

Providing Soft Real-time QoS Guarantees for Java Threads

James C. Pang¹, Gholamali C. Shoja, and Eric G. Manning²
Department of Computer Science
University of Victoria, Victoria, BC, Canada, V8W 3P6
Jcpang@Redback.com, {Gshoja, Emanning}@csr.UVic.ca

Abstract

The Java platform has many characteristics that make it very desirable for integrated continuous media processing. Unfortunately, it lacks the necessary CPU resource management facility to support quality of service guarantees for soft real-time multimedia tasks. In this paper, we present our new Java Virtual Machine, Q-JVM, which brings CPU resource management to the Java platform. Q-JVM is based on Sun's JVM version 1.1.5. It implements an enhanced version of the MTR-LS algorithm in its thread scheduler. Combined with admission control that could be implemented in an application-level resource manager, it is able to support QoS parameters such as fairness, bandwidth partitioning and delay bound guarantees, as well as the cumulative service guarantee. Our test results show that Q-JVM is backward compatible with the standard JVM from Sun, has low scheduling overhead, and is able to provide QoS guarantees as specified.

1 Introduction

The many-fold increase in raw processing power of microprocessors and network bandwidth over the last decade has made possible a wide variety of new multimedia applications. These applications are capable of handling data that represent digital continuous media (CM), such as digital audio and video, using relatively inexpensive and commercially available computing hardware. Furthermore, processing of digital continuous media can now be integrated with conventional applications, such as word processing, on general purpose computing platforms.

Integrated continuous media processing poses unique challenges to the underlying support environment: it imposes real-time requirements on the host operating system and its subsystems, as continuous media data must be presented continuously in time at a predetermined rate in order to convey meaning. However, software

that processes digital CM data is often classified as being *soft* real-time; it only requires the operating system to statistically guarantee quality of service (QoS) parameters such as delay and throughput. There often are deadlines for various tasks; fortunately, missing a particular deadline is not fatal, as long as it is not missed by too much, and most other deadlines are not missed.

Multimedia computing is supported to varying degrees by a number of current generation operating environments. Among these, the Java platform [1] has many desirable characteristics. Java is a simple and small language. It is object-oriented and supports many language features, such as interfaces and automatic memory management, that make it a robust environment for software development. It also supports multithreaded programming at the language level with built-in synchronization primitives, thus allowing a high degree of interactivity with the end user. Moreover, Java has a rich collection of application programming interfaces which support media manipulation and continuous media processing. Most importantly, Java was designed for embedded applications, and is ideal for the multimedia devices of the near future.

Unfortunately, Java does not have the facility to support soft real-time processing. Real-time programming is a matter of managing resources, most noticeably the CPU resource. A real-time programmer must start with resource control, before building an application around that layer. However, Java does not provide any mechanism which can be used to monitor, manage, or police the usage of the CPU resource.

In this paper, we present our new Java virtual machine, Q-JVM, based on Sun's JVM version 1.1.5. The objective for this platform is to support integrated continuous media processing on mobile or embedded devices, such as PDAs and TV set-top boxes, where soft real-time guarantees must be provided with limited system resources.

Q-JVM employs an enhanced version of the Move-To-Rear List Scheduling algorithm. This algorithm is a general-purpose resource management algorithm developed at the Bell Laboratories for providing quality of service guarantees to soft real-time tasks [2]. We have enhanced it to handle not only user threads, but also system threads which must express their urgency using priorities.

Our enhanced version of the MTR-LS algorithm is implemented in the thread scheduler of the new Java virtual machine in place of the standard static priority-based scheduler. It enables Q-JVM to support QoS parameters such as fairness and bandwidth partitioning for soft real-time tasks.

Preliminary test results have shown that Q-JVM has low scheduling overhead, and is able to provide quality of service guarantees as specified. Moreover, it is binary compatible with the standard version distributed by Sun.

The rest of this paper is organized as follows. Section 2 discusses related research in resource and Quality of Service management. Section 3 details our implementation of a new Java virtual

-
1. This author was supported in part by Natural Sciences and Engineering Research Council (NSERC) of Canada, and SONY Corporation. He is currently employed at Redback Networks.
 2. Departments of Computer Science and Electrical & Computer Engineering.

machine that supports resource management. Section 4 documents results of some of our experimentation on Q-JVM. Finally, Section 5 summarizes this paper and offers some concluding remarks.

2 Previous Work

There has been a lot of research in CPU resource management for soft real-time applications. Some researchers have studied existing systems that claim to support real-time multimedia applications, and found that static priority-based scheduling is not sufficient for multimedia soft real-time applications [8]. Others like [5] have borrowed from link scheduling, or have proposed new algorithms [2] for managing the CPU resource. At the same time, researchers have realized the potential of the Java platform for embedded real-time processing, and have proposed extensions to the base platform [9].

2.1 Static Priority Based Scheduling

The most common CPU resource management schemes employ static priority-based scheduling. In such a scheme, all execution entities are assigned fixed priorities. The scheduler rations the CPU to competing entities according to their priority.

UNIX System V Release 4 (SVR4) incorporates such a static priority based process scheduler. By incorporating this scheduler, SVR4 claims to provide system support for real-time and multimedia applications. However, an extensive quantitative analysis of this process scheduler, conducted by Nieh et. al. [8], demonstrated that this process scheduler was largely ineffective. It could even produce system lockup. Their conclusion was that a static priority based real-time process scheduler in no way allows a user to deal with the problem of CPU resource contention presented by multimedia applications.

Similarly, hard real-time scheduling algorithms, such as Earliest Deadline First and the Rate Monotonic Algorithm, are not suitable for integrated continuous media processing [8]. This class of algorithms either fails to achieve the desired efficiency for integrated computing environment, or requires prior analysis of computational requirements of the particular application mix. The latter is difficult, if not impossible, for dynamic systems.

