java environment carefully specifies a machine-independent intermediate byte-code representation which can be transferred between heterogeneous network nodes and interpreted or compiled to native code on demand by the local Java run-time environment.
- Application software is more robust because Java's run-time environment provides automatic garbage collection[1]. The Java language has been designed to eliminate the possibility of dangling pointers and memory leaks.

1. NewMonics Inc. was incorporated on March 20, 1996. Its address is 2501 N. Loop Dr., Suite 900C, Ames, IA 50010. Phone: 515-296-0897, Fax: 515-296-9910.

1. Garbage collection describes the process of automatically detecting memory cells that are no longer in use and adding them to the free pool so that they can serve future allocation needs.

- Applications are adaptable to changing environments because code modules can be downloaded dynamically from other network nodes without necessitating a system restart.
- Security is enforced by built-in protection against viruses and other tampering. This protection is implemented by simple "theorem provers" that analyze downloaded byte codes before attempting to execute them.
- High performance is achieved by incorporating support for just-in-time translation of portable byte codes to the native machine language of the local host. According to Sun, performance of translated code is roughly equivalent to the speed of current C and C++ programs[1].

Following public announcements of Java in early 1995, acceptance among developers has grown very rapidly. Within less than two weeks following its creation, the Usenix comp.lang.java news group was already carrying close to a hundred articles per day. And as of Oct. 9, 1995, Java-related electronic mailing lists comprise over 10,000 independent names. Netscape and Oracle have both recently contracted with Sun Microsystems to incorporate Java run-time environments into future World Wide Web browsers.

Java for Real-Time

According to David Wilner, Chief Technical Officer of Wind River Systems, most of the recent rapid growth in the embedded real-time marketplace has been in areas for which time-to-market pressures and high-volume per-unit costs are primary considerations of project managers [4]. Besides simplifying the development of WWW applications, most of Java's benefits enumerated above would be of great utility to developers of embedded real-time systems as well. A Real-Time Java implementation would be useful both for programming of distributed real-time WWW applications (e.g. stock market trading, interactive animation, video games, and teleconferencing) and for implementation of more traditional embedded real-time applications (e.g. in-vehicle navigation systems, pen- and voice-based computer interfaces, air traffic control, virtual reality environments, and missile defense systems). Clearly, real-time developers also need simple object-oriented languages that support rapid development, portability, robust operation, dynamic reconfiguration, security, and high performance. Some of the problems particular to the embedded real-time system domain which are addressed by the design of a Real-Time Java programming environment include:

- **`Portability:`** developers of embedded real-time systems often find it necessary to deal with a number of different host processors. For example, it is common to find mixtures of Motorola 68000, Power PC, and Intel 960 hosts in a single development laboratory. Even within an architecture family, code generation needs to be targeted to the specific host in order to achieve high performance (e.g. The PPC 604 chip needs different code than the PPC 601). This is particularly important in consumer electronics products for which both software and hardware components evolve during the product family's lifetime. Maintaining the appropriate cross-development tools for all of these different hosts is an administrative nightmare. Ensuring that all combinations of hardware and software components work correctly in a real-time

1. Sun has not yet provided any proof of this claim. Given the nature of the language and characteristics of its standardized implementation, it seems unlikely that Java will really run as fast as optimized C code. In the current absence of actual data, it seems more realistic to estimate that optimized Java will run on average approximately 20 – 30% slower than optimized C.

sense is a difficult challenge. And maintaining object code revisions and custom makefile configurations is a major headache for developers. Development would be much easier if programmers could write in Java (which by design is a fully portable language), maintain a single version of the translated byte codes, and depend on the localized Java run-time environments to configure new code as it is loaded onto particular hosts.

- **`Dynamic Adaptability:`** A frequent difficulty with maintenance of embedded real-time systems is that incremental software refinements must be installed without bringing the system down for a clean restart. This includes applications that must provide non-stop service to their user community (e.g. flight traffic control, telephone switching, and military reconnaissance), and applications for which the costs of downtime are prohibitive (e.g. nuclear reactor control, and manufacturing automation systems).
- **`Fault Tolerance:`** In the presence of network or node outages, it is often necessary to redistribute information and processing workloads. The Java programming environment greatly simplifies this burden since any node in the network, regardless of its processor architecture, is capable of performing any of the Java tasks that need to be performed. Simply download the byte-code representation of the task.

In response to a recent comp.lang.java post enquiring whether current Java “enthusiasts” felt there would be a market demand for Real-Time Java implementations, several individuals expressed strong support for the idea and requested additional information so that they could promote this possibility among their management. Eugene Devereaux, a Senior Principal Scientist at Boeing Company, stated that his company is beginning an “R&D project at Boeing’s Airplane Systems Laboratory (ASL) to look at using Java with real time projects.” Bruce Wong of Distributed Systems International, Inc. pointed out that “the security features and automatic storage management eliminate a whole class of programming errors that would manifest themselves as crashes or core dumps.” He projects: “I believe the benefits will be so great that it will be a crime if market forces do not make [embedded Real-Time Java] happen.”

The Needs of Real-Time Developers

Traditionally, real-time development has required that programmers analyze software prior to execution to make sure that all execution time needs of the software will be available when required. Resources that must be proven available include memory and CPU time.

Traditional Techniques for Real-Time Development

Traditional techniques for real-time development are very costly for a number of reasons [5]:

1. Greater care is required during development to perform the analysis required to guarantee compliance with real-time constraints.

2. Because most general purpose programming environments do not support real-time performance, developers of real-time applications are forced to use specialized operating systems and development environments. These environments lack the robust and powerful development tools that are available to developers of more traditional systems.
3. Since the techniques for analysis of real-time performance are intrinsically machine dependent, extra analysis effort is required to maintain real-time software in the presence of continual processor upgrades and integration of new I/O devices.
4. Because it is not practical to accurately predict worst-case task execution times, the real-time developer must reserve resources based on conservative upper bounds rather than actual worst-case requirements.
5. Even if it were possible to accurately determine worst-case execution times, execution time must be reserved for every task's worst-case performance. Note that for most tasks, typical execution times are much smaller than worst-case execution times.
6. Because of the difficulty of proving availability of dynamic memory, many developers of hard-real-time systems avoid all use of dynamic memory management. This requires that all of the memory required to satisfy each task's worst-case needs must be permanently allocated prior to execution. Memory not currently in use sits idle rather than contributing to the system's functionality and capacity.
7. The scheduling techniques that are typically used in real-time systems are unable to promise 100% CPU utilization. Furthermore, context switching is much more frequent in many real-time systems than is typical of traditional systems. As a result, context switching and scheduling overhead represent a significant fraction of a real-time system's total workload.

For all the costs and difficulties associated with the development of real-time applications using these "traditional" methodologies, you would think that the resulting benefits are well worth the effort. But in fact, most real-time systems fall far short of the ideal in terms of functionality and flexibility. Real-time software is notoriously brittle and feature poor. "There *has* to be a better way."

Current State of the Practice

The real-time methodologies described above are so cumbersome and costly that most developers of consumer real-time systems seem to ignore them entirely. Witness the current state of the practice:

1. Home computer users are expected to tweak and tune numerous mysterious parameters to enable multimedia applications to run "correctly" on their single-tasking personal computers. When not configured properly, both full-motion video and audio stutter.
2. Personal computers ignore typed characters and mouse clicks if entered when the computer is busy with other activities.
3. Personal digital assistants are unable to record voice dictations during receipt or transmission of faxes. Occasional 30-second response delays are common.
4. In current multi-tasking environments, such as high-performance desktop Unix systems, multimedia applications run well if they are the only applications running, but the real-time behavior of the multimedia applications degrades rapidly and uncontrollably when other CPU-intensive activities are added to the system's workload.

