Chocolate: A Reservation-Based Real-Time Java Environment on Windows/NT

Dionisio de Niz and **Ragunathan (Raj) Rajkumar**
Real-time and Multimedia Systems Laboratory
Departments of Electrical & Computer Engineering and Computer Science
Carnegie Mellon University
dionisio@cs.cmu.edu, raj@ece.cmu.edu

Abstract

In this paper, we present Chocolate*, a reservation-based Real-Time Java run-time environment that runs on Windows NT. We first present a brief overview of the emerging Real-Time Java standard and the Resource Kernel (RK) approach to providing timely, guaranteed and enforced access to system resources.*

Chocolate *is a real-time Java Virtual Machine that interfaces the real-time Java programming language with the abstractions of a resource kernel. It also supports memory regimes to control allocation time and a protocol to bound priority inversion. This version of* Chocolate *is implemented on top of NT/RK, an OS environment that includes a "portable resource kernel" within the NT kernel. Our detailed evaluation of* Chocolate *shows that the overhead introduced by NT/RK is acceptable. A Real-Time Java audio package on* Chocolate *demonstrates significantly better performance than its non-real-time counterpart. However, our Hartstone benchmark evaluations also show that our NT/RK implementation does have its drawbacks due to the lack of hard real-time capabilities within Windows NT.*

1. Introduction

Among the efforts to provide better development and deployment environments for real-time systems, Java is particularly interesting. Sun Microsystems originally developed Java as a special language for embedded systems. However, Java was widely accepted as a language to deploy applications on the web, tilting its evolution toward programming simplicity and downloadable applets. This programmer-friendly orientation introduces advantages and disadvantages in Java's usage as a development platform for real-time systems.

1.1. Java as a Real-Time Programming Language

The main disadvantages of Java as a real-time development tool are two-fold. First, its usage of a machine-independent executable code (byte code) that is interpreted at run time slows down performance. Secondly, its garbage-collected dynamic memory management could interrupt the execution of applications for unpredictable intervals of time. The first issue has been addressed partially using just-in-time compiling techniques that compile a program at run-time. No definitive solution has been established for the second memory management problem and different approaches are being explored.

An additional problem that must be solved to make Java a true real-time development tool is its linkage with the scheduler of the host operating system. Since the Java virtual machine relies on the host operating system to schedule its threads, this operating system must be capable of real-time scheduling. This is not true for all the operating systems where Java has been implemented and therefore a complementary solution should be provided.

Despite these drawbacks, numerous advantages support Java as a good language for modern real-time systems. Nielsen [2] provides a good summary of the main benefits of the usage of Java in real-time systems: language simplicity, enhanced portability, application robustness, high adaptability, and built-in security. These benefits have motivated different groups to work on the definition of a standard for Real-Time Java. Two such industry groups targeting a Real-Time Java specification are the "Real-Time Java Experts Group"[1], which is sponsored by Sun Microsystems and led by IBM, and the "J Consortium".

1.2. Scheduling in Real-Time Java

In addition to the RTJS, the J Consortium is defining another specification. This specification [19] proposes a new set of classes parallel to the normal ones to be used in real-time systems. The interactions between these new classes, called *Core* classes, and the regular classes are restricted. Priority ceiling protocol emulation is proposed as the priority inversion avoidance protocol. For dynamic memory, allocation contexts are proposed to group core objects that should be freed upon disposal of the allocation context. In addition, a real-time garbage collector is also proposed that should be driven by the real-time profile of the application.

[1] The second author of this paper is a member of the RTJEG.

Both Real-Time Java and J require a fixed-priority preemptive scheduling policy. In addition, Real-Time Java also specifies a scheduling policy interface that allows the incorporation of new scheduling policies in standard compliant ways. This interface was deliberately introduced to encourage and facilitate the insertion of new scheduling policies.

One of the policies envisioned by the RTJS for periodic thread scheduling reflects the work done in resource kernels. Resource kernels [1,3,4] provide an abstraction called "reserve"[2] that represents a guaranteed and enforced time-share of a time-multiplexed resource such as CPU, network or disk bandwidth. The kernel ensures that the reserves granted do not exceed the capacity of the resource and enforces their use.

In this paper, we study in detail the support for a resource kernel interface within Real-Time Java. We have modified the Personal Java 1.2 source code provided by Sun Microsystems to build a resource-based Java virtual machine semantically compliant with the first release of the RTJS specification.

1.3. Organization of the Paper

The rest of this paper is organized as follows. In Section 2, we provide some background on reservation abstractions in real-time operating systems and the Real-Time Java standard. An overview of related work is also presented. In Section 3, we present the design of our Chocolate Real-time Java system that allows reservation-based guarantees and enforcement. Notably, Chocolate runs on top of Windows/NT augmented with a portable resource kernel from Carnegie Mellon. In Section 4, we present a detailed evaluation of Chocolate including measurement of overheads and a detailed study of an audio application. Finally, we present our concluding remarks in Section 5.

2. Background

In this section, we briefly describe the core features of the Real-Time Java Specification, and the abstractions supported by the resource kernel.

2.1. The Real-Time Java Specification (RTJS)

The Real-Time Java Specification (RTJS) is being developed by the Real-Time Java Experts Group (RTJEG) to standardize the set of features and API a real-time JVM implementation should have. This specification addresses the real-time Java requirements proposed by NIST [21] that were also aimed at limiting the required modifications to the JVM. To develop this specification, the RTJEG defines six guiding principles that in turn led to seven functional areas in which the extensions were defined. The six guiding principles for Real-Time Java are

1. The specification should not include any requirement that limits its implementation to any particular Java version or environment (Embedded Java, Personal Java or Micro Java).
2. The modifications shall not prevent the execution of properly written non real-time programs designed for any other JVM implementation.
3. Even though the difficulty in specifying platform-independent real-time parameters is recognized, the 'Write Once Run Anywhere' philosophy must be pursued to the largest possible extent.
4. The specification should include the current practice features of real-time systems while allowing the addition of advanced ones in the future.
5. The most important feature to be pursued is predictability even at the cost of some general-purpose performance.
6. The modifications should not include any extension to the language that require compiler changes and hence increasing the likelihood of timely releases.