2.2 Resource Management Based on Fair Queuing

A large body of work exists on fair queuing. The Start-time Fair Queuing (SFQ) algorithm [5][6] is a notable example.

SFQ improves upon early fair queuing algorithms like Weighted Fair Queuing [3] and Self Clocked Fair Queuing [4] by removing their requirement for prior knowledge of the computational needs of competing tasks; it is also much more efficient than algorithms like Fair Queuing based on Start time [6]. Moreover, it handles fluctuation in available bandwidth due to sporadic interrupt processing better than other fair queuing algorithms [5].

SFQ was first developed for network packet scheduling in [6], and was successfully adopted for CPU scheduling later in [5]. Unfortunately, its delay bound increases linearly with the number of threads in the system [10], thus making it undesirable for complex and dynamic systems.

2.3 The MTR-LS Algorithm

The Move-To-Rear List Scheduling algorithm (MTR-LS) is a new resource scheduling algorithm developed at the Bell Laboratories. It is aimed to provide predictable service in a general purpose system with multiple resources including CPU, disk, and a network [2]. Besides the usual quality of service parameters such as fairness, bandwidth partitioning and delay bound, it also supports a new criterion called *cumulative service guarantee*: MTR-LS guarantees that the real service obtained by a process, given its specified service rate, on a shared server does not fall behind the ideal service it would have accumulated on a dedicated server at the same service rate by more than a constant amount.

Scheduling using MTR-LS is based on *service fractions*. A service fraction is a fraction assigned to a scheduling entity that represents the service rate of a virtual server in terms of the real server. A system constant, named the *virtual time quantum* T , is used to specify the total target time of servicing each and every active scheduling entity in the system exactly once.

Each scheduling entity is also assigned a time stamp and a quantum *left* when it requests service. The quantum size is calculated as the product of its service fraction and T . All active scheduling entities are kept in the *service list* L , and are sorted by their time stamps. Entities with earlier time stamps appear closer to the front of the list. The MTR-LS algorithm always schedules the first runnable entity on the service list.

A scheduled entity is preempted when its quantum is consumed, or when some other entity whose position on the service list is ahead of it becomes runnable. After the preemption, the time it has been serviced on the server is subtracted from its quantum *left* to yield an updated value of *left*. If the result is zero, then it is assigned a new time stamp and its quantum is re-initialized. Its position on the service list is adjusted according to its new time stamp; i.e., it is moved to the rear of the list.

The MTR-LS algorithm provides bandwidth partitioning and fairness guarantees for all competing entities with respect to their service fraction allocations. With admission control, it is also able to provide delay bounds and cumulative service guarantees [2]. Moreover, it is very efficient: even with a straightforward implementation, the computational complexity of the algorithm is $O(\ln(n))$ where n is the number of entries in the service list [2]. These properties persuaded us to build on the MTR-LS algorithm to provide quality of service guarantees for Java threads.

2.4 Real-time Extensions to Java

Other researchers have also realized the potential of the Java platform, and considered extensions to this environment for real-time computing [9]. The extension proposed by Nilsen uses a number of techniques such as analysis of worst-case execution time, measurement of representative function invocations, rate monotonic analysis, static cyclic scheduling, and real-time garbage collection with possible hardware assistance [9]. The proposed scheme is designed to satisfy hard real-time constraints, and is fairly sophisticated. Unfortunately, it is not compatible with the standard Java platform available from Sun.

To serve multimedia applications, however, it is sufficient for a system to provide only statistical guarantees. Soft real-time scheduling algorithms such as the MTR-LS algorithm are thus better suited to this class of integrated computing system. We recognize

that automatic garbage collection is a major stumbling block for providing real-time service in Java, as it is difficult to regulate its resource consumption. Nevertheless, we prefer to reduce this to a resource management problem, and leave the construction of a better garbage collector to other specialists.

3 A New Java Virtual Machine

Our enhanced version of the Java virtual machine is based on version 1.1.5 of Sun's reference implementation. Although the source code obtained from Sun supports both Windows and Solaris, we chose to base our virtual machine on Solaris. As our implementation makes few assumptions about the support provided by the underlying operating system, it is relatively easy to port to other platforms.

Java threads may be mapped to native operating system scheduling entities using One-to-One, Many-to-One or Many-to-Many models[7]. With a 1-to-1 mapping, each Java thread is supported by its own scheduling entity known to the operating system. Scheduling is handled by the OS; all threads have equal access to the kernel at the same time. Thus, this model is able to exploit any hardware parallelism that may be available.

With the *m-to-1* model, all Java threads are mapped onto a single scheduling entity supported by the OS; and scheduling of Java threads is handled by a user-level threads library. Only one scheduling entity is known to the operating system, and only one thread can access the kernel at any given time. This model does not exploit hardware parallelism; however, it has the advantage that the OS kernel is not required to support multithreading. Moreover, it is also very efficient, as all scheduling decisions and context switches can be handled in user space, without kernel intervention. This is a considerable advantage in uni-processor environments, where the additional heavy context switches to and from the kernel does not yield any benefit in terms of increased parallelism. It is thus ideal for mobile and embedded devices that do not have multiple processors, and where CPU bandwidth is at a premium.

The *m-to-n* mapping model is the most elaborate. It uses a user-level threads library in conjunction with a OS kernel that supports multi-threading: Java threads are mapped onto a pool of scheduling entities known to the kernel. The threads library manages the pool of scheduling resources and the mapping between Java threads and kernel threads, while the kernel schedules only the entities known to it.