5. Developers of real-time multimedia applications for personal computers must provide large technical support staffs to help end users debug, configure and maintain their applications running in the users' ever changing computing environments. Not only does the environment's software change, but it is also continually undergoing hardware upgrades in the form of add-on cache, memory, I/O expansion boards, and networking capabilities.

Summary

Real-time programmers approach software differently than developers of traditional systems. But traditional methodologies for development of hard-real-time systems are not appropriate for the majority of mass-market real-time software that needs to be developed in the coming years.

A Portable Model for Real-Time Computation

Though each real-time application domain imposes a different set of operating constraints on the real-time execution environment, most mass-market real-time applications fit within the model described below. Even many of the real-time applications that have traditionally been solved using more traditional real-time methodologies can be accommodated by this model.

Periodic Tasks

Many activities in existing real-time systems consist of tasks with fairly consistent typical execution times. These tasks are invoked at a regular time within a fixed period of execution. Examples of periodic real-time tasks include sampling of pen position in a pen computer, processing of radar or sonar signal inputs, playback of full-motion video at a predetermined frame rate, and recording of digitally sampled speech for teleconferencing applications. Typical execution frequencies for these tasks range from 10 to 1,000 times per second. Note that the mix of periodic tasks that comprise a real-time system may change. For example, all of the tasks related to a teleconferencing session are removed from the system workload when the teleconferencing session closes. It is important that implementors recognize that changes to the task mix occur much less frequently than execution of individual periodic tasks.

Sporadic Tasks

Another significant percentage of the total workload consists of sporadic activities which are typically triggered in response to particular external events. When responses to these external events must be delivered within a specified time, the activity is described as a sporadic real-time task. Examples of sporadic events that might require real-time response include mouse clicks on particular buttons, alarms raised by high temperature or pressure readings, and radar detection of a previously unidentified flying missile.

Spontaneous Tasks[1]

Spontaneous tasks represent an even smaller percentage of the total system workload. These are similar to sporadic tasks in that they are triggered at unpredictable times, in

1. In a previous draft of this document, the word "dynamic" was used in place of "spontaneous" for characterization of real-time tasks with unknown execution frequency.

response to events or conditions detected in the external environment. Unlike sporadic tasks, there is no upper bound on the frequency at which spontaneous tasks are executed. Another difference between spontaneous and sporadic tasks is that no resources are preallocated or guaranteed a priori for execution of spontaneous tasks. Each time an occasion arises to instantiate a spontaneous task, the real-time executive determines on the fly whether sufficient resources are available to service the request.

There is no periodic behavior in spontaneous tasks. A spontaneous task is characterized by a desired start and finish time. If sufficient resources are available to add the requested spontaneous task to the current workload, without renegotiating the current workload, the real-time executive accepts responsibility for executing the spontaneous task. Otherwise, the real-time executive declines the request. If developers are unable to tolerate the possibility that a task might be declined, they should describe the task as a sporadic rather than spontaneous workload.

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. Suppose the desire is to optimize the speed of robot walking. Since the computation and time required to take each step on an uneven terrain varies greatly, it would be wasteful to describe walking as a periodic task in which sufficient time is reserved in each period for whatever is the worst-case time required to take a step on the worst possible terrain. Better efficiency would be realized by treating each step as a spontaneous activity and scheduling the next step as soon as the previous step has successfully completed. In this manner, the robot would be able to walk very rapidly over flat surfaces, but would be able to slow down as necessary when climbing or descending hills. Note that in both cases, a step should not be taken if any of the relevant real-time tasks cannot be scheduled. This suggests the need for a two-phase protocol for execution of spontaneous tasks. In the first phase, the real-time executive determines whether it can accept the additional workload. In the second, it executes the workload.

Real-Time Threads

Another type of real-time activity might best be characterized as a fair-share thread. 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 in order to alert traders to opportunities to trade at favorable profits. Suppose this task is responsible for generating an updated report once every ten minutes. Much of the work required to prepare report *N* is redundant with the work required to prepare report *N – 1*. Assume it is most natural to implement this task as a large loop that makes incremental refinements to previous recommendations based on the receipt and assimilation of whatever new data has arrived since the previous report was generated. Given this, it would not be appropriate to implement this activity as a traditional Java thread because Java threads have no control over how much time they might be allowed to execute. Programmers need a scheduling abstraction in which this application can be treated as a thread with guaranteed execution time. In particular, the programmer desires to specify that the task will be granted a certain amount of execution time during each ten minute period, with additional control over how the time is distributed within the ten minute period. Note that it would not be very useful for this task to have all of its execution time granted at the beginning of the ten-minute period because that would require it to make all of its recommendations without considering the additional ticker-tape information that is likely to arrive prior to generation of the recommendation report.

What is Real-Time Java?

Unlike most languages designed for real-time programming, Java was designed more to simplify programming than to enable programmers to write software that complies reliably with real-time constraints. Many real-time engineers would argue that the spontaneous nature of Java is totally inappropriate for real-time application development. Nevertheless, Java has much to offer the real-time programmer. By combining special real-time implementations of the Java virtual machine with Java code written to comply with special conventions for description of real-time activities, it is possible to develop Java applications that rigorously conform to real-time execution constraints. The same code, executed on non-real-time implementations of the Java virtual machine, also supports soft real-time performance. Both real-time and non-real-time virtual machines can run combinations of real-time Java code and non-real-time Java code.

Architecture of a Real-Time Java Program

Preparatory to developing detailed designs and implementations of Java classes, it is necessary to reach consensus regarding the general role to be served by each class. The emphasis of this section is on specifying the general functionality of the major classes that comprise the Real-Time Java API[1].

It is important to keep in mind that simplicity is one of Java's greatest strengths. If it is not possible to preserve simplicity while adding real-time capabilities to Java, then Real-Time Java really has nothing more to offer the embedded real-time community than is already offered by C, C++, and Ada. One of our greatest challenges is to design an architecture that supports development of both simple and complex programs without adding unwanted complexity to simple programs.

Much of Java's simplicity comes from the abstractions that have been built into the language. Abstraction helps programmers, including real-time programmers, deal with complexity. But real-time developers must use abstraction with discretion. They must be able to break through layers of abstraction whenever this is necessary in order to understand or exercise control over real-time behavior.

A Real-Time Java program consists of an arbitrary number of real-time activities accompanied by an arbitrary number of runnable threads. The runnable threads have no time-constrained behavior. The discussion of real-time activities provided below makes frequent reference to the real-time executive. See "The Real-Time Executive" on page 14.

1. NewMonics Inc. is currently in the process of refining the "Real-Time Java" API and is promoting a standard API under the product name PERC^{TM}, an acronym that stands for Portable Executive for Reliable Control. In a number of cases, the draft API differs slightly from the design suggested in this document.

A Real-Time Activity

A real-time activity consists of a *configuration manager*, an *administrator*, an arbitrary number of *real-time tasks*, and an arbitrary number of *runnable threads*. The point in specifying runnable threads as part of a real-time activity is to allow them to be packaged with the other components that comprise the real-time activity in such a way that they share access to particular variables. There are several different kinds of real-time tasks, independently known as *cyclic*, *sporadic*, *spontaneous*, and *ongoing*. For all but spontaneous tasks, the execution model assumes that tasks are ready to execute at the start of their period, and are allowed to execute any time within the period as long as they terminate prior to the end. Much research in real-time scheduling has focused on obtaining real-time schedules for more precisely specified constraints. This report takes the position that such refined control adds unnecessary complexity to the real-time developer's job. Instead, real-time programmers can find ways to structure their real-time activities within the proposed scheduling model. The main benefit of imposing this restriction is that it enables efficient modularization and integration of independently developed portable real-time activities on a single shared real-time execution platform.