The seven functional areas of focus for Real-Time Java are thread scheduling and dispatching, memory management, synchronization and resource sharing, asynchronous event handling, asynchronous transfer of control, thread terminations, and physical memory access. Chocolate currently provides expanded support for the first three of these areas while including support for reservation-based scheduling and enforcement.

- **Thread scheduling and dispatching**. A fixed-preemptive priority scheme with FIFO order inside priorities is defined to be the base scheduling policy. In addition, an API and a set of features are defined to host a variety of other scheduling mechanisms and policies.
- **Memory Management**: Both the advantages and drawbacks of garbage-collected memory are recognized. Therefore, two simultaneous approaches are used to overcome the drawbacks: new memory regimes *without* garbage collection are introduced and standard interfaces to install real-time garbage collectors are also available.
- **Synchronization and Resource-Sharing**. The priority ceiling emulation and priority inheritance protocols are established to bound the priority inversions in the monitors used to implement the keyword `synchronized`.
- **Asynchronous Event Handling**. New extensions are specified to allow event-driven real-time applications to be implemented in Real-Time Java. Asynchronous event objects are added to represent asynchronous events that are expected to occur. These objects are then mapped to asynchronous event handlers to be executed whenever the event happens. Finally, these handlers are executed as real-time threads.
- **Asynchronous Transfer of Control**. The semantics of the `interrupt()` method is expanded allowing to

[2] Short for resource reservation

occur anywhere in the code instead of only in certain blocking calls. An `AsynchronouslyInterruptedException` is then added to specify, through the `throws` clause in its signature, whether a method is able to receive this exception.

- **Thread Termination**. A means is defined to request the orderly termination of a thread. It uses the asynchronous events mechanism to trigger the `interrupt()` method of the thread. The new semantics of this method would then enable correct thread termination.
- **Physical Memory Access**. A new memory region that can be mapped to a fixed physical location is defined. This enables threads inside Real-Time Java to communicate with physical devices directly.

2.2. Resource Kernels

A resource kernel is an OS subsystem that permits the timely, guaranteed and enforced access to system resources [1,3,4]. A resource kernel uses the resource reservation as the core abstraction to implement both access control and enforcement. This *reservation* represents a portion of a shared resource like CPU, network bandwidth, or disk bandwidth. Because a thread (or process if no threads exist in the operating system) can use more than one of these resources, reservations are aggregated in a *resource set* that in turn is associated to this thread. This is illustrated in Figure 1. Moreover, multiple threads can be attached to a single resource set.

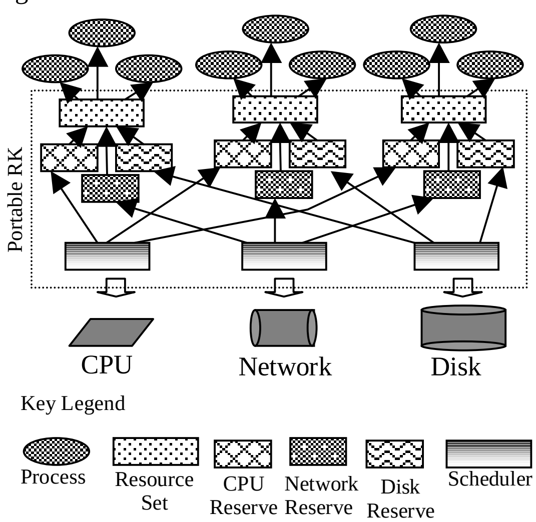


Figure 1: Portable RK Architecture

A resource kernel defines four mechanisms to implement its functionality:

- *Admission control*: The creation of a *reserve* is conditioned not to exceed the capacity of the resource, considering the previously granted *reserves*.
- *Scheduling* implements the dynamic allocation of the resources at run-time to give the threads their reserved amount.
- *Enforcement* limits the consumption of the resources to the reserved quantity.
- *Accounting*: An accounting mechanism is used to support the scheduler and enforcement mechanisms.

These *reserves* are expressed in time units given that they are time multiplexed. This multiplexing is expressed as a fixed amount of time-share of the resource during well-specified periodic intervals. Three parameters[3] are used to specify this time-share:

- the dedicated time *C* that specifies the time the resource should be used for this reserve in each period,
- the period (T) itself, and
- the deadline (D) that specifies the time, after the beginning of the period, in which the dedicated time should be available to the thread.

An additional parameter for enforcement and replenishment is attached to a reserve to obtain fine-grained control over unused CPU resources. This determines the scheduling of a thread with a depleted reserve, i.e. its *C* time for the period is exhausted. The parameter can have the value HARD, FIRM or SOFT. HARD implies that the thread should not be scheduled until the reserve is replenished (at the beginning of the next period). FIRM means that the thread can be scheduled only in the case that no other unreserved task is ready to execute. Finally, SOFT means that the thread can be executed in background priority.

2.3. Related Work

Both academic and commercial efforts are looking at various aspects of Real-Time Java. Our work provides real-time scheduling, CPU time guarantees and enforcement, memory regimes and priority inversion avoidance mechanisms. The combination of these features is not implemented in any other work discussed below.

