Mackinac: Making HotSpot™ Real-Time.

Greg Bollella, Bertrand Delsart, Romain Guider, Christophe Lizzi, Frederic Parain
First.Last@Sun.COM
Sun Microsystems
180, Avenue de l'Europe - Zirst de Montbonnot
38334 S^t-Ismier Cedex - France

Abstract

The Real-Time Specification for Java™ (RTSJ) is an API specification that allows developers to write real-time applications using the Java programming Language.

Project Mackinac is the code name of Sun Microsystems' RTSJ development effort. This paper reviews the main requirements from the RTSJ itself, presents additional requirements defined by the Mackinac implementation team, explains various implementation strategies used to meet the requirements, and presents performance results from tests executed against the current development version of Mackinac.

1 Introduction

The Real-Time Specification for Java™ (RTSJ) [3] is a specification which defines a set of library calls and semantics which, when implemented within a general-purpose Java virtual machine, allow developers to correctly reason about and control the temporal behavior of application logic written in the Java Language and executing within that modified JVM[1].

The Executive Committee of the Java Community Process approved version 1.0 of Real-Time Specification for Java in January of 2002 and since then a number of commercial implementations have started to appear.

Project Mackinac is the code name of Sun Microsystems' RTSJ development effort. This paper reviews the main requirements from the RTSJ itself, presents additional requirements defined by the Mackinac implementation team, explains various implementation strategies used to meet the requirements, and presents performance results from tests executed against the current development version of Mackinac.

[1]The terms "Java Virtual Machine" and "JVM" mean a Virtual Machine for the Java™ platform.

The RTSJ requires classes, methods, and semantics which address thread scheduling, memory management, synchronization, asynchronous activities, and physical memory access.

The RTSJ provides a rich framework for scheduling by defining specific objects which are managed by the scheduler, the notion of a feasibility set and a formalized means for communication between the application and system about the application's temporal requirements (e.g., cost, period, inter-arrival time, start time, etc.). The RTSJ requires only one 'base' scheduler but anticipates implementations providing others.

Commercial Java implementations typically utilize a garbage collector (GC). GC algorithms tuned for general-purpose performance (application throughput) do not provide the predictability needed by the class of applications served by the RTSJ. Additionally, real-time GC algorithms, which may provide better predictability, introduce overhead and other issues. Hence, the RTSJ provides a thread type which is immune to any interference by GC algorithms. Logic executing in this context can obtain latency and jitter magnitudes that rival the best commercial real-time operating systems. There is, however, no free lunch. This good predictability comes at the price of a programming model more complex than Java, but, fortunately, less complex than that required for real-time applications written in traditional development processes (e.g., C or C++).

The RTSJ designers believed that first-class support for asynchronous activities, although not in the actual design space of real-time systems software, would be at the core of applications for which the RTSJ targeted. Thus, the RTSJ does provide such support. Notably, the RTSJ has decoupled the notion of an occurrence of an event of the execution of associated logic.

In this paper we desire to convey to the reader some of the more interesting and clever implementation hurdles we had to surmount in meeting requirements imposed by the RTSJ as well as requirements we imposed ourselves and give the results of performance benchmarks. In the next section we list some of the requirements imposed by the RTSJ and ourselves without comment. The following sections explain in detail the implementation of some key features of Mackinac on the Solaris^{TM} Operating System and refer back to the the requirements section to show why such deliberate care was needed.

A word about nomenclature and formatting. In the text, when we refer to a type from `java.lang.*` we will display it as, **Thread**, and references to types from `javax.realtime.*` will be displayed as, *NoHeapRealtimeThread*.

When we use the term, 'Thread', in the normal body font we will, in most cases, be referring to an instance of **Thread**. Exceptional cases will be obvious from the context. The use of the term, 'thread' in the normal body font will refer to the general operating system notion of a locus of control within a process.