It might appear that our proposal is based on an implicit assumption that rate monotonic and static cyclic scheduling techniques are sufficient to satisfy the needs of all applications. Note, however, that additional scheduling control can be self-implemented within particular real-time activities. For example, if a particular real-time activity desires to use earliest-deadline-first scheduling, its administrator can describe its scheduling needs to the real-time executive as an ongoing real-time task that requires 30 milliseconds of execution time in each period of 100 milliseconds. This ongoing real-time activity would decide for itself which of its internal "tasks" to schedule. In this implementation style, internal tasks might best be represented by object methods. This is an area that requires further study in order to refine the selection of standard services with which application programmers can develop their own self-scheduled real-time activities.

It is important to emphasize that the notion of task priority in a rate monotonic real-time environment is very different from the notion of priority in traditional systems. In the real-time environment, priorities are selected according to decreasing order of execution frequency, and have nothing to do with the relative "importance" of individual tasks. In order for real-time activities to reliably coexist with uncooperative traditional non-real-time components, we recommend that the range of priorities available for prioritization of traditional Java threads all be lower than the range of priorities dedicated to real-time activities.

Configuration Manager. When a new real-time activity is introduced into the system, the real-time executive invokes the activity's configuration manager to allow it to adjust for the local computing environment. Configuration consists of determining which methods will be interpreted and which will be translated by the just-in-time compiler, calculating method and task execution times, and determining the activity's memory requirements. Configuration management is the appropriate place to implement inter-task blocking analysis, if such analysis is necessary. According to the Real-Time Java software architecture, tasks are only blocked by other tasks that are part of the same real-time activity. In order to support configuration management, the run-time environment needs to make certain services available. For example:

1. A task execution time analyzer that determines worst-case execution times for simple control structures through analysis of the control-flow graph.

2. A task execution time analyzer that determines typical execution times for arbitrary tasks by measuring representative executions.
3. A memory requirements analyzer that allows the configuration manager to determine the local sizes of particular data structures.

Note that the configuration manager is developed by the same team of programmers that writes all of the real-time tasks that comprise a particular real-time activity. The rationale for this software architecture is that these programmers are in the best position to determine what sort of configuration information is necessary for their real-time activity to run reliably in the current environment.

Administrator. The responsibility of the administrator is to negotiate for resources with the real-time executive. Following invocation of a real-time activity's configuration manager, the real-time executive invokes the activity's administrator. The administrator communicates the real-time activity's resource needs, based on analysis of the configuration manager, to the real-time executive. Execution time requirements are described to the real-time executive in terms of execution frequency, minimum execution time, and desired execution time for each of the cyclic, sporadic, and ongoing tasks that comprise the real-time activity. Prior to communicating its execution time needs to the real-time executive, the administrator may choose to adjust the activity's task periods so that they align more evenly with the real-time executive's existing least common multiple of real-time task periods. In response to the administrator's resource requests, the executive provides pessimistic, expected, and optimistic resource budgets. The two resources that are managed during this negotiation are execution time and dynamic memory. The pessimistic budgets represent a lower bound on the amount of the resource that will be provided to the activity. Non-real-time virtual machine implementations may not be able to guarantee any resources at all, in which case they report a pessimistic budget of zero. The expected budget is the amount of the resource that the real-time executive intends to provide to the activity, assuming average operating conditions. The optimistic budget reports the maximum possible amount of the resource that will be made available to the activity. It is the activity's responsibility to make effective use of whatever resources are made available to it. The activity's administrator initializes relevant instance variables to represent the activity's budgets so that the individual tasks that comprise the real-time activity can pace themselves appropriately.

In an execution environment that is as dynamic as Java, resource budgets must be continually reevaluated. Whenever the real-time executive must reevaluate budgets, it establishes a dialogue with the corresponding activities' administrators to renegotiate the resources that are available to the respective activities. Examples of events that might trigger the real-time executive to renegotiate budgets include:

1. When new real-time activities are introduced into the environment, resources may need to be withdrawn from current activities.
2. When old real-time activities become inactive, additional resources may become available to the remaining real-time activities.
3. Certain real-time activities may discover that their resource needs have changed. These activities can communicate this information to the real-time executive, which may respond by revisiting the resource allocation decisions made previously.

Atomic Code Segments. Special syntax is provided to identify certain segments of code as atomic[1]. The rationale for requiring atomic segments to have bounded execution times is as follows:

1. Analysis of blocking interactions between tasks requires knowledge of how much time particular tasks may be excluded from making forward progress.
2. One possible lock-less implementation of atomic segments invoked from within real-time tasks is to check the time remaining on this task's time slice on entry into the atomic segment and to allow entry only if the time remaining is greater than the known time required to execute the atomic segment.
3. Another possible lock-less implementation would be to simply disable all interrupts during execution of the atomic segment. This implementation would not strictly comply with real-time requirements in cases for which execution of the atomic code segment might result in the task being allowed to execute longer than its budgeted time (because the timeout tick could not be delivered while interrupts were disabled). Nevertheless, by bounding the duration of the atomic segment, developers can analyze the amount of jitter that might be introduced into system performance by this possibility. In fact, Java implementors might choose to select between alternative feasible implementations of atomic segments, depending on the calculated worst-case times required to execute them.

All Java implementations, regardless of the degree to which they might fail to rigorously comply with all real-time execution constraints, must execute atomic code segments according to the following requirements:

1. Either the atomic segment is executed in its entirety or not at all, insofar as visible side effects are concerned. Partial execution of an atomic segment is permitted only if there are no visible side effects.
2. Execution of the atomic segment may be preempted (and later resumed) only by threads that are unable to see or manipulate the intermediate state resulting from suspending the original thread in the middle of executing the atomic segment. In practice, the most straightforward implementation may be to simply prohibit all preemption of atomic segments, thereby eliminating the analysis that would be required to demonstrate that certain tasks cannot possibly see or modify the variables managed within the atomic segment.

As discussed below, startup and finalization components of real-time tasks are also treated as atomic segments. Even in execution environments that lack the ability to determine through analysis the worst-case execution times of startup, finalization, and atomic segments, atomicity is never compromised. In such environments, worst-case execution times can be approximated through measurement. Whenever execution of an atomic segment causes the corresponding task's time slice to be exceeded, the real-time executive corrects the problem as quickly as possible by shortening the amounts of time

1. Atomic segments resemble, but are not identical, to Java's synchronized methods. The synchronized qualifier enforces the idea that only one thread at a time is allowed to execute particular code segments at a time. An "atomic" qualifier also allows only one thread at a time to access the corresponding code segment. Additionally, the atomic qualifier guarantees that either the entire segment or none of the segment will be executed each time entry into the segment is attempted.

available to subsequent tasks, until all tasks are once again executing on schedule. In a strictly complying real-time implementation, however, execution of atomic segments should never result in particular tasks being allowed to run longer than their budgeted times.

Cyclic real-time tasks. A cyclic real-time task is characterized by a single runnable thread; a startup segment with bounded worst-case execution time; a finalization exception handler with bounded execution time; one or more atomic segments of code, each of which has bounded execution time; and a desired execution frequency. Each component is optional, but it would not be meaningful to omit all components.

The startup segment combined with the finalization segment represent the minimal functionality offered by this task. Each of these segments is always executed atomically and in its entirety. The runnable thread represents a variable-quality component of this task's effort. Typically, the startup segment computes a very rough estimate of the task's intended result and the runnable thread makes incremental improvements to this initial rough estimate. The activity's configuration manager and administrator work together to arrange for a reasonable amount of execution time to be allocated to the runnable thread, on average. When the runnable thread's time expires, the real-time executive aborts the thread if it hasn't already terminated and passes control to the finalization method associated with the task. In cases for which the work to be performed by a task is small and constant, all of the work can be performed by the startup and finalization code, and the runnable thread can be omitted.