Efforts in Academia: The Java Nanokernel (JN) [7] is a specialized JVM designed to work on a single chip attached to the Internet. It was developed by B. R. Montague and his group from UCSC. This JVM was designed to support soft real-time applications and is implemented on top of a nanokernel also developed by the group. This nanokernel was developed only to support the Java threads, avoiding any other functions regular kernels implement. This implementation uses priorities to schedule its threads and does not implement any priority inversion avoidance protocol or special memory regimes. At the University of Karlsruhe, Germany, a Multithreaded Java microcontroller was developed to implement thread-oriented real-time event-handling. In this project, Brinkschulte et al [8] developed a special microcontroller that substitutes the interrupt service routines with what they call "Interrupt

[3] In our current implementation of Chocolate and NT/RK, only C and T are specified and D is assumed to be the same as T.

Service Threads" (IST). These ISTs are blocked threads waiting for an event (interrupt) to happen. Whenever the event happens, the thread is activated to attend to the interrupt. This scheme allows several interrupts to be serviced at the same time in a multithreaded fashion. Moreover, these threads can be scheduled with different policies, even though their specific implementation focuses on proportional sharing. The hardware uses different sets of registers on the chip to achieve zero latency context-switching among up to four of these threads. Neither priority inversion avoidance nor real-time dynamic memory schemes are addressed in this implementation.

Commercial Efforts: PERC [9] from NewMonics is a real-time Java implementation for hard real-time applications. This JVM uses compile-time techniques to annotate class files with real-time requirements. Four new types of execution entities, called tasks, are added: Periodic, Sporadic, Spontaneous, and Ongoing. The Periodic task is scheduled according to the requested duration of execution and period. The other three types of tasks are triggered by events with different modes of limiting their frequency, execution time and deadline. Time-limited execution blocks were also added to cope with unbounded priority inversions. Whenever this block is run, a time-out exception must be caught to execute an orderly abortion. Chai [10] from HP is a VM implementation for soft real-time applications. It is specially designed for embedded applications. Chai implements an incremental garbage collector that runs concurrently with the application threads. The priority of the threads is internally mapped to the host RTOS priorities and no indication of priority inversion avoidance mechanisms were found in the documentation. Finally JBED [11] from Esmertec is a JVM that implements EDF scheduling with access to physical memory and an incremental real-time garbage collector. No priority inversion avoidance mechanisms are implemented.

3. The Design and Implementation of *Chocolate*

In this section, we first describe NT/RK, the host OS for our current implementation of Chocolate.[4] We then describe the class structure proposed by the RTJS, and the Chocolate design and implementation of the structure. RTJS adds a new package to JDK called `javax.realtime`. This package is composed by all the interface classes that a real-time JDK can implement. The classes can be divided into the seven areas discussed in Section 2. We focus our discussion only in the areas currently supported by Chocolate.

[4] Chocolate is currently being ported to Linux/RK, our implementation of the portable resource kernel within Linux [3].

3.1. NT/RK[5]

Windows NT 4.0 supports 32 priority levels divided into two classes, variable-priority and real-time priority. The variable-priority class uses the lower 16 priority levels with a "priority boost" mechanism to increase the responsiveness of interactive applications. On the other hand, the real-time priority class uses the higher 16 priority levels and the threads running in this priority class share these levels with kernel threads. This priority class is associated with a process and its threads are then limited to the priority levels of that class.

We have added our Portable Resource Kernel to run as a privileged device driver within the Windows NT 4.0 kernel with SP 3. This driver works as a kernel module to form "NT/RK" and provides the semantics of the Resource Kernel abstractions of reserves by hooking into a context-switch hook available within the "Checked Build" version of Windows NT. NT/RK uses the variable-priority class to schedule all the threads appropriately and utilizes a 'background' priority to schedule a depleted SOFT reserve. The current version of the resource kernel in NT (NT/RK) implements only CPU reservations. These reservations are used to implement Java real-time threads with periodic scheduling parameters in Chocolate.

NT/RK uses these priorities along with two timers and a thread-swap notification from the NT kernel to schedule the threads. For each CPU reserve in the system, one timer is set to the beginning of the period. Whenever this timer expires, the reserve is replenished, and another timer is set to the end of the C time. In addition, the thread is 'activated' raising its priority to the highest of the variable-priority class. Then, whenever the C timer expires or RK is notified that this thread was swapped out of the processor, its reserve is marked as depleted, its used time is updated, and the thread is assigned a '*background*' or '*idle*' priority depending on the enforcement mode of the reserve.

3.2. The Core Real-Time Classes

This section describes the class hierarchy of the RTJS and the subset that Chocolate implements. This hierarchy illustrates the API available to the applications.

3.2.1. RealtimeThread

The core of the `javax.realtime` package used by Real-Time Java is the `RealtimeThread` class. Figure 2 depicts the class hierarchy associated with the `RealtimeThread` class. As can be seen in the figure, two parameters (children with diamond arrow-heads) are attached to the `RealtimeThread` class. Both parameters are given to the constructor of the `RealtimeThread` class to specify the desired behavior of the scheduler and the memory allocator for this thread.

[5] RK stands for "Resource Kernel".

The RTJEG specification envisions four types of scheduling parameters:

- the periodic parameters to specify the period, cost and deadline of threads to be used with schedulers based on rate-monotonic analysis,
- the priority parameters, to specify a fixed priority to be used for the scheduling, and
- two subtask parameters to specify the parameters for a dynamic deadline scheduler.

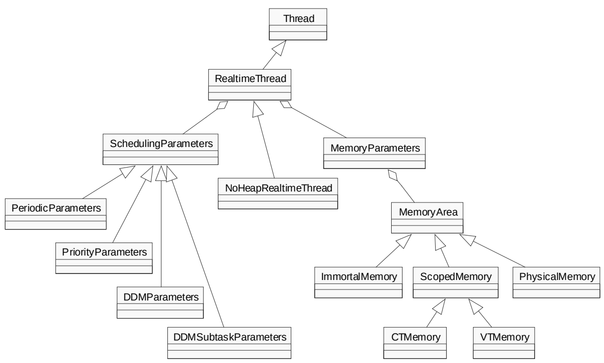


Figure 2: The RTJS class hierarchy

On the other hand, the memory parameters specify the memory regime to be used for the new allocations inside the thread, and the allocation rate to be matched by the garbage collector for the garbage collected memory.