On Solaris, one has the option of using either the Many-to-Many model with the Solaris native thread library, or the Many-to-One model with the Green Thread library. Since it is not common to have true hardware parallelism on personal computing platforms we target, we decided to base our changes on Green Thread using the *m-to-1* model.

Our approach was to build the resource management facility into the lowest level of the Java virtual machine, in this case, the Green Thread library, so that resource consumption by all threads, including the threads spawned by the JVM and its native libraries, can be managed effectively.

3.1 Green Thread

Green Thread is a traditional priority-based threads package. It relies on three system threads for its operation; and it uses a stateless scheduler function which runs on the preempted thread's stack.

It is also tightly integrated with the Java virtual machine.

Scheduling in Green Thread is based on priority. Priorities are represented by integer values, where larger integers represent higher priority. Although the basic architecture does not limit the range of priorities, user threads use only ten values: integers from 10 to 1. Java threads use only these priority values as well

There are three system threads in Green Thread that use priorities outside of the range for user threads. These three threads are the *ClockHandler* thread, the *TimeSlicer* thread, and the *Idler* thread (Figure 1).

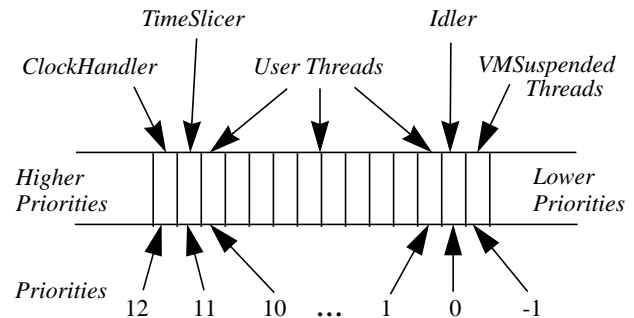


Figure 1: Priorities of Green Threads

The *ClockHandler* thread is responsible for maintaining alarms for all other threads. It lies dormant most of the time and wakes up only when an alarm expires or when one is registered or removed. When it wakes up, it notifies all threads whose alarms have expired. It then scans all the currently active alarms to calculate the next time out period. After registering an alarm with the operating system to deliver a signal to it after the time out period, it suspends itself again. Running at priority 12, the *ClockHandler* thread is the highest priority thread in the system.

The *TimeSlicer* thread runs at priority 11. Like the *ClockHandler* thread, it lies dormant most of the time. When it wakes up, it registers an alarm that expires at the end of the next preemption interval, and then goes into sleep again. Since it has a higher priority than any other threads in the system, except the *ClockHandler* thread, it will be scheduled as soon as it is runnable, i.e., when its alarm expires. By waking up and going into sleep again, it preempts the current user thread, and allows the scheduler to schedule a new thread according to its scheduling policy.

The use of this time slicing mechanism is optional in Sun's implementation of Green Thread for the Java virtual machine. By default, this thread is not loaded. In this case, threads may be blocked only when they perform some system operations such as performing I/O or attempting to enter a monitor. An operator can enable time slicing and set the quantum size via command line switches to the JVM.

Running at priority 0, the *Idler* thread is the lowest priority thread in the system. Hence, it is scheduled only when there are no other runnable threads. Whenever it is scheduled, the *Idler* thread will reclaim memory occupied by the stacks of terminated threads, and then yield the CPU to the operating system.

In addition to the three system threads, Green Thread also maintains two other user threads in its system space: the *GC* thread, and the *Finalizer* thread. The *GC* thread runs the garbage collection routines in the JVM, while the *Finalizer* thread runs the `finalize()` routines of discarded Java objects. Both of these two

threads run at the lowest priority of user threads, priority 1. However, when the *GC* thread runs, it runs to completion and thus is non-preemptable. Furthermore, when a low memory situation is detected, the Java virtual machine may suspend all user threads and run the garbage collection routines on its own behalf.

Green Thread manages threads using priority queues that are implemented as linked lists. All runnable threads are kept in the *runnable queue*. Java monitors also use queues to manage threads. Every monitor has a *wait queue* for threads that are waiting to enter it, and a *condition variable wait queue* for threads that are waiting for some conditions to become true. Threads in these queues are sorted in the order of their priorities. In addition, there is an *active queue* that links together all threads that have been created but not yet terminated. This queue is not sorted in any particular order.

Green Thread's scheduler is a function that runs on the stack of the last scheduled thread. It is invoked by the context switching code every time the current thread yields or is blocked or preempted. Green Thread's scheduling policy is implemented in the queues. Higher priority threads are inserted into the front of the queues before all lower priority threads; equal priority threads are inserted in the order of their arrival. When invoked, the scheduler function always schedules the first thread it finds on the runnable queue. As the *Idler* thread never blocks, the scheduler is always able to find at least one thread to schedule.

Each Green thread also has a block of private information that helps to facilitate its management. This includes its priority, its state, the lists of monitors it has entered or is waiting on, and information about its stack memory and machine context.

3.2 Extensions to Green Thread

Our purpose is to support soft real-time scheduling of Green threads (and in turn, Java threads) through the addition of a resource management algorithm to the Green Thread library. In our final implementation, Green Thread's original system threads and the monitor infrastructure were largely unchanged. However, the thread private data structure, the queues and the scheduler function were extended to accommodate the new scheduling policy. The pre-emption and context switching mechanisms were also modified to track CPU resource consumption by individual threads.

The first change is to extend the thread data structure to include fields for a time stamp, a service fraction specification, and the time left in a thread's current quantum, i.e., the *left* value. The service fraction is specified by the user, and may be changed at any time. It is used to calculate a thread's quantum, in conjunction with the virtual time quantum. The time stamp is used to determine the position of a thread in the service list *L*: threads having earlier time stamps appear at nearer the head of *L* than threads with later ones. When a thread finishes a quantum, it is assigned a new (later) time stamp and thus moved to the rear of the list.