The rationale for this software architecture is that it is not economically feasible to determine accurate worst-case execution times for tasks of even moderate complexity [6]. A measurement-based analysis of task performance is much more accurate for typical execution behavior. But measurement-based analysis does not represent worst-case behavior. Whenever the task requires more time than was anticipated by the measurement-based analysis, it is better to deliver an approximate answer on schedule than to run the risk of pushing all other tasks in the system off schedule.

Because it is not practical to derive tight worst-case bounds for execution of each real-time task, we make no attempt to do so. Rather, we guarantee sufficient resources to execute only the task's startup segment and its finalization segment (each of which is characterized by a "conservative" worst-case execution time bound).

The general execution model for a cyclic real-time task is for the real-time executive to invoke the initialization method and then to startup the runnable thread with a watchdog timer set to prevent this thread from taking longer than its allotted time. If the thread terminates on schedule, the real-time executive then calls the finalization exception handler explicitly. Otherwise, the real-time executive aborts the thread by sending it the finalization exception. Note that this protocol guarantees that the finalizer will be invoked exactly once for each execution of the cyclic task.

Descriptions of other kinds of real-time tasks follow. There are many similarities between the various flavors of real-time tasks, and it is our intention that the object-oriented class hierarchy that implements real-time activities will represent these similarities. In the descriptions that follow, details already provided in the description of cyclic real-time tasks are intentionally omitted.

Sporadic real-time tasks. A sporadic real-time task consists of a single runnable thread; a startup segment with bounded worst-case execution time; a finalization exception handler with bounded execution time; one or more atomic segments of code, each of which has bounded execution time; and a worst-case execution frequency. As with cyclic real-time tasks, all components are optional. Such tasks are typically triggered by:

1. An interrupt which is translated by the run-time system into an activation of this task, or
2. Upon recognizing a particular condition, a cyclic task activates the corresponding sporadic task.

Note that it may be possible for hardware interrupts to occur at a higher frequency than was specified by the Java programmer. If this occurs, the ability to comply with real-time constraints may be compromised. In fact, a vulnerability of many current real-world systems is that they can be crashed by overloading the system with externally generated interrupts, such as might result from LAN network broadcast storms. In order to achieve reliable compliance with hard real-time execution constraints, the software developer must coordinate with the hardware implementors to ensure that the hardware does not generate more frequent interrupts than have been specified.

Spontaneous real-time tasks. A spontaneous real-time task consists of a single runnable thread; a startup segment with bounded worst-case execution time; one or more atomic segments of code, each of which has bounded execution time; and a finalization exception handler with bounded execution time. The minimum amount of time scheduled for execution of a spontaneous real-time task is the sum of the startup and finalization segments. The run-time executive takes responsibility for interrupting the runnable thread when the time remaining in the task's time slot equals the time required to execute the finalization segment.

Ongoing real-time tasks. An ongoing real-time task consists of a single runnable thread; a startup segment with bounded worst-case execution time; one or more atomic segments of code, each of which has bounded execution time; a finalization exception handler with bounded execution time; and a desired resumption frequency. Unlike cyclic tasks, this thread is resumed rather than being restarted on each period of execution. The real-time activity's configuration manager coordinates with the activity's administrator to arrange with the real-time executive that each resumption of this task is of sufficiently long duration.

Because a typical task's execution needs are not entirely deterministic, the ongoing real-time task may find it necessary to adjust the quality of its efforts on the fly. It does this by pacing itself against the real-time clock.

Conceptually, an ongoing task runs forever. But in practice, the activity in which the ongoing task is a participant may not last forever. When the corresponding real-time activity terminates, the real-time executive invokes the ongoing task's finalization exception handler.

The Real-Time Executive

The primary responsibilities of the real-time executive are to make and enforce resource allocation decisions. Whenever real-time activities are added to or deleted from the real-time workload, or whenever the resource needs of existing real-time activities change, the real-time executive must decide how much CPU time and how much dynamic memory can be budgeted to each activity.

This report suggests that the negotiation process and resource allocation decisions need not be time constrained. While negotiations are taking place, the system continues to execute the workload that was previously negotiated. Once negotiations are complete, the system transitions “instantaneously” to the newly negotiated workload. If the new workload replaces a cyclic schedule, the transition may be delayed until the end of the current cycle.

Since spontaneous tasks are not periodic, they are handled specially. The real-time executive provides a service which takes as input parameters descriptions of an arbitrary number of spontaneous tasks and schedules the tasks for execution if sufficient resources are available to satisfy all of the tasks’ needs, or reports failure and schedules none of the tasks if any of the task’s execution requirements can not be satisfied. Each task description consists of a start time, a completion time, a minimum execution time, and a desired execution time. By design, incorporation of additional spontaneous tasks into the workload does not trigger reconfiguration of the existing cyclic schedules. Spontaneous tasks are accepted for execution only if they can be serviced without interfering with the cyclic tasks scheduled previously.

To some degree, the efficiency of resource utilization is correlated with the effort spent in making careful resource allocation decisions. However, optimal resource allocation is NP-hard, and even though we impose no real-time constraints on the resource allocation problem, it is the implementor’s responsibility to provide “responsive” performance. Perhaps this is one aspect of the Real-Time Java design that requires further refinement? Should we quantify the timeliness of resource negotiation? Should we allow particular resource allocations to be saved and restored on the fly, so as to avoid the effort required to renegotiate each time the system mode changes?

To conclude discussion of the real-time executive, we point out that both time and memory can be allocated with very efficient straightforward algorithms. If implementors desire to use more sophisticated allocation techniques, more power to them. First, consider allocation of memory. Sum all of the memory allocation requests. Upon completion of each garbage collection pass, compare the amount of memory used by each activity with the specified upper bound on memory needs for that activity. Call the difference for activity i d_i. Sum the differences to obtain the total amount of additional memory that might legitimately be requested by these activities. Call this D. Let P represent the total amount of free memory at the time that garbage collection completes. Compute the memory allocation increment budget for activity i by multiplying d_i by P/D.

Now, consider calculation of static cyclic schedules and analysis of sporadic workloads to determine whether they are run-time schedulable. First, compute the total workload represented by sporadic and periodic tasks as a percentage of the CPU’s total capacity.

Note that following resumption of a task that was preempted, memory that had been cached prior to the preemption may no longer be present. The impact of the preempting task on the cache contents of the preempted task is limited by the cache "footprint" of the preempting task. As described in reference [7], the overhead of cache interference between tasks can be modeled by adding to the cost of the preempting task the time required to restore discarded cache entries into the cache following execution of the preempting task. A conservative upper bound is represented by the time required to execute however many cache read misses are required to fill the complete cache. When computing the total workload, add this time to the cost of each cyclic and sporadic real-time task in the system. If the total workload is less than or equal to 69%, it is schedulable [8]. Otherwise, it may not be. For simplicity, assume that if the workload exceeds 69% it is not schedulable. In this case, we must shrink the workload before proceeding to construct the cyclic scheduling table. We shrink the workload by reducing the service quality of particular real-time tasks and/or by refusing to add new real-time activity's to the workload.

Once the workload has been sufficiently reduced, we construct a cyclic dispatch table by simulating a rate monotonic scheduler on all of the tasks that comprise the workload, simulating each sporadic activity at its worst-case execution frequency. Time slots corresponding to sporadic tasks are left idle in the cyclic schedule. During execution, application-level interrupts are enabled only during idle slots of the cyclic schedule. Assign priorities to sporadic tasks according to rate-monotonic order.