We will distinguish between `type` and `instance` in all cases. We will refer to a `type` by name, e.g., *Timer* and we will be talking about the type itself not an instance of the type. When we wish to discuss instances we will either be explicit, e.g., 'instances of *Timer*, or use the following abbreviations:

- `RTT` for an instance of *javax.realtime.RealtimeThread*
- `NHRT` for an instance of *javax.realtime.NoHeapRealtimeThread*

HotSpot will be used to refer to a particular implementation of a JVM, Sun's current Java 2 Standard Edition distribution and Mackinac will refer to a JVM modified according to the requirements and semantics of JSR-01 (the RTSJ).

2 Requirements

2.1 RTSJ Requirements

The RTSJ is a set of API specifications and a set of requirements on other parts of the Java implementation. In this section, we review some of the requirements that have affected our design of the modifications to HotSpot.

Requirement 1 *Any implementation of the RTSJ must satisfy the requirements expressed in the Java Virtual Machine Specification [9].*

Requirement 2 *Any implementation of the RTSJ must guarantee that NHRTs stay unaffected by the garbage collector as long as they do not synchronize with a GC-affected thread.*

Requirement 3 *The JVM shall not be implemented in a way that permits unbounded priority inversion in any scheduling interaction it implements.*

2.2 Mackinac Specific Requirements

In the previous section we reviewed some of the requirements that the RTSJ expresses. Those requirements leave some room for other requirements that are implementation specific. We review, in this section, the main requirements that are Mackinac specific.

Requirement 4 *We require that Mackinac can be configured so that the latency of hard real-time periodic activities is bounded within the tens of microseconds.*

Requirement 5 *We require, for the Mackinac project, that any linear section[2] of compiled application code can be configured to run deterministically, whatever bytecode is contained in this section.*

Requirement 6 *We require, for the Mackinac project, that the asynchronous event handling mechanisms are implemented in a resource efficient manner.*

Requirement 7 *We require that the impact of RTSJ on non real-time threads is limited, both in terms of performance and memory consumption.*

Requirement 8 *We require that Mackinac supports multiprocessor architectures.*

Requirement 9 *We require that Mackinac supports cost enforcement.*

3 Memory Model and Memory Managment

State of the art real-time garbage collectors target pause times of a few milliseconds [2, 4, 12]. In addition, they impose an overhead on non real-time threads that can not be ignored. Because of the limited determinism achieved by those garbage collectors and because of their overhead, they contradict requirements 4 and 7.

For those reasons, we focused on achieving very good determinism for the NHRTs while limiting as much as possible the overhead on Threads. We kept the efficient generation-based garbage collectors offered by HotSpot [6] and addressed the issues raised by the RTSJs' memory model. Those issues come from the fact that *ImmortalMemory* and *ScopedMemory* contains root pointers for the GC and are concurrently modified by the NHRTs (other threads are stopped during the GC phases). NHRTs can run concurrently with the GC because their execution stack can not contain references toward garbage-collected objects and, as such, do not need to participate in root scanning. However,

[2]We interpret this to mean an executed stream of machine code not including a backward branch.

NHRTs and the GC access ImmortalMemory and ScopedMemory areas concurrently which causes a number of issues.

We describe the memory model in the next section and then describe our solutions to the problems raised by this memory model.

3.1 Memory Model

The memory model introduced by the RTSJ is made of three different kinds of memory area: the usual garbage-collected **HeapMemory** that is found in regular JVM, *ImmortalMemory* that is never garbaged-collected and *ScopedMemory*, in which life-times are determined by lexical scoping. Threads can change the memory area in which they allocate objects by calling the `enter` method of the memory area they want to enter. The logic to be executed in the entered memory area is passed as a parameter of the `enter` method. A reference count is increased when a *ScopedMemory* is entered and decreased when it is exited (when the `enter` method terminates its execution). The `enter` method clears the *ScopedMemory* area it exits if the reference count reaches zero.