However, we implemented the time stamp not as a reading of the clock, but as a 64 bit long integer, where larger integer values represent earlier time. The earliest time stamp is the largest 64 bit positive integer value. When a thread is assigned a new time stamp, it is given a 64 bit integer that is smaller than all assigned time stamp values in the system.

This decision stems from the realization that there is a parallel between time stamps and priorities: threads with earlier time stamps are closer to the front of the service list than threads with

later time stamps; they need to be serviced before other threads. In effect, these threads have a higher effective priority than other threads. In the original Green Thread library, larger integer values represent higher priority. Using a monotonically decreasing integer in place of a real time stamp permits one to re-use most of the Green Thread library, which assume a priority-based threads model, with minimum modification.

The key to implementing the MTR-LS algorithm is the service list *L*. It is supposed to be an ordered list of all active threads in the system, sorted by their time stamps. However, the order of threads is significant only when they are scheduled. Therefore, our implementation only keeps the *runnable queue* sorted; the *active queue* is left unsorted.

The new runnable queue is a priority queue with the time stamp as priority. The original runnable queue in Green Thread is implemented as a linked list. Such a data structure is quite inefficient for priority queues. We developed a new priority queue based on the heap data structure. The time complexity of this queue is in the order of $O(\ln(n))$. This new queue abstract data type (ADT) enabled us to realize the full potential of the MTR-LS scheduling algorithm in terms of efficiency. Other parts of the Green Thread library, such as the monitor code which uses priority queues to manage threads, are also modified accordingly to take advantage of this ADT.

A pair of in-line functions is added to Green Thread's context switching code to monitor resource consumption by individual threads. The first in-line function is inserted just before the place where control is transferred to a newly scheduled thread. This function saves the system hardware clock reading in a variable private to the scheduler. The second in-line function is inserted immediately following the place where context is switched away from a thread. It takes another reading of the system clock, and subtracts from it the last reading of clock taken when the current thread was scheduled. The difference is the time that the last scheduled thread has run on the CPU. This difference is then subtracted from the *left* value of the last scheduled thread.

The context switching code is executed whenever control of the CPU changes hands. Thus, this pair of in-line functions is able to track CPU usage of all threads managed by the Green Thread library. However, the context switching code itself does consume CPU resource that is not fully accounted for. Fortunately, these routines are simple and short; and their invocations are statistically predictable. Hence, their resource consumption is expected to be a constant but negligible amount. The small discrepancy can be dealt with by reserving a small portion of the CPU that is not allocated to any threads.

The basic scheduler function from the original Green Thread library is also modified to implement the MTR-LS algorithm. It still runs on the stack of the last scheduled thread; it is still invoked by the context switching code every time the current thread yields or is blocked or preempted; however, it now assumes extra duties.

When invoked, the scheduler fetches the thread with the earliest time stamp from the runnable queue and checks its *left* value. If this value is less than or equal to zero, it moves this thread to the rear of the list by assigning it a new time stamp and reinserting it into the runnable queue. The *left* of this thread is re-initiated to be $\alpha \cdot T + left$, where α is the service fraction of this thread, *T* is the virtual time quantum, and *left* is this thread's last *left* value. The

scheduler function repeats this operation until it finds a thread with a positive *left* value. It then invokes the context switching code to record the start time of this quantum and transfer the control of CPU to the scheduled thread.

Finally, time slicing is enabled by default in the new JVM; however, the mechanism is modified to accommodate the new scheduling algorithm. The *TimeSlicer* thread is now loaded automatically during system initialization and enters suspension immediately. When the scheduler runs, it sets an alarm for *TimeSlicer* that expires after Δt time units, where Δt is the smaller of a pre-emption interval and the *left* value of the thread being scheduled. When this alarm expires, *TimeSlicer* becomes runnable. It will preempt the current thread, and cause a rescheduling. If the current thread is blocked for other reasons before this timer expires, the scheduler will cancel the alarm and set a new one when the next thread is scheduled.

3.3 Extension to the MTR-LS Algorithm

The MTR-LS algorithm is used in our new Java virtual machine to schedule not only user threads but also the system threads that must express their urgency using priorities. For example, the thread scheduler must recognize that the *ClockHandler* thread must be scheduled as soon as it is runnable and the *idler* thread should only be scheduled when there are no other runnable threads.

Unfortunately, the MTR-LS algorithm is based on service fractions and CPU resource consumption. It does not have an inherent notion of urgency as may be expressed with priorities.

Our solution is to extend the MTR-LS algorithm to handle the system threads as a special case. We took advantage of our earlier realization that MTR-LS schedules threads according to their positions on the service list *L*, and that the time stamps which determine the positions of threads on *L* can serve as effective priorities.

Under the MTR-LS algorithm, a thread will not be scheduled if there are other threads ahead of it on the service list. Hence, if high priority system threads are placed at the beginning of the list, i.e., ahead of all user threads, then as soon as they become runnable they will be scheduled. Similarly, if the low priority system threads are placed at the rear of the list, i.e., after all user threads, then they will not be scheduled until there are no other threads that are runnable.

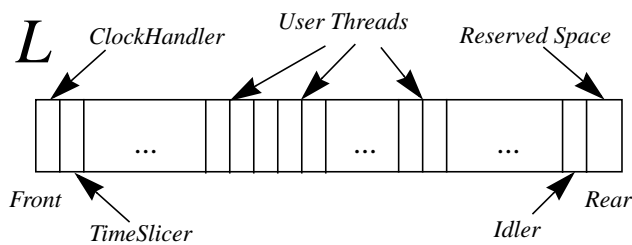


Figure 2: Positions of the System Threads and User Threads on the Service List *L*.