Various optimizations to this scheduling process are possible. Several simple optimizations that offer potential for relatively high payoff include:

1. When constructing the cyclic dispatch table, do not precisely simulate the rate-monotonic scheduling technique. If it is possible to eliminate preemption of a low-priority task by delaying execution of the high-priority task until after the low-priority task has completed (without violating the time constraints on the high-priority task), do so.
2. In cases for which it was necessary to degrade service quality in order to bound the total workload by 69% of CPU capacity prior to constructing the schedule, it may be possible to expand the amount of execution time granted to particular tasks once the cyclic dispatch table has been constructed.
3. Whenever the cyclic dispatch table includes chains of tasks that are executed one after another, additional analysis is performed to determine whether the tasks can be started ahead of schedule if the previous task(s) in the chain completes ahead of schedule. If so, configure the dispatch table to so indicate. This will result in longer, more useful "idle" times during which spontaneous and sporadic tasks and non-real-time threads can execute.
4. If the static cyclic schedule is considered to be too long because the least common multiple of task periods is too large, simply use rate monotonic scheduling for all tasks. The real-time executive would need to maintain a dynamic queue of awake times in order to trigger execution of periodic tasks at appropriate times.

Real-Time Management of Dynamic Memory

Ideally, the real-time programmer would be assured that all of the memory required for execution of a real-time activity would be available in the requested sizes at the desired times. While this may be possible in certain execution environments, it is not practical in others. And relatively high memory and/or run-time overhead costs are associated with providing these sorts of guarantees. As with management of CPU time, we propose that for most execution environments, memory be treated as a real-time resource that can be shown to be available most of the time, but cannot always be guaranteed. For those users who require absolute guarantees of memory availability, higher cost hardware implementations are available, assuming that they are willing to limit themselves to execution environments that have been specially designed to provide hard-real-time responsiveness. We view this compromise as unavoidable.

The dynamic memory needs of a particular real-time activity can be characterized in terms of the maximum amount of memory that the application needs to retain as live at any instant of time and the maximum rate at which the application needs to allocate new objects. The second of these parameters is directly related to the rate at which the garbage collector must reclaim objects.

As mentioned above, the real-time activity's configuration manager has the responsibility of determining the values of these parameters in terms of the local execution environment. The activity's administrator negotiates with the real-time executive to determine how much memory will be made available to the application. The real-time executive grants a memory budget to the activity's administrator which is expressed in terms of pessimistic, expected, and optimistic values for each parameter. The pessimistic budget represents the minimum amount of memory that the real-time executive will provide to this activity. In some execution environments, it will not be possible for the real-time executive to promise any amount of dynamic memory; thus application developers who desire to write code that runs in such environments should be prepared to deal with the possibility that dynamic memory cannot be guaranteed. The expected budget represents the amount of memory that the real-time executive expects to be able to provide based on average-case behavior of the garbage collector and the application. The optimistic budget represents an upper bound on the total amount of memory that the real-time executive intends to make available to the activity. For example, the sum over all real-time activities of their optimistic memory budgets probably should not exceed the total amount of available memory[1].

Implementation

There are many possible ways to implement the run-time support required for execution of Real-Time Java. The simplest and most portable implementation of the real-time executive would be written in Java itself and could be downloaded into anyone's existing Java virtual machine. Of course, such an implementation would not be able to provide a high degree of compliance with hard real-time execution constraints, but would at least serve as a common foundation upon which programs that care about the passage of real time could execute.

The Java Virtual Machine

In general, we recognize that virtual machine support for real-time activities is a matter of degree:

1. The sum may actually exceed the total amount of available memory in environments that do not enforce any sort of logical partitioning of the dynamic heap.

1. The nature of most current desktop computing environments is such that interference from, for example, other Unix processes is beyond the control of the Java virtual machine. In these environments, real-time response is provided “as much as possible.” Within the Java run-time system, resources are allocated according to the real-time execution plan, as it is adjusted dynamically to accommodate for interference from other non-Java processes. Note that because of non-determinism in the execution environment, it is even more important in these systems to be able to refine the plan for execution of real-time tasks on the fly.
2. In embedded Java systems (e.g. Java running on a single-tasking CPU, or, for example, on a dedicated network terminal), interference from other activities outside the control of Java is eliminated. However, such systems may choose not to use time-deterministic implementations of all language features in order to provide higher throughput and/or more efficient utilization of available system resources (e.g. memory).
3. In embedded Real-Time Java environments, great care would be taken to ensure that all components of the Java implementation work together to provide time-constrained execution of real-time activities. Average-case performance may suffer because of trade-offs selected in order to achieve tight real-time control.
4. In order to achieve the tightest possible latency bounds combined with highest possible system throughput, it will be desirable to integrate a custom software implementation with custom hardware designed to support real-time garbage collection and fine-grained control of time. With proper hardware support, tight real-time guarantees can be provided without degrading average-case system throughput.

Note that this approach represents an important advance over the current practice, even for run-time environments that do not guarantee rigorous compliance with real-time constraints. In current general purpose computing systems, the run-time environment has no awareness of what real-time behavior is desired and what parameters are available within which to adjust the execution of individual real-time activities in order to achieve the desired real-time behavior. Though many run-time implementations may not provide strictly “correct” real-time behavior, we expect that most environments will find it possible to quickly adjust for any real-time noncompliance by dynamically adjusting the quality of service of individual components.

Code Generation Model

In order to enable analysis of worst-case execution times of all atomic segments, including the startup and finalization code associated with real-time tasks, these code segments must be distinguished from other code by the Java translator. Whether the distinguishing characteristic consists of special byte-code instructions or simply of reserved method names remains to be determined.

The Byte-Code Analyzer

In traditional Java run-time implementations, Java byte-code programs are analyzed prior to execution in order to verify that the code conforms to expected conventions. In an implementation of a Real-Time Java run-time system, the byte code analyzer has the additional responsibility of determining through analysis the worst-case times required to execute atomic segments of code, including the startup code and finalization code associated with real-time tasks. We do not expect to be able to analyze arbitrarily complex code segments. Part of the design of Real-Time Java that remains to be refined is a standard that describes the control structures that the byte-code analyzer is capable of analyzing to determine worst-case execution times. Furthermore, we do not necessarily expect to obtain tight bounds on task execution times. The purpose of execution time analysis is to enable reliable operation of the real-time system. Scheduling decisions are based more on average-case measured task execution times rather than on worst-case times derived through static analysis.

Whenever possible, it is desirable that the byte-code analyzer also determine the worst-case stack size of each thread and real-time task. If this information is available, the creation of threads and tasks is likely to be more efficient in that the amount of memory set aside to represent the stack is typically smaller. Further, run-time efficiency is improved because the object methods invoked during execution of the corresponding thread or real-time task need not check for stack overflow.

Byte Code Translation

When translating code segments that are intended to be executed atomically, the byte code translator must generate code to enforce atomicity. For best performance, the atomicity enforcing code should be in-lined at the point from which the atomic segment is invoked. Atomic code invoked from within a sporadic task may need to be surrounded by invocations of kernel functions that provide dynamic mutual exclusion protection. When the same code is invoked from within a periodic task, the most efficient implementation may be to simply check the time remaining in the task’s time slice before entering into the atomic segment of code. If the run-time system is able to efficiently make this information available, the byte code translator should generate the code required to obtain and compare with this time.