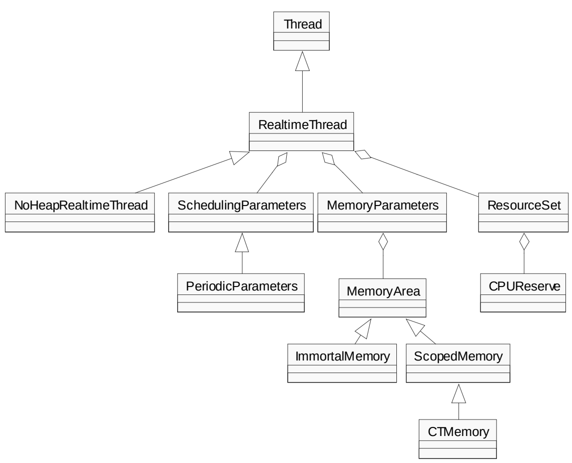


Figure 3: Chocolate Class hierarchy

From this hierarchy, Chocolate uses a subset with some additions as depicted in Figure 3. As can be seen in the figure, Chocolate uses only the periodic scheduling parameter and implements just immortal and constant time memory regimes. On the other hand, two new classes are added to the hierarchy to hold the reserves needed by RK.

Both the `ResourceSet` and `CPUReserve` objects attached to the `RealtimeThread` object are created in the constructor of the `RealtimeThread` class. From the constructor of the former classes, RK is then called to create the corresponding RK instances and the handles are stored inside the Chocolate objects. Finally, the resource set is attached to the native thread whenever this thread is created inside the `start()` method of the Chocolate thread.

3.2.2. RealtimeSystem

A new `RealtimeSystem` class is introduced in the `javax.realtime` package to set system-wide parameters. These parameters include the maximum number of concurrent locks and the monitor control for priority inversion minimization. In addition, references to the garbage collector, security manager, and scheduler can be obtained with static methods implemented by the `RealtimeSystem` class. These references can then be used to query or set parameters for these particular components. In Chocolate, only the method to set the monitor control is currently implemented.

3.3. The Architecture of *Chocolate*

Chocolate was developed from the Sun Microsystems Personal Java JDK1.2 source code for Windows NT. Figure 4, depicts a partial architectural diagram of JDK 1.2. As can be seen in this figure, JDK 1.2 is structured in layers. The class layer is composed of the classes available to the application. These classes include Java classes and native methods that call the JVM API. The JVM API is the second layer and isolates the C code common to all platforms (currently only Solaris and Windows NT 4.0) from the platform-dependent code. Finally, this platform-dependent code constitutes the platform-dependent layer that embeds the code and system calls that depends on the host operating system and hardware.

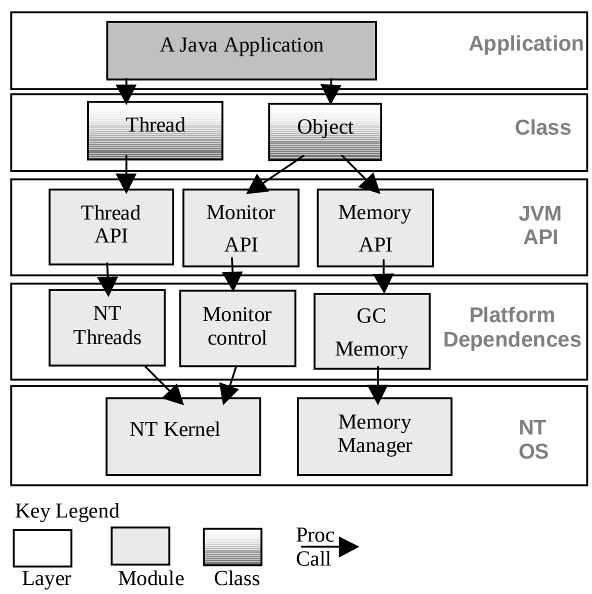


Figure 4: JDK 1.2 Architecture

Based on this architecture, modules were added as shown in Figure 5 to each of the layers (including the OS) to implement Chocolate. In the class layer, the classes added have already been discussed in the previous subsection. In the JVM API layer, a new module was added to support the API and the RK interface to support the implementation of the CPU reserves, resource sets, and real-time threads classes. In addition, the memory API module was modified to use the time-bounded memory allocation module for the memory allocations done by a thread with a constant time or immortal memory area attached. Three modules were then added to the platform-dependent layer. The first module was the NT/RK implementation that basically loads and calls the NT/RK driver. The second was the priority inheritance module that is called by the monitor control module whenever priority inheritance is active. Finally, the last module is the time-bounded memory module that handles the memory allocation from constant time and immortal memory areas. Both the priority inheritance policy and the NT/RK port of Chocolate use the NT/RK driver to implement their functionality.