A map of positions of the system threads and user threads on the service list *L* is shown in Figure 2. The *ClockHandler* thread and the *TimeSlicer* thread are assigned the two earliest (largest) time stamps. This puts them at the beginning of the service list. At the same time, the *idler* thread is assigned an artificially late (small) time stamp which puts it near the end of the service list¹.

Moreover, all system threads are tagged so that when context is switched away from them, their *left* values are not updated. As a result, they will never be moved to the rear of the list. In other words, their positions on *L* are stationary.

This approach takes full advantage of the natural ordering of threads on the service list and the parallel between time stamps and priorities. It provides a simple and straightforward solution for the problem of using the MTR-LS algorithm in a root scheduler which must allow system threads to express their urgency.

We observe that the system threads do consume CPU resource, but their resource consumption can be bounded. Both high priority system threads are sporadic; their invocations are statistically predictable. Moreover, as they perform very simple and dedicated functions, their resource consumption is expected to be small and remain statistically constant. To account for this amount, the allocated CPU bandwidth counter can be set to a very small number, e.g. 1%, when the thread library initializes, before any other threads are created.

The resource consumption by the other system thread, *Idler*, is negligible. This thread is only scheduled when there are no other runnable threads, i.e. when there is no resource contention. It is pre-empted as soon as another thread requests access to the CPU.

Finally we note that Java VM threads such as the *Finalizer* thread and the *GC* thread are treated as user threads. They are subjected to the same resource consumption accounting as any regular user threads. They are assigned small service fractions (e.g. 1% and 5%, respectively) that may be changed by a user-level resource manager to suit application requirements.

3.4 Resource Consumption Accounting

An interesting problem arises when our new JVM is executed on top of a multi-programming environment such as Solaris. The operating system may preempt the Java VM runtime environment and give the CPU to another program. This could cause resource accounting to be disrupted.

For example, assume at time *t* a thread *T* is scheduled, and at time $t + \Delta t_1$ the JVM is preempted by the operating system. Green Thread's context switching mechanism is not invoked as this preemption is transparent to the user program. At time $t + \Delta t_1 + \Delta t_2$, the JVM regains control of the CPU and *T* continues its execution. At time $t + \Delta t_1 + \Delta t_2 + \Delta t_3$, thread *T* is pre-empted by Green Thread. The resource accounting mechanism would record that *T* has executed for $\Delta t_1 + \Delta t_2 + \Delta t_3$ time units. However, because of the preemption by the operating system, *T* has only had $\Delta t_1 + \Delta t_3$ time units of CPU time. The other part, Δt_2 , was taken away by the operating system to execute system functions and other user programs.

Our experiments on Solaris have shown that the effect of such operating system preemption is significant even when the system is sparsely loaded with only the standard mix of daemons.

1. The last few positions on *L* are reserved for implementing a JVM system function which suspends all user threads before the garbage collector is invoked. The original implementation sets the priorities of all user threads to -1. Our implementation moves the user threads to a position on *L* that is behind the *idler* thread.

The solution to this problem is found partially outside the scope of our modification to the Java VM. Most modern operating systems like Solaris and Windows NT have a so called “real-time” scheduling facility. It is designed for programs that require maximum control over their own scheduling. Using such a facility, a user program may be arranged to have a higher priority than any other tasks in the system including the operating system kernel; it will preempt even system activities such as paging when it becomes runnable. This is a dangerous facility to use, as not leaving enough CPU time for system tasks will result in their starvation and produce system deadlocks.

Our solution is to take advantage of this facility but at the same time create a new VM thread, named the *OS thread*, that does nothing but yield the CPU to the operating system. This thread is just another user thread, and is allocated a service fraction. The runtime environment is thus able to regulate its CPU resource consumption, and in turn, control the CPU bandwidth consumed by the operating system and other concurrent user programs. The service fraction allotment for the *OS thread* can be set via a command line switch to the Java virtual machine. It is adjustable according to system load using a resource management API.

3.5 High Level API

The enhanced capabilities of the new Java virtual machine are made available to the application layer via a high level application programming interface. This API is encapsulated into two Java classes: the `QThread` class and the `QThreadGroup` class. These two classes supersede the `java.lang.Thread` class and the `java.lang.ThreadGroup` class, respectively.

The `QThread` class provides resource and, in turn, quality of service management to Java threads. It supports all normal threads operations such as thread creation, termination and communication. At the same time, it supports specification of resource requirements in terms of service fraction reservations.

Security for thread control can be implemented using the standard Java convention: an optional “Security Manager” may be installed to regulate access to particular threads. The default Security Manager allows access within the same thread group.

The `QThreadGroup` class provides support for CPU bandwidth partitioning for groups of threads. It is structured as a subclass of the standard Java thread group class `java.lang.ThreadGroup`. It augments the original thread group class with resource management capabilities.

The `QThreadGroup` class inherits all methods of its superclass. It is able to support most management functions found on the original Java platform. However, priority manipulation methods of the standard Java `ThreadGroup` class are overwritten. In their place, `QThreadGroup` provides methods for CPU bandwidth partitioning.

The `QThreadGroup` class implements a simple admission control policy: the aggregate bandwidth allotment of all child threads of a `QThreadGroup` must not exceed the allotment to the `QThreadGroup` itself. However, as we do not wish to impose any resource management policies on user applications, the use of this facility is optional. In other words, a `QThread` is not forced to be a member of any `QThreadGroup`. It would be, of course, a member of some instances of `java.lang.ThreadGroup`.