The Real-Time Executive

The real-time executive has three major responsibilities:

1. To make resource allocation decisions. In order to minimize its impact on system throughput, such decisions are made relatively rarely. Once made, individual activities do much of the work required to manage the resources that have been granted them without necessitating further interaction with the real-time executive.
2. To dispatch tasks and raise watchdog timeout exceptions at appropriate times. These events may be very frequent (e.g. thousands of events per second) so it is important to minimize the effort required to service them.
3. To maintain an accurate representation of real time and make this available to independent real-time activities.

Resource allocation decisions. In the best of worlds, resource allocation is straightforward because there are sufficient resources to satisfy every application’s desires. But in the real world, resources are limited and every application desires as much as it can get. The simplest method for resource allocation is to divide resources equally in proportion to the sizes of each application’s requests. Alternately, it may be desirable for the resource allocator to treat certain activities as more important than others, thus favoring their requests for resources. We view these matters as local administrative issues and do not consider them to be part of the Real-Time Java programming interface. It appears that it would be straightforward for particular run-time environments to provide users with menus that enable them to specify their preferences in this regard. It also seems possible that, at the user’s discretion, automatic determination of importance could be provided by tools that, for example, monitor which windows are visible and/or active and automatically treat the corresponding activities as more important.

Dispatch of tasks and watchdog timers. Two of the most popular scheduling techniques for implementation of current real-time systems are static cyclic and rate monotonic scheduling. A static cyclic schedule, which is computed prior to execution time, is simply an agenda denoting time slots when particular tasks are to be executed. Once the agenda has completed, it is repeated. There are a number of reasons that static cyclic scheduling is often preferred in, for example, commercial avionics and military systems:

1. Because the schedule is computed prior to execution time, it is perfectly known exactly when each task will execute. There is no uncertainty, for example, in timing analysis introduced by the possibility that certain tasks will be blocked from executing important code by semaphore-like locks owned by lower priority activities.
2. There is very low run-time overhead associated with a static cyclic agenda because all scheduling decisions are made prior to execution time. The dispatcher treats the agenda as a circular queue and always looks at the head of the queue to determine the next event with which it must concern itself.
3. There is no need to incur any run-time overhead or kernel calls in the implementation of mutual exclusion locks. Such locks can be enforced prior to run-time by scheduling tasks in such a way that they are known not to interfere with one another.
4. Because scheduling decisions are made prior to run time, an arbitrarily large amount of effort can be spent in computing an efficient schedule. With sufficiently large expenditure of scheduling effort, it is possible to achieve 100% system utilization. With much less effort, utilization of 69% is easily achieved.

The main disadvantage of static cyclic schedules is that they are static, unchanging. Sporadic events are not easily handled in these environments, but changes in the system workload can be accommodated by replacing one cyclic schedule with another.

Rate monotonic scheduling characterizes a technique in which the worst-case execution times and the worst-case execution frequencies of all tasks are known prior to run time, but the exact times at which particular tasks will be invoked is not known. The general technique is to assign task priorities in order of decreasing execution frequency. So the task that executes most frequently has highest priority and the task with least frequent execution has lowest priority. The run-time scheduler has the responsibility of ensuring that at all times, the highest priority task that is ready to run is scheduled for execution.

Note that there is more run-time overhead with a rate-monotonic scheduler because the scheduler must maintain a priority queue of all tasks that are ready to run. Additionally, there is execution-time overhead and uncertainty introduced because of the need to enforce mutual exclusion using dynamic locks.

Our recommendation is to implement cyclic and ongoing real-time tasks using a dynamically constructed static cyclic schedule and to implement sporadic tasks using rate monotonic scheduling techniques. The scheduling of spontaneous tasks is implemented through the use of a second dispatch queue that interleaves time slots with the cyclic dispatch queue.

Ideally, tasks that are executing as part of a static cyclic schedule would be allowed to enquire of the run-time kernel to determine how much time is remaining in their current execution time allotment. This time can be used as a guidepost to determine whether it is possible to complete the next atomic segment of code based on knowledge of the worst-case time required to execute the atomic code. In systems for which this is possible, this offers an efficient implementation of mutual exclusion enforcement for cyclic and ongoing real-time tasks.

Run-Time Services

In Real-Time Java, most scheduling decisions are based on typical execution times rather than worst-case times. Since Java software is developed on different hardware than the systems on which it normally runs, and since developed Java byte codes run on a variety of different hardware configurations, it is necessary to analyze the code in the environment in which it is going to run.

The application developer is responsible for providing a configuration manager that is capable of exercizing each task in the local execution environment to determine its execution time requirements. In order to measure task execution times, the run-time system must provide the ability to accurately measure time. Furthermore, it may be desirable that the run-time environment provide an ability to invalidate and/or disable memory caches so as to allow measurement of code when it is not benefiting from cache speed-ups.

Facilities must be provided in the run-time environment to explicitly invoke sporadic and spontaneous tasks from within other real-time tasks, and for embedded Java implementations, to automatically invoke sporadic tasks in response to particular interrupts. The question of whether to expect the run-time system to enforce sporadic task execution frequencies remains to be addressed.

The real-time activity's administrator is responsible for negotiating with the real-time executive to obtain time and memory budgets for execution of the real-time tasks that comprise the activity. In order for the administrator to determine the time required to execute the atomic segments of code that comprise part of this activity, the run-time system needs to provide a mechanism whereby the worst-case execution times of atomic segments, as determined by the real-time version of the byte-code analyzer, can be determined. Perhaps this information can be obtained by invoking a special worst-case-execution-time method that is associated with each atomic segment object.

In order to allow an activity's administrator to adjust task periods so as to align them with the existing cyclic dispatch table, there must be a standard mechanism by which Java programs can determine the current length of the dispatch table.

At the Java source and byte-code levels of abstraction, all atomic segments of real-time code use the same representation. However, translation of atomic segment byte codes to native machine instructions will depend on the mutual exclusion enforcement mechanisms that are used in the host Java virtual machine implementation. In some cases, implementation may consist simply of disabling interrupts. In others, it may consist of setting a particular variable to point to the instruction that follows the critical segment of code so that any attempt to context switch could be preceded by execution of all of the instructions that precede the instruction so identified. Yet another possible implementation is to check how much time is remaining in the current task's time slot and only allowing entry into the atomic code segment if the remaining time is at least as great as the known worst-case time required to execute the atomic code segment. Whatever the implementation technique, it is important that the run-time support system provide the mechanisms necessary to implement it.

In embedded Real-Time Java systems, it will be necessary to develop custom device drivers for nonstandard hardware components. Such device drivers can be implemented as native methods. Regardless of implementation technique, it would be desirable to standardize the interface to custom device drivers. This would benefit both the implementor of the device driver and the device driver's users. Note that it will also be necessary to allow such native drivers to include interrupt handlers that trigger execution of sporadic Java tasks.

In order to support reliable real-time garbage collection, it is desirable to eliminate all aspects of conservative pointer scanning from the system. Otherwise, it is not possible to defragment memory and memory leaks may be introduced by conservative scanning. Consequently, we recommend that new conventions be developed for the implementation of native methods in environments that intend to provide fully accurate garbage collection. In these environments, all access to Java data structures must be stylized through, for example, use of C macros in order to make it possible for the garbage collector to accurately distinguish pointers from non-pointers within the native method's stack frame and static variables.

Real-Time Garbage Collection

There are a variety of garbage collection techniques that can, to varying degrees, support real-time garbage performance as it has been characterized in this report. Below, several of the feasible techniques are summarized briefly. This list is not intended to be exhaustive. First, we identify several of the ways in which the garbage collection subsystem may introduce unpredictability into the run-time environment.