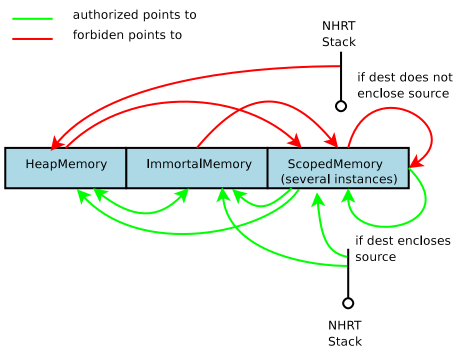

Figure 1. The RTSJ Memory model

The *ScopedMemory* areas provide real-time developers means to allocate memory with a linear time complexity which makes it deterministic (it only depends on the size of the allocated object). In addition, because the entire area is reclaimed at once, those areas are fragmentation-free. This device is much more powerful than usual stack allocation mechanisms because it allows sharing of scope between threads.

The constraints that the RTSJ imposes are illustrated figure 1. Those constraints prevent objects from pointing to shorter lived objects which prevents dangling pointers. They also prevent NHRT from having a reference toward a heap-allocated object in their execution stack. Those constraints are enforced by runtime checks that raise an exception whenever a memory model violation is encountered. Those checks are called respectively assignment checks and access checks.

The model, however, does not ¡prevent *ImmortalMemory* and *ScopedMemory* to contain pointer to heap-objects.

3.2 Concurrent Allocation

The first phase of a GC cycle consists in scanning a set of root pointer to discover live-entry-points to heap objects. In Mackinac, root scanning is complicated by the fact that NHRTs can concurrently allocate objects in ImmortalMemory or in a ScopedMemory area. Because of this, the GC might encounter objects which header is not initialized and has to skip them (they cannot contain any root). The header is used by the GC to recover the object size through its class. In the absence of header, we take profit from the fact that the memory areas are contiguous chunks of memory and that the Java Programming Language requires that allocated objects are filled with zeros. We guarantee this last point by pre-initializing all memory areas with zeros. We do not provide the details of the algorithm, but we are able to exactly detect objects concurrently allocated by NHRTs and to skip them.

3.3 Concurrent Object Moving

The GC offered by HotSpot does a compaction of the heap and moves objects to achieve this. Moving an object requires that all references to that object are updated. This includes references from immortal or scoped objects which might be concurrently accessed by NHRTs.

To make this possible we have to prevent sequences where the GC reads a reference, an NHRT over-writes it and the GC re-writes the updated reference immediately after the NHRT. That is, we have to serialize concurrent memory writes.

We guarantee this using atomic compare and swap operations. During the update phase of the compaction, the GC uses a compare and swap to perform all the updates of root pointers that are in the *ImmortalMemory* or in a *ScopedMemory*. In this scheme, NHRTs do not suffer the overhead of compare and swap operations.

3.4 Concurrent ScopedMemory Cleaning

Our solution for concurrent GC update is sufficient for the ImmortalMemory area but the ScopedMemory areas introduce another problem with the concurrent update because they can be exited and cleaned while the GC is running.

There, we prevent sequences where the GC reads a reference `r` at address `a`, an NHRT exits the scoped memory area that contains `a`, re-enters it and populates it with objects in such a fashion that a value that is not a reference but which binary value is equal to `r` is stored at address `a`. The GC has no means to be aware that the value stored at address `a` is not a reference and must not be updated.

We have designed and implemented two solutions to this problem that meet the RTSJ 'no interference on NHRTs' requirement. These solutions embody the classic time vs. space tradeoff, albeit in more of a real-time notion of time.

Solution 1 is a low-level device driver that allows an NHRT to modify the GC update code for a scoped area and clear the GC instruction cache, ensuring that the `CAS(&I.f,O,O')` will not be executed. This solution is less memory intensive but introduces a higher latency jitter.