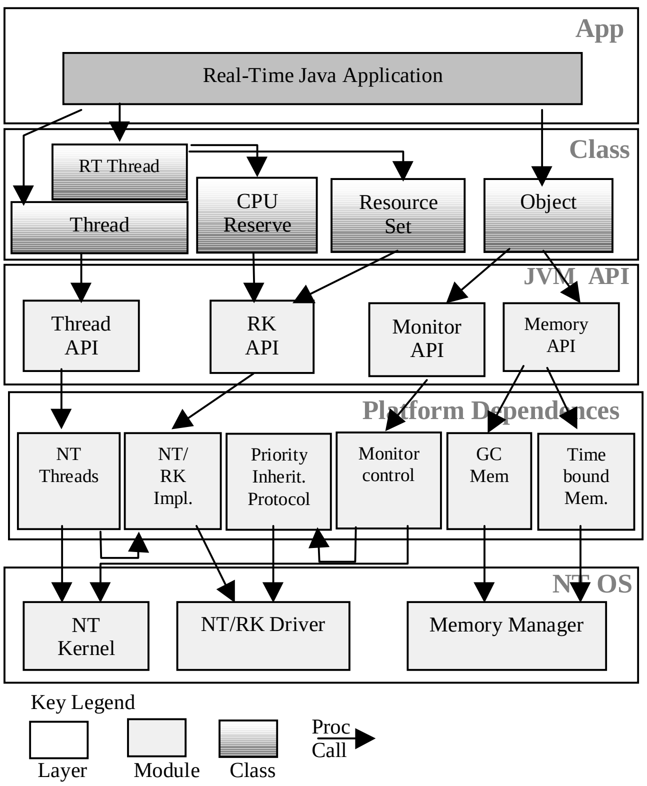


Figure 5: The Architecture of Chocolate

3.4. Constant Time and Immortal Memory

In Real-Time Java, memory regions called *regimes* can be created to have well-defined access properties. Real-time threads can then be created so that the objects they create and/or access are associated with these memory regimes. Two memory regimes of interest to real-time applications are immortal and constant-time memory regimes. Objects created in an *immortal memory regime* can persist outside the scope of the application that created the objects. Objects can be created in a *constant-time memory regime* in constant time independent of the size of the object (modulo maximum size possible). Maximum size constraints apply to all special memory regimes.

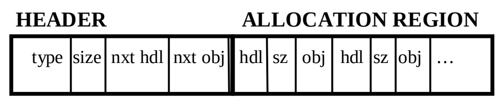


Figure 6: CT and Immortal memory structure

Both the immortal and the constant-time memory regimes are implemented in Chocolate as a continuous memory area allocated at creation time. Figure 6 depicts the structure of the memory area. This area is divided into a header and an allocation region. Inside the header, the type field stores whether this area is a constant time or immortal memory area. The size of the allocation region is also kept and the last two fields keep track of the next available handle and object. In the allocation region, new objects are created along with their handles. Because the size of the objects is variable, this size is stored in the first word following its handle.

The bounded-time memory module allocates objects from both types of memory in identical fashion except for the fact that the allocation from the immortal memory area is considered a critical section and thus it is protected with a monitor. This is due to the fact that in Chocolate the constant-time memory is not shared between threads whereas the immortal memory is. This limitation was imposed to the constant time memory area to avoid the time variation a synchronization mechanism would imply. The limitation is not necessary for the immortal memory because it is *not* required to have a constant-time allocation operation.

The memory API was modified to allocate objects with the time-bounded or the regular heap allocator depending on the existence of a memory area associated with the calling thread.

3.5. Bounding Priority Inversion

The problems of priority inversion that can arise in the use of synchronization mechanisms in real-time systems are rather well-understood [5,20]. In [5], the author discusses two protocols to bind the time of the priority inversion: the priority ceiling emulation protocol and the basic priority inheritance protocol [20]. In this section, only the priority inheritance protocol is discussed given that is the only one Chocolate implements.

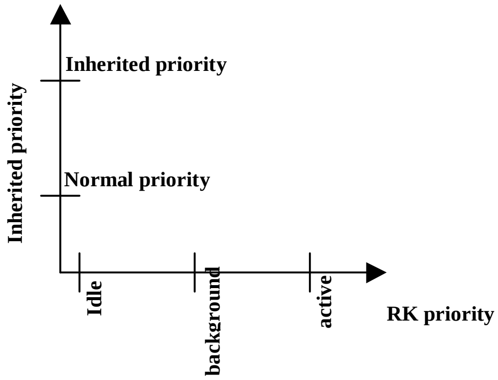


Figure 7: Priority Dimensions

A priority inversion occurs when a thread trying to lock a synchronization mechanism (mutex, semaphore, monitor, etc.) is blocked waiting for another thread to release this mechanism, and the holder of the lock has lower priority than the blocked one. Borger and Rajkumar in [6] describe an implementation of the priority inheritance protocol in an Ada run-time environment. In this implementation, the blocked threads waiting for the lock are pushed on to a stack while the one holding the lock inherits the priority of the thread being blocked if it is higher that its own. This stack has the property of always having on the top, the thread with the highest priority. Thus when the lock is released, it is granted to this thread[6].

In the presence of RK, the implementation of this mechanism is more complex. This is because RK changes the priority of a thread depending on the state of its reserve[7]. Therefore, a race condition can occur with these two mechanisms changing the priority without coordination.

In the coordination scheme used in Chocolate the two mechanisms were designed to work on separate dimensions: the inherited priority and the RK priority, as can be seen in Figure 7. A third mechanism is then used to set the effective priority of the thread to be the larger of the two priorities.

3.6. The *Chocolate* API

A further discussion of the API and its usage is presented here in order to provide a better understanding of the programming model.

Periodic scheduling parameters: These are used to specify the periodic CPU time needed by a real-time thread. An object of this class is passed to the constructor of the `RealtimeThread` class for this purpose. The constructor of the periodic parameter class takes as parameters the *C* and *T* times, discussed previously, each specified as two sub-parameters, one for the number of seconds and the other for the nanoseconds. In addition, given that NT/RK needs the operation mode of the enforcement, replenishment and scheduling, these are combined with bit operations into the *importance* field of the periodic parameter. This is done to be compliant with the RTJS specification.

Memory parameters: Chocolate uses the memory parameters to pass a memory area object to the thread constructor. This memory area is either an Immortal memory area or a Constant time memory area. In turn, this area is created with the size (in bytes) that should be available for allocation.