Furthermore, both the `java.lang.Thread` class and the

`java.lang.ThreadGroup` class are re-implemented to provide compatibility for existing Java programs. The `Thread` class is implemented as a subclass of `QThread`. The thread control methods are mapped directly. However, the thread creation and priority manipulation methods are mapped to the thread creation and service fraction manipulation methods of the `QThread` class with a simple formula to convert priorities to service fractions. The `ThreadGroup` class remains unchanged for the most parts. Only its priority manipulation methods are modified to return some suitable defaults.

4 Experimental Results

To verify the viability of Q-JVM, we developed a test suite to measure its performance. Tests were designed to instrument the effect of using Solaris’ “real-time” scheduling facility and the scheduling overhead of the new JVM. We showed that the new platform is able to provide predictable resource allocation and resource partitioning. A similar test suite was executed on an un-modified Java VM with Green Thread; the results were compared to those gathered on the enhanced platform. A number of standard test suites are also executed without modifications on the enhanced Java VM to verify its compatibility with the standard JVM available from Sun.

4.1 Test Setup

Performance evaluation was done on a Sun SPARCserver 20 with a 60MHz Ultra-SPARC CPU and 64 MB of main memory running Solaris 2.4. All experiments were conducted in multi-user mode with the standard complement of daemons like *lpd*, *sendmail*, *NFS*, and a very lightly loaded *HTTP* server. Moreover, all experiments were conducted when there was no interactive user activities.

Most of our experiments were carried out using the *RaceTest* test suite. This test suite is a multithreaded Java program that simulates a CPU intensive application. It is modeled after the well known *Dhrystone* benchmark. When the application starts, it spawns a number of threads, called *runners*, each of which executes an arithmetic calculation repeatedly in a loop. The number of loops completed in a given time by one or more *runners* is used as the performance metric.

Two versions of the test suite were constructed. The first one runs on the enhanced JVM and allows an operator to specify the number of *runners* to start and the service fraction for each *runner*. The other version runs on the standard JVM. Instead of service fractions, it allows an operator to specify priorities for the *runners*. Otherwise, the two versions are identical. A standard Java virtual machine was built from un-modified JVM source from Sun to run the standard version of *RaceTest*. It was configured to use Green Thread as well.

Besides the basic test application, a shell script named *Gen-Load* was developed to generate a greedy system load for simulation of CPU resource contention in a general purpose computing environment. This script uses *tar* and *gzip* to archive a large directory tree repeatedly. When executed alone, it results in a steady system load average around 1 on our SPARC server test hardware. This script is executed when we need to simulate a busy system for the experiments.

4.2 The Baseline Performance

Before testing our enhanced JVM, we first ran the test suite on the standard platform to establish baseline performance. In order to create an environment that is closest to the new platform, the standard Java virtual machine was started with time slicing enabled and a quantum size of 20 milliseconds. The performance numbers are listed in Table 1.

	1 Runner	4 Runners	10 Runners
Aggregate Average Throughput (loops per second)	602,797 (L.L.)	602,084 (L.L.)	601,040 (L.L.)
	268,221 (H.L.)	303,672 (H.L.)	333,136 (H.L.)
Average Standard Deviation (% of throughput)	2.52% (L.L.)	22.05% (L.L.)	26.04% (L.L.)
	10.53% (H.L.)	27.70% (H.L.)	33.97% (H.L.)

Table 1: The Baseline Performance

The *RaceTest* application was run with one, four, and ten *runner* threads. All *runners* were started at priority 5. The first row in Table 1 reports the aggregate average throughput of all *runners* in loops completed per second. This figure is arrived at as follows. One thousand throughput samples are taken for each *runner*. If there is only one thread, the average is reported. When there are more than one *runner*, the sum of their averages is reported.

The second row in Table 1 measures how much variance there is in the throughput data. When there is only one thread, the standard deviation of the throughput samples is calculated; it is reported as a percentage of the average throughput. When there are more than one *runner*, the average of the standard deviation figures from all *runners* is reported. This figure gives an indication of how much jitter a thread experiences. Small values represent less jitter and better quality of service.

Two figures are reported in each cell in Table 1. The first number, tagged by (L.L.), is the result of running the test suite on a *lightly loaded* system with only the standard complement of daemon processes but no interactive user activity. The other number, tagged by (H.L.), is the result of running the test suite on a *heavily loaded* system. The extra system load is generated by *GenLoad*.

From the data given in Table 1, we observe the following. When the system is lightly loaded, the aggregate average throughput decreases with the increase of the number of *runners*. This may be attributed to the scheduling overhead of the JVM. However, under heavy load, when there is competition for the CPU resource from other activities in the system, the aggregate average throughput increases with the number of *runners*. This phenomenon is attributed to the fact that the JVM is competing more aggressively with other applications when it has more threads activity.

Finally we note that the jitter increased significantly, from 2.52% to 10.53% in the case where there is only one *runner*, when another application started to compete for CPU with the JVM. This is expected, as the time slicing mechanism is affected by the multi-programming operating system. The lower figure, 2.52%, may be viewed as the *noise level*, as there is no resource contention in this scenario. It is also apparent from data in Table 1 that the jitter increases with the number of threads.

4.3 Effect of the RT Scheduling Class

We have discussed in Section 3.4 that a multi-programming operating system may steal CPU cycles from our resource manager. To remedy this problem, we suggested that the *real-time* scheduling facility available on Solaris should be utilized. This facility permits Q-JVM to run without interruption until it is ready to release the CPU. This will result in more accurate resource accounting and better quality of service. We designed a series of tests using the *RaceTest* suite to experimentally validate this claim.