In solution 2 each *ScopedMemory* area is associated with two spaces. Once the non NHRT threads are stopped, the GC locks the current space. If an NHRT needs to clear a locked space, it just switches to the alternate (empty) space so that objects that it parses are not concurrently replaced by other objects.

The drawback of solution one is that it doubles the space consumed by each *ScopedMemory* area, but its induced latency jitter is much lower than solution one.

We expect to provide both mechanisms in Mackinac and allow the developers to choose the appropriate tradeoff based on application requirements.

4 Low-Latency, Time-Constrained Activities

Meeting the requirement for release jitter of periodic activities to the tens of microseconds required us to pay special attention to the time source used as the basis for time manipulations, and to the software and hardware events that are susceptible to degrade this jitter. This section describes how Mackinac achieves this objective.

4.1 High-Resolution Time Source

Mackinac relies on a unique high-resolution time source for all time-related computations. This time source provides the number of nanoseconds elapsed since the boot of the machine. As the RTSJ requires *AbsoluteTime* objects to represent points in time past the standard Jan, 1, 1970 epoch, a constant offset computed at JVM startup time is then applied. Mackinac also performs all time-related computations according to nanosecond-resolution, absolute values. This eliminates the uncertainty that would be cause by using relative time values to implement the abstraction of absolute times. For instance, the next release time of a periodic activity is never defined as a delay relative to the current time, but as an absolute time that only depends on the activity's start time (itself expressed as a absolute date) and on the activity's period.

4.2 Cyclic Operations

Implementing high-precision, low-latency time-constrained activities (such as periodic threads) require the JVM to closely interact with the underlying operating system. Even if the Solaris operating environment exhibits strong real-time capabilities [7], the regular timer programming interface does not provide the level of temporal performance Mackinac demands. First, this interface features a signal-based interface, and the related signals are not directed to a particular thread, but are sent to the process that created the timer. Signal delivery and execution of the signal handler would be serialized, thus adding to the threads waiting for their release a jitter deemed unacceptable. Blocking on a synchronization object and explicitly relying on a timeout mechanism to be hopefully released on time is also not an option, as the precision of such timeouts is there limited by the frequency of the system's clock, typically 10 ms.

Mackinac thus implements a specific interface for the handling of time-related operations. This interface relies on a dedicated device driver that enables the real-time JVM to use system services that are otherwise restricted to the Solaris kernel.

This device driver is essentially built on top of Solaris kernel's cyclic subsystem [10] which provides low-overhead interval timers whose accuracy is only limited by the underlying hardware.

To achieve this objective, the Solaris cyclic subsystem has been designed to take advantage of processor architectures that have the ability to interrupt based on an absolute timestamp value. On UltraSPARC® processors, this capacity relies on a high-resolution timestamp-compare register that can be reprogrammed without introducing error in the system's notion of time. The subsystem relies on the exact same time source than used by the JVM. It implements per-processor, balanced heaps sorted by cyclic expiration time, so that the cyclic with the earliest expiration time is always root of the tree. All the time-related functions of the Solaris kernel – from the system clock to the user-level timers – are actually consumers of the cyclic subsystem.

The Mackinac cyclic driver is thus essentially another consumer of this subsystem. Not only does it export the existing kernel functionality, but it also implements additional semantics specific to the RTSJ. For instance, the ability to deschedule and later reschedule a periodic thread is directly handled at the driver level. The cyclic driver is also at the root of synchronization operations performed within the Mackinac JVM, as it additionally implements interruptible, high-resolution waits based on absolute timeout values. Such operations are basically implemented as "one-shot" cyclics removed from the cyclic heap after their first expiration.

4.3 Processor Isolation

The Solaris kernel features a number of scheduling classes implemented on top of a base dispatcher. Mackinac takes advantage of the real-time (RT) scheduling class that provides a fixed-priority, FIFO preemptive scheduling policy which is granted the highest range of scheduling priorities on the system. It is then guaranteed that the runnable thread with the highest RT priority is always selected to run before any other thread in the system.