Monitor control: The priority inheritance protocol is activated calling the static method `setMonitorControl` of the `RealtimeSystem` class. In Chocolate, the priority inheritance protocol can be activated for a particular object's monitor or for any monitor. Therefore, the `setMonitorControl` method can take one or two parameters. The first one is the object over which the priority inheritance protocol is to be activated, and the second is an object of the class of the monitor control to be used (only the priority inheritance protocol is supported at this time).

`NoHeapThread`. Finally, this special kind of real-time thread requires a constant time memory area to be passed to its constructor given that it is restricted from allocating objects in the garbage-collected heap.

4. Experimental Evaluation of *Chocolate*

In this section, we present some experiments done with Chocolate and NT/RK as the host OS being used. Our goals are two-fold. First, we measure the overheads introduced by the additional functionality of reservation-based scheduling within NT/RK. Secondly, we perform a detailed experiment with a real-time application to determine the validity of our approach and the resulting timing behavior. In this experiment, the Java sound package was modified to use the new thread class. Then it was compared with the original sound package with regular threads.

All experiments described here were run on a 300 MHz Intel Pentium II with 96MB of memory. The network driver was de-activated to minimize variability during the experiments.

4.1. Overhead Measurements on NT/RK

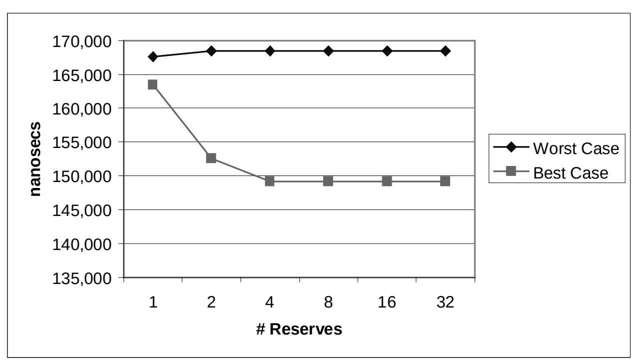


Figure 8: Overhead of Admission Control .

---

[6] This works only when threads do not suspend inside the critical section.

[7] See 3.1 for a full explanation.

The experiments on NT/RK were designed to measure the overheads imposed by its added functionality. Three overheads are especially pertinent for Chocolate: admission control, replenishment and enforcement.

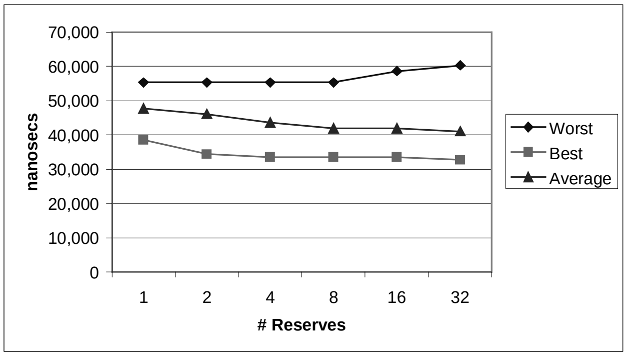
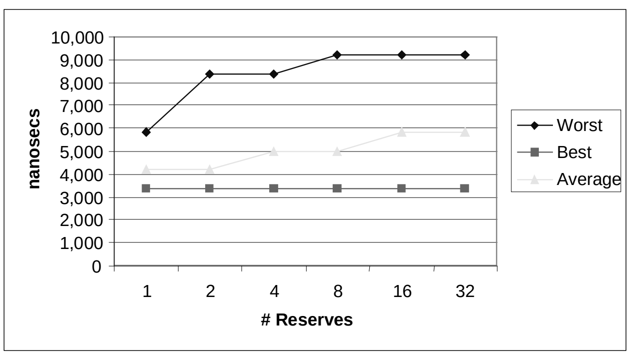


Figure 9: Replenishment execution time

The results for admission control are presented in Figure 8. As can be seen, the best case overhead is near 150 µs while the worst case overhead is about 170 µs. Figure 9 presents the overhead numbers for the replenishment execution-time. As can be seen, the overhead is almost constant with the worst case growing 5 µs in the case where the number of reserves is 32.



A Figure 10: Enforcement execution time

Figure 10 presents the overhead for the enforcement of reserve execution time. As can be seen, the worst case is almost double that of the average case. This variation is likely to be due to the preemption of other kernel threads. These preemptions have a larger impact on the execution time of the enforcement mechanism than on the other two mechanisms given that its execution time is smaller. These overhead numbers are smaller due to the fact that the replenishment routine includes an enforcement sequence and the replenishment routine also involves computing the excess time to be discounted from the *C* time for the next period.

We draw two generally positive conclusions from the overhead measurement experiments. The first is that the overheads are of the order of tens of microseconds and acceptable. The second conclusion is that overheads, though some increase with reservations, are not extremely sensitive to the number of reservations in the system.

4.2. Real-Time Java Applications on *Chocolate*

The experiments on Chocolate are aimed to verify the effectiveness of the timing guarantees it can provide. The first experiment is the design of a real time-sensitive application to test the time guarantees in a real application. On the other hand, the second experiment is designed to discover the limits of this functionality.

4.2.1. Time-Sensitive (Audio) Application in *Chocolate*

Figure 11: Time trace of the sound player thread (trace used for calibrating *C* and *T* values)

We now modify the Java sound package to use real-time threads in an effort to verify the timing guarantees available to the real-time threads. Then, this new package was tested playing an audio clip while threads to overload the processor are created. This test is executed with and without a CPU reserve associated with the sound thread.