	Average Throughput (loops per second)	Standard Deviation (% of throughput)
Lightly Loaded System, TS class	601,582	2.76%
Busy System TS Class	315,971	16.44%
Lightly Loaded System, RT class	521,247	1.84%
Busy System, RT class	507,638	2.65%

Table 2: Effect of the RT Scheduling Class

First, we ran the test suite on a lightly loaded system in the *time sharing*(TS) class. Then, we loaded the test application into the *real-time*(RT) class with a service fraction of 15% allocated to the *OS thread*. Next, we started the *GenLoad* script and ran the test suite again in the TS class. Finally, with the *GenLoad* script still running, we loaded the *RaceTest* application into the RT scheduling class with 15% of the CPU assigned to the *OS thread*. In all cases, only one *runner* was started in *RaceTest*, with a service fraction assignment equal to 80% of total CPU bandwidth.

The results are given in Table 2. We can observe that the enhanced JVM running in the RT scheduling class on a busy system achieves approximately 85% of throughput compared to when it is running in the TS class on a lightly loaded system. Moreover, it provides better quality of service (less jitter) when running in the RT class, even under heavy load.

4.4 Scheduling Overhead

A major concern in using a complex resource management scheme, such as the one built into our enhanced Java virtual machine, is that the scheduling overhead may be high. To evaluate this overhead, we compared the aggregate average throughput achieved by the *RaceTest* application on the enhanced platform to the standard JVM. The results are presented in Table 3 on the next page.

In order to ensure that the collected data are directly comparable, both the enhanced platform and the standard JVM are started under the *time-sharing* scheduling class. All tests are done on a lightly loaded system with no interactive user activity.

From Table 3, we can observe that the throughput differences among all figures are very small: the throughput of the application running on Q-JVM is only 0.19% to 0.36% lower than the same

application running on the standard JVM. As the application is CPU intensive, and does not involve any I/O or kernel service call, this result indicates that the additional scheduling overhead of Q-JVM is very small compared to the standard Java virtual machine.

	1 Runner	4 Runners	10 Runners
Standard JVM	602,797	602,084	601,040
Q-JVM Configuration1	601,399	599,860	599,714
Q-JVM Configuration2	601,582	600,939	599,882

Table 3: Effect of Scheduling Overhead on Thread Throughput
All reported figures are *Aggregate Average Throughput* (in loops per second). On the standard JVM, all threads (*runners*) are started with priority 5. On the Q-JVM, all threads are assigned 1.5% of total CPU bandwidth in configuration 1. In configuration 2, all threads are assigned equal service fractions totaling 80%. For example, when there are 10 threads, each is assigned a service fraction of 8%.

We can further observe the following: 1) The scheduling overhead increases with the number of active threads, although the increase is not significant. 2) The scheduling overhead is smaller when the total allocated service fraction is closer to 100%; however, there is a point of diminishing return where this saving in overhead is eventually overpowered by the increase in overhead caused by having more threads to schedule.

4.5 Predictable Resource Allocation

We have argued that one major benefit of Q-JVM will be predictable resource allocation. In particular, it will give equal access to the CPU to threads with equal service fraction assignments. This is not possible on the standard platform: there is no provision to specify CPU resource allocation. One may assign equal priorities to competing threads and hope that they gain equal access. We predicted that one will still see a degree of unfairness in such an arrangement.

To validate this claim, we compared average throughput of 10 *runner* threads on both Q-JVM and the standard JVM. Each *runner* is allocated 9% of the CPU on Q-JVM. They are also assigned equal priorities (5) when running on the standard JVM. The standard JVM is started with time slicing enabled, with a quantum size of 20 milliseconds. In order to get comparable results, all tests are conducted in the *time sharing* class of Solaris.

The results are presented in Figure 3. It is evident that the throughput which the *runners* are able to achieve is more uniform on the enhanced JVM than on the standard one. Upon close examination, we find that the differences of throughput among *runners* on the new JVM are all within 0.1% of the average. However, the differences on the standard platform are around 1% of the average. This is equal to one decimal order of magnitude improvement in predictability and fairness.

4.6 Resource Partitioning

Another major benefit of the new JVM is its support for resource

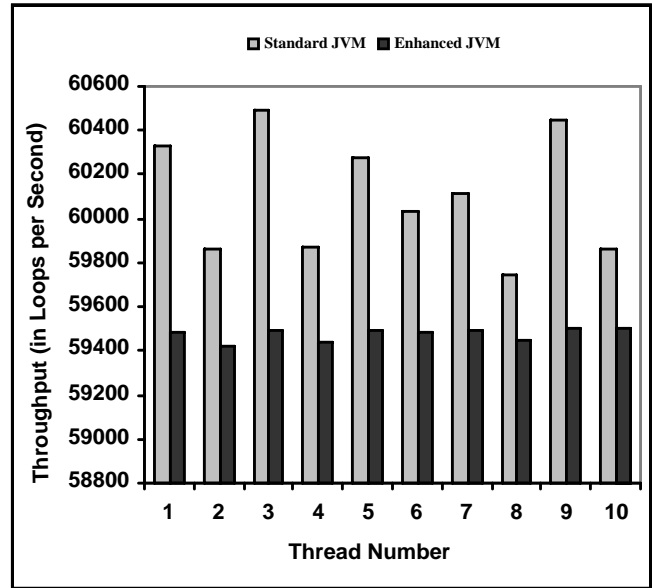


Figure 3: Fairness comparison between the standard JVM with time slicing and the enhanced JVM

partitioning. We again set up the *RaceTest* suite to verify that one is able to assign portions of CPU bandwidth to specific threads, and have the underlying virtual machine enforce the allocation.

The experiment is conducted on Q-JVM running in the *time sharing* class of Solaris. Two *runners* are started on each run; their service fraction assignments are varied each time. The experiments successfully demonstrated that the throughput achieved by each *runner* satisfies its service fraction allotment as we argue below. Data from one of the test runs is presented in Table 4.