Uncontrolled rescheduling operations may however degrade the overall performances of the real-time activities. The Solaris dispatcher is free to migrate threads where appropriate with respect to criterion of its own. This migration first requires the dispatcher to perform cross-CPU calls, but also implies that the periodic thread will resume its execution on a processor with cold caches. As a consequence, the release jitter of the thread may be negatively impacted, until the processor's caches are appropriately populated.

Mackinac addresses this particular problem by allowing the available processors to be partitioned into *processor sets*. Processor sets are a Solaris feature that allow CPUs to be partitioned into a number of smaller, non-overlapping groups. Processors assigned to a given processor set are reserved for the processes explicitly bound to that set. Other processes not assigned to the processor set cannot use these CPUs.

Binding the Mackinac JVM to a dedicated processor set has a number of consequences. First, other non-real-time processes can not interfere with Mackinac real-time activities, because they just can no longer use the processors assigned to JVM. Mackinac then goes a bit further by also optionally unbinding the non-NHRTs threads from the processor set assigned to the JVM. The unbound threads then only execute on the remaining, non-assigned processors, thus reducing cache thrashing on assigned processors. If Mackinac is actually bound to a single-CPU processor set, this also prevents NHRT threads from being migrated to other processors, thus raising the cache hit rate at their exclusive advantage and minimizing accordingly their release jitter.

We also use this fact to further protect NHRTs from system activities by setting their dedicated processor *no-intr* meaning that they are sheltered from unbound interrupts. Processor in *no-intr* state only receive interruptions that are explicitly targeted to them.

5 Resource Efficient Asynchronous Event Handlers

The RTSJ has decoupled the notion of asynchronous event, instantiated through the *AsyncEvent* class, and the logic associated with the occurrence of this event, instantiated through the *AsyncEventHandler* class. An event can be associated with many handlers and a handler can be associated with several events. Every time an event occurs, all of its associated handlers are released for execution. The thread used to execute the code embodied in a handler is an instance of either *RealtimeThread* or *NoHeapRealtimeThread* depending on parameters given to the *AsyncEventHandler* constructor. We represented figure 2 a typical configuration of instances of *AsyncEventHandlers* and *AsyncEvents* [3]. The arrow between instances represent the referencing direction. The line between AEH and RTT and NHRT represents an indirect association.

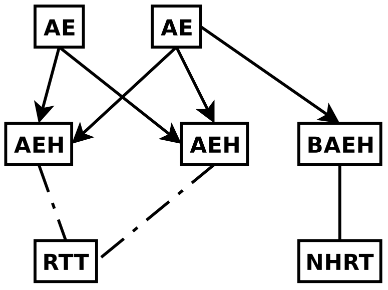

Figure 2. AsynchronousEventHandler Instance Configuration

The RTSJ designers expected thousands or tens of thousands of handlers to be instantiated in a typical application, but only a few of them to be released for execution at the same time. This is why the RTSJ has defined handlers as "lightweight"; because they might be not bound to a native thread, called a server thread, until they have been dispatched to a processor. The RTSJ doesn't specify any particular scheme for the binding between handlers and server threads, and each implementation is free to choose the way to perform it. For applications which have strong requirements for a small release latency for some handlers, the RTSJ has defined the *BoundAsyncEventHandler* class. Each instance of this class has a dedicated server thread, and then, obviously, a simpler, and faster, release process.

Traditional schemes to support aperiodic activities in a real-time systems, like the aperiodic server or the slack stealing algorithms are not applicable because in the RTSJ, handlers have their own time constraints and scheduling properties. Once released, they have to be scheduled like any other instance of *Schedulable*. Associating a thread to each handler is not realistic considering the potentially high number of handlers. Providing handlers with threads when they are released for execution is more reasonable but it raises two issues: the number of server threads can still

[3] we make reference to instances of those classes using the abbreviation, respectively, AE and AEH. The abbreviation BEAH is used to represent instances of the class BoundAsyncEventHandler

be very important if lot of handlers are released at the same time, and the creation of a new thread at each handler's release can introduce jitter in the release latency.