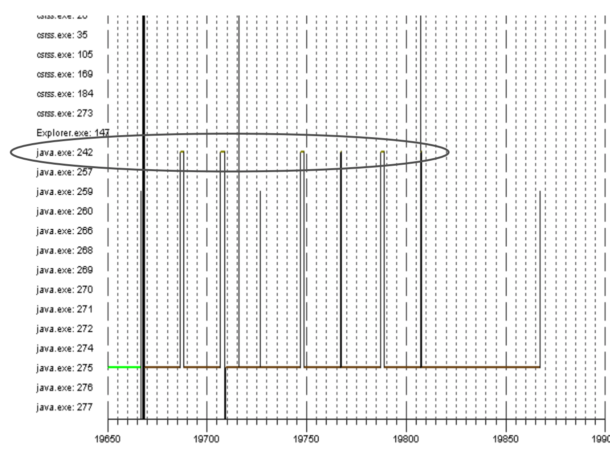


Figure 12: Time trace chart of the Java sound player thread ("java:242") *<u>without</u>* any reservation

To design the test, the thread that controls the inner loop of the sound player was converted to a real-time thread. Then the *C* and *T* times were determined using the profiling package TimeTrace, a commercially available tool from TimeSys Corporation. Figure 11 depicts the active thread at any instant of time. This graph helped us to determine both *C* and *T*. In this case, *T* was set to 11 ms while *C* was chosen to be the largest time used by the thread plus some extra time to cope with its cache-related variance. In this case, the *C* time was chosen to be 4 *ms*. Next, the tests with and without the reserve attached were executed. Figure 12 shows the time trace of the sound player thread *without* any reservation. Two observations are important: first, the period is not uniform. And second, the thread is preempted by an unbounded period of time as can be seen after the last activation of the thread around 19800 ms. In this case, the next activation is not shown in the chart but is beyond 19900 ms

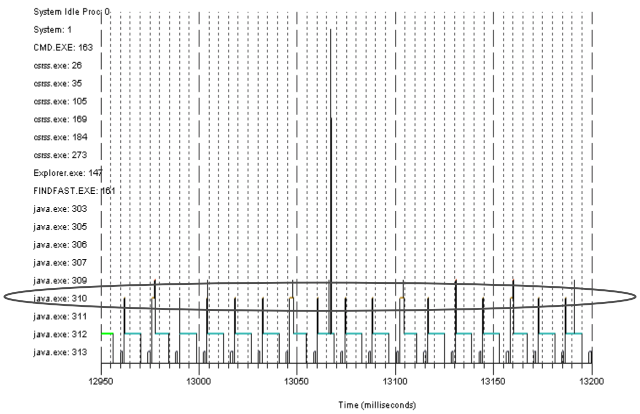


Figure 13: The Time Trace of the Real-Time Java sound player thread ("java: 310") *with* reserves

On the other hand, when a reserve is attached to the thread and the replenishment and enforcement mechanisms are actively working on it, both *C* and *T* are guaranteed to this Real-Time Java thread.

Figure 13 shows the time trace chart of the thread. Notice that the period is now uniform. However, the *C* time is not uniform. This is due to the fact that the thread is free to yield the processor whenever it is done for the period, even though its reserve could have a larger number for it.

4.2.2. The Hartstone Benchmark

The Hartstone benchmark [15] was developed by the Software Engineering Institute to test the ability of a system to cope with the deadlines of real-time threads with different workloads and frequencies. The workloads are based on the Whetstone benchmark [17,18], and different values for both the workload and the frequencies are tested to discover the limits of a system.

This benchmark was used to characterize the hard real-time features of Chocolate. Two versions of the benchmark were implemented: the first with regular Java threads and the second with Chocolate real-time threads.

Table 1 shows the base case used for our test. The load is expressed in kilo Whetstones (KWS) and both the frequency and the period are shown.

Four numbers are gathered in each run of the test: the met deadlines, the missed deadlines, the skipped deadlines and the cumulative late time. The meaning of the first two numbers is obvious but the last two need further explanation. The skipped deadlines are due to the periods skipped in a missed deadline to prevent cumulative lateness from triggering another missed deadline. On the other hand, the cumulative late time is the sum of the elapse time from the beginning of the period to the activation of the thread whenever a deadline is missed. From the base case, different cases were tested increasing the threads' frequency in each new test.

Raw Speed 135,000.00 KWS/s

Thread	Frequency	Period	KWS	KWS/s	Requested Workload Utilization
1	2	0.5	320	640	0.474%
2	4	0.25	160	640	0.474%
3	8	0.125	80	640	0.474%
4	16	0.0625	40	640	0.474%
5	32	0.03125	20	640	0.474%
				Total	2.370%

Table 1. The Base Case for the Hartstone Benchmark.

Raw Speed 135,000

Regular Java Threads

Task	Frequency	Load (kws)	Load %	Deadlines Met	Missed	Skipped	Lateness (ns)
1	2	3038	4.50%	20	0	0	0
2	4	1519	4.50%	40	0	0	0
3	8	760	4.50%	80	0	0	0
4	16	380	4.50%	160	0	0	0
5	32	190	4.50%	314	3	3	157,500
	Totals	5887	22.51%				

RT Java Threads

Task	Frequency	Load (kws)	Load %	Deadlines Met	Missed	Skipped	Lateness (ns)
1	2	3038	4.50%	20	0	0	0
2	4	1519	4.50%	40	0	0	0
3	8	760	4.50%	80	0	0	0
4	16	380	4.50%	160	0	0	0
5	32	190	4.50%	320	0	0	0
	Totals	5887	22.51%				

Table 2. Real-Time Java vs. Non-Real-Time Java Comparison

In Table 2, we present a set of results comparing Real-Time Java with its non-real-time counterpart using a standard Hartstone task set. As can be seen, Real-Time Java threads do not miss any deadlines but non-Real-Time Java threads do miss their deadlines at just 23% processor utilization. However, we also found that the Real-Time Java threads are *not* schedulable at high levels of processor utilization before deadlines begin to be missed. Such high degrees of processor utilization would be expected from a real-time system performing appropriate scheduling. One of our main conclusions from these tests is that Chocolate running on NT/RK is *not* well suited to host hard real-time applications. We attribute this primarily to the lack of hard real-time support within Windows NT where the kernel is non-preemptive, and the DPCs (deferred procedure calls) of device drivers can execute at higher priority than the scheduler itself leading to potentially serious priority inversions. Despite these observations, Chocolate does offer reliable timing behavior and protection for soft real-time applications. In addition, we expect Chocolate to be well suited for hard real-time applications running on Linux/RK [3] where the scheduling is under direct programmer control.