	Service Fraction Assignment	Average Throughput (loops / second)
Runner 1	70%	442,728
Runner 2	20%	158,069

Table 4: Resource Partitioning

In this case, the two *runners* are assigned service fractions of 70% and 20% of total CPU bandwidth, respectively. If we take the throughput of the standard JVM running a single *runner* on a lightly-loaded system, as reported in row 2, column 2 of Table 1, to be a close approximation to the full capacity of the system, then the two *runners* in this test case have achieved 73% and 26% of the maximum throughput, respectively. This is evidence that both *runners* received more than sufficient bandwidth to satisfy their respective reservations.

4.7 Compatibility

In addition to the quantitative tests described in the previous subsections, we also ran a number of standard Java applications without modification on the new virtual machine.

First, we ran the *CaffeineMark* suite on the new VM. *CaffeineMark* is a popular benchmark suite for the Java platform. It analyses

Java system performance in areas like integer and floating point calculations, loops, logic operations, string manipulation, method invocations, graphics operations, and common GUI operations. It gives an indication of the overall performance of a Java platform. Being able to run it successfully also indicates, to a degree, the compatibility of the new platform with respect to the standard one. Our new Java VM is able to complete this benchmark suite without any incident. The results are given in Table 5.

	Q-JVM (TS Class)	Q-JVM (RT Class)	Standard JVM
Scores	137	148	144

Table 5: *CaffeineMark* Scores

We then successfully tested the *HotJava* browser on Q-JVM. *HotJava* is a full featured web browser developed by Sun. It is a complex application and is 100% pure Java. Being able to run it without incident is a strong indication that Q-JVM is binary compatible with the standard JVM. We tested *HotJava* version 1.1.4 extensively with no ill effect.

Finally, we tested Q-JVM using the Java Media Framework (JMF) version 1.0 from Sun. This test was not exhaustive; it was targeted at verifying compatibility with the standard platform. We were able to run all the sample applications shipped with the JMF package. Furthermore, we were able to use the included media player applet to playback pre-recorded movie and sound tracks in various formats with satisfactory performance.

5 Summary

In this paper, we have presented our work on supporting soft real-time tasks on the Java platform. We developed a new Java virtual machine that is able to support resource allocation and consumption regulation for the CPU resource. This in turn provides quality of service (QoS) guarantees for Java threads.

We extended a service fraction-based resource management algorithm, the Move-to-Rear List-Scheduling algorithm developed at Bell Laboratories, to handle system threads that must express their urgency, which is normally expressed using priorities. We then incorporated this algorithm into a new Java virtual machine based on the source code of JVM version 1.1.5 licensed from Sun.

The result of our work is a new Java platform that is able to support soft real-time tasks. It provides mechanisms for resource allocation and management, as well as guarantees for a number of Quality of Service parameters like fairness and bandwidth partitioning.

Preliminary test results show that our scheme for resource accounting and management is viable and the new Java VM is indeed able to provide these QoS guarantees as specified. Moreover, its scheduling overhead is very small comparing to that of the standard version. Yet, our JVM is compatible with the standard JVM distributed by Sun. It is able to support many existing Java applications, including the *CaffeineMark* suite, the *HotJava* browser, and the Java Media Framework, successfully without any modifications.

5.1 Future Work

There are still much work to be done to transform the Java platform completely into a quality of service oriented platform suitable for continuous media processing. The most immediate work involves examining the class libraries that come with the JVM package, such as the Abstract Windowing Toolkit API, Swing, and networking APIs to take advantage of the enhanced capabilities of the new JVM. Extension APIs such as the Java Media Framework must be examined as well, in order for them to take full advantage of the new platform.

Another direction to take would be application and middleware development. Using Q-JVM, it is possible to develop a multimedia middleware based on the Java Media Framework with resource management capabilities. In addition to the media manipulation features supported by the JMF, it could feature a general purpose and extensible resource manager and security manager. However, the suitable management policies for such managers will be the subject of future research.

6 References

- [1] K. Arnold and J. Gosling. The Java Programming Language, 2nd Edition. Addison-Wesley, Reading, Massachusetts. 1998.
- [2] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. Move-To-Rear List Scheduling: a new scheduling algorithm for providing QoS guarantees. In *Proceedings of The Fifth ACM International Multimedia Conference*, pp.63-73, Nov. 9-13, 1997.
- [3] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of ACM SIGCOMM*, pp. 1-12, September 1989.
- [4] S.J. Golestani. A Self-Clocked Fair Queueing Scheme for Broadband Applications. In *Proceedings of the 13th Annual Joint Conference of the IEEE Computer and Communications Societies on Networking for Global Communication*. 2:636-646, IEEE Computer Society Press, June 1994.
- [5] P. Goyal, X. Guo, and H.M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation*, pp. 107-121, October 1996.
- [6] P. Goyal, H.M. Vin, and H. Cheng. Start-time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *Proceedings of ACM SIGCOMM'96*, pp. 157-168, August 1996.
- [7] JavaSoft. Multithreaded Implementation and Comparisons, A White Paper. Available at <http://solaris.javasoft.com/developer/news/whitepapers/mtwp.html>. Part No.: 96168-001. JavaSoft. April 1996.
- [8] J. Nieh et al., "SVR4 UNIX Scheduler Unacceptable for Multimedia Applications", *Lecture Notes in Computer Science*, Vol. 846, D. Shepherd et al. (eds.), Springer Verlag, Heidelberg, Germany, 1994, pp. 41-53.
- [9] K. Nilsen. Java for Real-Time. In *the Journal of Real-Time Systems*, 11(2):197-205, Number 2, 1996.
- [10] I. Stoica, H. Abdel-Wahab, and K. Jeffay. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *Proceedings of the IEEE Real Time Systems Symposium*, December 1996.