To limit the number of required server threads, Mackinac's binding process uses the fact that when an event is fired, all of the handlers associated with it are released for execution, but because a handler's execution is done according to its scheduling parameters, most of them won't execute immediately. Thus, the binding between handlers and server threads is not performed at release time but at election for execution time. With this property, the number of server threads required to execute handlers is equal to the maximum number of *started* handlers plus one, which is, most of the time, lower than the maximum number of *released* handlers.
To perform this late binding, the released handlers are not directly bound to server threads, but inserted into a sorted queue of all released, but not yet bound, handlers. All of those handlers are sorted according to their execution eligibility, from the highest to the lowest. While that queue is not empty, the virtual machine ensures that a server thread with the execution eligibility of the first handler of the queue is always immediately available.
When this server thread is dispatched, it removes the first handler of the queue and executes its logic. Before executing the handler's logic, the server thread checks if the waiting handler queue is empty. If it isn't, it replaces itself, as a waiting server thread, with a new server thread and gives it the execution eligibility of the new handler at the head of the waiting queue.
With this scheme, two server threads are enough to serve all the handlers if there is no blocking and no preemption by other handlers. But every time a handler blocks or is preempted, it will require a new server thread to ensure that the released but not bound yet handlers will have a server thread immediately available to execute them.

To avoid the jitter that thread creation may introduce in the release latency, Mackinac maintains two pools of server threads, one to serve instances of *RealtimeThread* and the other to serve instances of *NoHeapRealtimeThread*. This separation is required to isolate NHRTs from synchronization with others schedulables that might be impacted by GC activity. Every time the binding process needs a new server thread, it takes it from the corresponding pool. If the pool is empty, then a new server thread is created. When a server thread has completed the execution of a handler, it automatically returns into the pool and goes to sleep until a new thread takes it out of the pool.
How is it possible to get a deterministic behavior with such a dynamic scheme? Simply by specifying to the Mackinac the initial number threads to put in each pool. With the analysis of the application code, to identify blocking handlers, and the scheduling analysis, to compute the number of concurrently running handlers, an application developer can be able to compute the number of server threads the application will needed. Once the pools are populated with the right number of threads, all bindings are performed without thread creation, and thus with a deterministic latency. If the analysis is wrong, and the number of server thread is insufficient, the binding process will still work but the application may suffer larger release latencies than expected due to the creation of new native threads.

6 Predictible Execution Time of Bytecodes

Our goal with respect to bytecode execution is to allow developers to write predictable code while obtaining the best performance for both the real-time and non real-time parts of their application. The most stringent requirement there is requirement 5 which states that any linear section of compiled code must execute in bounded time.

We also need to preserve full Java compliance (see requirement 1) which implies (i) that we should initialize classes on demand and (ii) that we allow dynamic class loading.

We first describe how we handle class loading and initialization and then review how we implement the various bytecodes to give them predictable execution times. Finally, we describe our compilation scheme.

6.1 Class Loading and Initialization

Java bytecodes contain symbolic references to classes and class members that must be resolved at runtime for the bytecode to be executable. Bytecodes that contain a reference to a static class member and object allocation bytecodes may also trigger initialization of classes they reference, before they can be executed.

Those operations are not time-predictable so that for the bytecode to execute in a bounded amount of time, we have to perform those actions on bytecodes in advance, when possible, of their execution.

The Java specification requires that class initialization and resolution failure reporting are made *on-demand*. It doesn't require, however that resolution itself is made *on-demand*.