5. Concluding Remarks

In this paper, we described an implementation of a JVM compliant to the RTJS based on NT/RK, a resource kernel. Three main features were implemented: a real-time scheduling policy, a priority inversion avoidance protocol, and real-time memory regimes. The implementation of the scheduling policy allowed us to test the portability of the Resource Kernel [3], and the flexibility of its time parameters. Both characteristics helped us to construct a soft real-time execution environment that enabled us to port the Java audio package. A direct comparison of the Java audio package and our Real-Time Java package visually and aurally revealed the better timing behavior delivered by our Real-Time JVM named Chocolate. However, the Hartstone benchmark [15] helped us to determine the limits of the Windows NT implementation, leading us to conclude that the time protection implemented by NT/RK does *not* include protection against kernel threads and therefore is not suitable for all hard real-time applications.

We also described a way to implement the priority inheritance protocol in the presence of a resource kernel. We showed in our experiments the use of TimeTrace, a profiling tool to determine the computation requirement of real-time threads. Given the memory regimes used in Chocolate, the memory allocation profile is as important as the execution time. Therefore, techniques are necessary to determine this memory usage profile.

This work needs to be extended in multiple directions. Firstly, the complete Real-Time Java specification needs to be implemented and experimented with. Next, in order to provide full functionality to real-time applications, the services offered by the OS and the Real-Time Java extensions built on top of them should be modified to offer timing guarantees in accessing other resources. Disk bandwidth and network bandwidth management [16] are among the most important but with implications to co-scheduling the controlling CPU resource with controlled resoruces. Finally, while Java offers a reasonable security model, the resource reservation scheme introduced by resource kernels presents new potential challenges. Not only is it necessary to determine who has access to what resource but now we need to know how much a user can consume. Furthermore, Java allows dynamic code migration. The protection scheme for Real-Time Java, therefore, needs to reflect the semantics of the application and the working environment.

6. References

1. Cliff Mercer, Ragunathan Rajkumar, and Jim Zelenka. *Temporal Protection in Real-Time Operating Systems*. In Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software. May 1994
2. Kelvin Nilsen. *Issues in the Design and Implementation of Real-Time Java*. Newmonics. Inc. July 19, 1996.
3. Shuichi Oikawa and Ragunathan Rajkumar. *Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior*, IEEE Real-Time Technology and Applications Symposium, June 1999.
4. Ragunathan Rajkumar, Kanaka Juvva, Anastasio Molano and Shuichi Oikawa. *Resource Kernels: A Resource-Centric Approach to Real-Time Systems. Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking, January 1998.*
5. Ragunathan Rajkumar, *Synchronization in Real-Time Systems: A Priority Inheritance Approach*, Kluwer Academic Publishers, 1991, ISBN 0-7923-9211-6
6. Mark W. Borger, and Ragunathan Rajkumar, *Implementing Priority Inheritance Algorithms in an Ada Runtime System*. Technical Report. April 1989. SEI, Carnegie Mellon University.
7. Bruce R. Montague. *JN: An Operating System for an Embedded Java Network Computer*. IEEE Micro. Vol 17 No. 3 May/June 1997.
8. U. Brinkschulte, C. Krakowski, J. Kreuzinger, Th. Ungerer. *A Multthreaded Java Microcontroller for Thread-Oriented Real-time Event-Handling*. Technical Report. University of Karlsruhe. Germany. 1999.
9. K. Nielsen, S. Lee. *PERC™ Real-time API (Draft 1.3)*. NewMonics Inc. July 18, 1998.
10. *ChaiVM technical Summary*. Hewlett Packard. Octuber 22, 1998.
11. J. Tryggvesson, T. Mattsson, H. Heeb. *JBED: Java for Real-Time Systems*. Dr. Dobb's Journal. November 1999.
12. http://www.esmertec.com, as of December 10, 1999.
13. Anastasio Molano, Kanaka Juvva, and Raj Rajkumar. Real-Time File Systems: Guaranteeing Timing Constraints for Disk Accesses in RT-Mach. Proceedings of the IEEE Real-Time Systems Symposium. December 1997
14. Sourav Gosh and Ragunathan Rajkumar. Network Bandwidth Reservation using the Rate-Monotonic Model. Proceedings of the International Conference on Software, Telecommunications and Computer Networks, 1999.
15. Nelson Weiderman. Hartstone: *Synthetic Benchmark Requirements for Hard Real-Time Applications*. Technical Report CMU/SEI-89-TR-23, D TIC: ADA219326, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213, June, 1989
16. Patrick Donohoe, Ruth Shapiro, Nelson Weiderman. *Hartstone Benchmark Results and Analysis*. Technical Report CMU/ SEI-90-TR-7 ESD-90-TR-208. Carnegie Mellon University, June 1990
17. H.J. Curnow and B.A. Wichmann. *A Synthetic Benchmark*. Computer Journal 19(1):43-49, January, 1976
18. S. Harbaugh and J. Forakis, Timing Studies Using a Synthetic Whetstone Benchmark. Ada Letters 4(2):23-34, 1984.
19. J Consortium. *Draft International J Consortium Specification*. 1999.
20. L. Sha, R. Rajkumar and J. P. Lehoczky. *Synchronization in Real-Time Systems*. IEEE Transactions on Computers, 1990.
21. L. Carnahan, and Markus Ruark Editors. *Requirement for Real-Time Extensions for the Java Platform*. Report from the Requirements Group for Real-Time Extensions for the Java Platform. NIST. September 1999.