Garbage collection faults. Since all of the threads and real-time activities running in a particular Java execution environment share use of a single dynamic heap, it is possible for one uncooperative thread to crash all other threads by simply allocating and hoarding all available memory. To prevent this from happening, the real-time executive must enforce dynamic memory allocation budgets. One possible implementation consists of:

1. Tagging every allocated object with a field that identifies the activity that allocated it.
2. Tallying the total amount of each activity's dynamic memory that is live at the termination of each garbage collection pass.
3. Dividing the free memory that is available upon completion of garbage collection between currently executing activities, reserving some of this memory for new activities that might be instantiated prior to completion of the next garbage collection pass. This partitioning of memory is represented as an activity-specific allocation budget.

According to this convention, task *A* cannot reallocate the memory corresponding to a newly dead object until after the garbage collector has reclaimed this memory. Note that the time required by the garbage collector to reclaim this memory is system dependent. However, this system dependency is properly abstracted by the real-time activity's administrator when it negotiates for access to dynamic memory both in terms of the maximum amount of live memory and in terms of the maximum rate of memory allocation. See "Administrator" on page 11.

The ideal garbage collector would instantaneously reclaim and defragment the memory associated with each object that becomes dead. However, practical implementations of garbage collectors need time first to recognize objects as dead, and then to defragment free memory segments. Therefore, the memory allocator may fail to satisfy legal allocation requests either because dead memory has not yet been reclaimed, or because reclaimed memory is fragmented to the degree that there is not a sufficiently large free segment currently available, even though the total amount of free memory is larger than the size of the request. In both of these cases, waiting for additional garbage collection to complete before reissuing the allocation request may solve the problem.

In this model of dynamic memory management, there are two possible reasons why an allocation request cannot be immediately satisfied: (1) the task's total allocation budget has been exceeded, or (2) the available free pool does not have a sufficiently large allocatable segment because of fragmentation. We recommend that individual tasks be able to distinguish between these two situations at the time their allocation requests are rejected. This suggests the need for a per-thread state variable that identifies the problem associated with the thread's most recently denied allocation request.

In a different execution model, the run-time system might allow tasks to allocate beyond their dynamic memory budget, under the assumption that the garbage collector will subsequently discover that the task has released sufficient memory to justify the allocation request. This allows more aggressive utilization of available memory amongst well-behaved trusted tasks. However, if the garbage collector subsequently discovers that a particular task's memory budget has been exceeded, the only way to recover from this error is to kill the task, reclaiming all of its memory. Meanwhile, availability of memory for other tasks in the system has been compromised. Additional time is required to reclaim the memory that had been erroneously allocated to the rogue task in order to make it available to the other tasks for which the memory had originally been reserved. Currently unresolved is the exact handling that is given to a task that is discovered after the fact to have violated its memory allocation budget. Presumably, we would define an exception to be raised in this case, and would disable any dynamic memory allocation from being performed within the corresponding exception handler. Upon termination of the exception handlers, the activity is considered to be dead.

Conservative mark and sweep garbage collection. Conservative garbage collectors are unable to reliably distinguish between pointers and non-pointers [9]. Because the compiler is not required to generate the code that would be necessary to tag pointers, conservative garbage collection is fairly easy to implement and has a low run-time overhead. In conservative garbage collectors, the garbage collector treats any word that contains a value that represents a valid address as a pointer. This means that it is possible for an integer whose value is in the range that represents legal addresses to cause dead memory to be conservatively retained. Furthermore, since it is uncertain whether the suspected pointer really represents raw data or a memory address, it is not possible for the garbage collector to relocate the referenced object in order to defragment memory.

Nevertheless, conservative techniques are very popular in implementations of garbage collection for C and C++, and it appears that they will also be popular in implementations of Java. They perform well on average, both in terms of memory allocation throughput, and in terms of memory utilization.

In order to make a conservative garbage collector compatible with real-time constraints, it is necessary to partition the free pool into segments of different sizes and to divide the total garbage collection effort over time. Suppose, for example, that free list 0 represents objects ranging in size from 16 to 31 bytes, that list 1 represents objects ranging from 32 to 63 bytes, and so on. Given this organization, the time required to allocate memory is bounded by the time required to examine each free list. On a 32-bit computer, there would be no more than 32 free lists.

For purposes of discussing time division of the total garbage collection effort, assume that the conservative collector uses a mark and sweep garbage collection technique. The total effort required to perform garbage collection consists of the effort required to mark and scan all live objects added with the effort required to sweep through the complete heap. An upper bound on the number of live objects is the sum of the number of objects that were live upon completion of the previous garbage collection pass and the number of objects that were allocated since completion of the previous garbage collection pass. At the moment garbage collection begins, the free lists contain a certain known amount of memory. Pace the allocation of this memory against the progress of the garbage collector, making sure, for example, that 30% of the garbage collection effort has completed prior to allocation of 30% of the remaining free pool.

Note that there is no worst-case bound on memory leaks that might be introduced by conservative scanning. Further, defragmentation of the heap is not possible. Thus, it would not be possible for a Java virtual machine that is using conservative garbage collection techniques to guarantee any lower bound on the amount of memory that will be available to particular real-time activities. Nevertheless, since conservative garbage collection has been demonstrated to perform well on average, expected and optimistic memory availability will be useful quantities.

Copying garbage collection. Copying garbage collection consists of periodically copying all live objects from one region of memory, called *from-space* to another equal-sized region of memory, called *to-space* [10]. If some of the objects residing in *from-space* are no longer live, the copied objects will not fill *to-space*. Thus, it is typically possible to allocate new objects in *to-space* while old live objects are being relocated, under the assumption that much of the current contents of *from-space* is dead. If all tasks that share access to the dynamic heap are trusted to be well behaved, this is a reasonable assumption. But if some of the tasks are unknown or untrusted, as is typical in Java execution environments, then it is somewhat risky to allocate new objects from *to-space* until after garbage collection has terminated.

One of the greatest benefits of copying garbage collection is that it fully defragments the free pool each time it completes a garbage collection pass. Thus, tasks that stay within their allocation budgets are guaranteed that memory will never be denied to them because of fragmentation problems. In fact, this is the only garbage collection technique that we are aware of which offers this guarantee to the client applications.

A disadvantage of copying garbage collection is that it imposes a high run-time overhead on execution of software. First, copying garbage collection is incompatible with conservative garbage collection techniques. Thus, extra code must be executed to maintain tags that distinguish pointers from non-pointers. Second, to coordinate the sharing of data structures between background garbage collection activities and ongoing execution of application software, it is necessary to execute several extra instructions each time the application software refers to a dynamically allocated object. In particular, each time a word is fetched from memory, its value must be examined to determine if it is a pointer to *from-space*. If so, the word is replaced with the corresponding *to-space* pointer before making its value available to the application software. Note that this requires the referenced object to be copied to *to-space* if it had not already been copied. This overhead has been measured to more than double the execution time of certain benchmark applications [11]. A third disadvantage of copying garbage collection is that it requires a minimum of twice as much memory as is actually accessible to the application at any instant of time.

Brook's optimization to copying garbage collection. This optimization is designed to reduce the execution-time penalty of copying garbage collection at the expense of one extra word per object [12]. The extra word serves as an indirection pointer. For the old copies of *to-space* objects residing in *from-space*, the indirection pointer refers to the corresponding *to-space* object. For a *from-space* object that has not yet been copied, the indirection pointer refers to itself. Similarly, for every *to-space* object, the indirection pointer refers back to itself. Every access to an object follows the indirection pointer to find the currently active version of the object. Meanwhile, garbage collection consists of sweeping through the objects copied into *to-space* and replacing all *from-space* pointers with the corresponding *to-space* addresses. Brook's optimization replaces the conditional range-checking test required by each memory read operation with a level of indirection associated with every read and write operation. In comparison with traditional non-real-time implementations of Lisp on Motorola 68000 processors, Brooks reports that the cost of fetching a pointer out of a dynamically allocated object takes 125% longer in the original copying algorithm and only 37.5% longer in his improved algorithm [12].

Accurate incremental mark and sweep garbage collection. Note that the incremental mark and sweep garbage collection technique described above is compatible with accurate garbage collection techniques as well as with conservative techniques. In comparison with conservative mark and sweep garbage collection, the benefit of accurate garbage collection is that memory leaks cannot be introduced by the conservative scanning process. However, there is a high cost associated with tagging of all pointers to enable accurate garbage collection. In comparison with copying garbage collection, the benefits include much higher utilization of memory and smaller run-time overhead; since live objects are not relocated, less effort is required to coordinate garbage collection with activation processing.

Mostly stationary real-time garbage collection. Mostly stationary garbage collection is a hybrid between copying and accurate mark and sweep garbage collection. The free pool is divided into *N* equal-sized demi-spaces. Two of the demi-spaces serve as *to-* and *from-space* respectively. The rest are collected using mark and sweep techniques. The benefit of this technique is that it offers memory utilization efficiencies close to that of mark and sweep techniques on average while still allowing memory to be defragmented in real time. Mark and sweep garbage collection is typically at least 50% more efficient than copying garbage collection [13]. In the worst case, the memory utilization of mostly stationary garbage collection is approximately the same as for the fully copying technique [13].

Real-time generational garbage collection. One straightforward technique for implementation of generational garbage collection is to adapt the mostly stationary garbage collection technique. Treat the fully copying region as a nursery, and treat the mark and sweep region as a second generation. Generational garbage collection performs well on average, but it is unable to find garbage residing in the second generation. Occasional full garbage collection passes are necessary, using the mostly stationary technique.

Note that generational garbage collection techniques do not improve worst-case latencies. Rather, they are intended to improve average case behavior at the cost of less predictable worst-case behavior. And there are many applications for which the assumptions on which the potential performance benefits of generational garbage collection depend are not valid. For example, Wade Hennessey, the principal scientist who oversaw garbage collection of ScriptX at Kaleida Laboratories, reported that in many of the multimedia titles he has studied, being able to quickly reclaim large amounts of recently discarded memory is much more important than improving average-case throughput [14]. According to Hennessey, it is quite common for multimedia applications to build over time relatively large data structures, and then to release the entire data structure in a single action. If parts of the data structure live long enough to be promoted into the older generations, which is quite likely, it would be difficult for a generational garbage collector to quickly reclaim the corresponding memory.

Hardware-assisted real-time garbage collection. Though there are numerous software-implemented real-time garbage collection techniques available, the benefits of adding a hardware accelerator to support garbage collection are very significant [15-17]. Hardware support consists of a special integrated circuit that sits between the system's level-two caches and memory. Some of the particular benefits of the hardware accelerator include:

1. Hardware support significantly reduces the run-time overhead required to coordinate garbage collection with ongoing application processing, including the cost of tagging memory to identify pointers. Depending on the garbage collection technique that is being implemented and the nature of the workload that is being measured, hardware support improves overall throughput by 30 – 50% or even more.
2. Hardware support shrinks the amount of memory required to reliably support particular workloads by 50% or more. This is a very important benefit, considering that memory is the single most expensive component of many embedded real-time computer systems. This benefit is made possible by efficient use of defragmenting garbage collection techniques and by the hardware accelerator's ability to parallelize much of the effort of garbage collection, making it possible to reclaim and recycle memory much more quickly than if all of the garbage collection has to be performed by the main CPU during "idle" times.
3. The hardware accelerator enables garbage collection primitive operations to be performed much more quickly. Whereas software garbage collection techniques typically offer worst-case execution latencies measured in tens of milliseconds, hardware-accelerated garbage collection offers worst-case latencies measured in microseconds.

Commercialization Opportunities

We are committed to commercialization of the Real-Time Java language standard described in this report. Once open standards have been established, we intend to develop embedded implementations of the standards for commercial sale to developers of embedded real-time systems. Our hope is to deliver a software implementation of Real-Time Java by 3rd quarter, 1996 and a hardware-accelerated implementation by 1st quarter 1997. The hardware accelerator will be available for purchase separately, either as a single chip, as a royalty license for use of the chip's Verilog description, or as a VLSI core[1].

We welcome opportunities to partner with other companies who might share our goal of supporting Real-Time Java as a development environment for creation of reliable, high performance, portable real-time software components. We are also seeking additional venture capital to help finance our development efforts as we strive to meet the announced product delivery dates.

We feel that the application domains for Real-Time Java represent a very important emerging market that is much larger than any single company can hope to dominate. We encourage others to become involved in filling some of the voids that might currently exist within this marketplace. Examples of potential opportunities for participation include development of:

1. Software development tools, such as graphical user interface generators.
2. Reusable component software libraries.
3. Real-time debugging and monitoring tools.
4. Application software for both clients and servers.
5. Infrastructure support to facilitate the development of real-time distributed applications including, for example, wireless telecommunication for in-vehicle navigation computers.

References

1. Sun Microsystems Inc., *The Java Language Environment: A White Paper.* 1995, Sun Microsystems, Inc.: Mountain View, CA.
2. Ritchey, T., *Java!* 1995, Indianapolis, Indiana: New Riders Publishing. 365.
3. Sun Microsystems Inc., *The Java Language Overview.* 1995, Sun Microsystems, Inc.: Mountain View, CA.
4. Wilner, D., Chief Technical Officer of Wind River Systems, *Personal Conversation*. 1995.

1. The hardware accelerator for real-time garbage collection is protected by four pending patents.

5. Nilsen, K. *Real-Time is No Longer a Small Specialized Niche*. in *Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*. 1995. Orcas Island, Washington: IEEE Computer Society Press.
6. Nilsen, K.D. and B. Rygg. *Worst-Case Execution Time Analysis on Modern Processors*. in *ACM SIGPLAN 1995 Workshop on Languages, Compilers, and Tools for Real-Time Systems*. 1995. San Diego, California: ACM SIGPLAN.
7. Basumallick, S. and K. Nilsen. *Cache Issues in Real-Time Systems*. in *ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*. 1994. Orlando, Florida: ACM.
8. Liu, C.L. and J.W. Layland, *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.* Journal of the ACM, 1973. **20**(1): p. 44 - 61.
9. Boehm, H.J. and M. Weiser, *Garbage Collection in an Uncooperative Environment.* Software - Practice and Experience, 1988. **18**(9): p. 807 - 820.
10. Baker, H.G., Jr., *List Processing in Real Time on a Serial Computer.* Communications of the ACM, 1978. **21**(4): p. 280 - 293.
11. Nilsen, K., *Garbage Collection of Strings and Linked Data Structures in Real Time.* Software - Practice and Experience, 1988. **18**(7): p. 613 - 640.
12. Brooks, R.A. *Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware*. in *ACM Symposium on LISP and Functional Programming*. 1984: ACM.
13. Nilsen, K. *Progress in Hardware-Assisted Real-Time Garbage Collection*. in *Lectures on Computer Science*. 1995. Kinross, Scotland: Springer-Verlag.
14. Hennessey, W., *Discussion at Kaleida Laboratories*. 1995.
15. Nilsen, K. and W. Schmidt, *A High-Performance Hardware-Assisted Real-Time Garbage Collection System.* Journal of Programming Languages, 1994. **2**(1): p. 1 - 40.
16. Nilsen, K., *Reliable Real-Time Garbage Collection of C++*. Computing Systems, 1994. **7**(4): p. 467-504.
17. Schmidt, W.J. and K. Nilsen. *Performance of a Hardware-Assisted Real-Time Garbage Collector.* in *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1994. San Jose, CA.