If the resolution of a symbol fails during the execution of a program, then the resolution of the same symbol will always fail during the same execution of the program. This makes it possible to load and link classes early as long as we do not signal the resolution error until executing the faulty bytecode. In Mackinac if a resolution fails during a pre-resolution phase, then, the faulty bytecode is compiled in a way that will propagate the error if and only if that bytecode is actually executed. By using such early resolution, we can eliminate the jitter due to dynamic class loading.

6.2 Interface Method Call and Runtime Type Checking

The `invokeinterface` bytecode is used to implement method calls for which the static receiver type is an interface [9].

Because the relationship between interfaces is multi-valued, as opposed to inheritance which is single valued, it is not possible to use the classical dispatch mechanism used for the `invokevirtual` bytecode. Much work has been devoted the reduction of the overhead of virtual call in general [8, 1]. All those efforts are heading toward optimizing the common case and have a maximum execution time for the operation that is linear in the number of methods that are present in the receiver class. Because receiver classes cannot be statically inferred, this operation is not temporally predictable

Siebert has proposed an implementation of `invokeinterface` based on a *implements table* that represents the *implements* relationship between classes and interfaces [13]. This structure allows the execution of the `invokeinterface` bytecode and the runtime type checks that involve interfaces in constant time. We implemented in Mackinac an optimized version of *implements-table* that allows Mackinac to limit space consumption as well as execution time.

We have also implemented a similar solution for runtime type checks in such a manner as to allow Mackinac to execute the bytecodes *instanceof, checkcast, aastore* in constant time as well.

6.3 Initialization Time Compilation

The way compilation is performed in non real-time JVMs can cause execution time variations because compilation itself occurs during program execution.

Because the compilation is done at runtime, the only executable code that a VM loads is Java bytecodes. This is at the crux of Java safety because bytecode streams are verifiable in a way such that it is guaranteed that any such verified code executed by a compliant JVM is safe. Thus, for the sake of safety, Mackinac will not provide a static precompiler. Instead, we use a modified version of the standard HotSpot dynamic client compiler that compiles methods of a class during the initialization of the class rather than at execution time.

No method of a class could possibly have been executed before its defining class has been initialized and we have seen that class initialization is not intended to be predictable. Compilation at initialization time thus ensures that code is compiled before it is needed and that compilation will not occur while the application needs predictable execution time.

Because it would not be practical to compile all the methods of an application, Mackinac takes input from the application in the form of a the list of methods that should be compiled at initialization time. Then, during the initialization of a class, the system compiles these methods and, importantly, does not decompile them nor attempt to recompile them during execution.

6.4 Unpredictable Compiler Optimizations

One of the main design guidelines that directed the development of the client HotSpot compiler is the *focus-on-common-case* principle [5]. This principle instructs that any operation that can be split into a common case and an uncommon one must be implemented in that way and that the common case be highly optimized. For instance, virtual method calls are treated this way by the means of *inline caches* [8] where the first method called is considered the most common. In this scheme, monomorphic call sites, which are the most common, are executed without method lookup. However, a backup solution is needed in case the site becomes polymorphic. Backing up involves complex VM operations that causes execution time variability.

To circumvent this, we consider that all call sites are polymorphic and never use *inline caches*. For similar reasons, we also discard other optimizations in the C1 compiler like inlining of small, non final, methods, a typical tradeoff between average-case throughput and predictable execution time.

An important point about Mackinac's compiler scheme is that the JVM may compile a method up to three times, corresponding to the three thread contexts in the RTSJ, **Thread**, *RealtimeThread*, and *NoHeapRealtimeThread*. Each compiled version of the method will have code specifically designed for the particular tradeoffs necessary to optimize the metrics important for that context. For example, NHRTs will have assignment checks but Threads will not. This scheme allows Mackinac to preserve, as much as possible, the throughput for Threads but give the necessary temporal execution predictability to NHRTs.

7 Results

The results on which we comment in this section have been obtained on a Sun Fire V210 which includes two 1GHz UltraSPARC-IIIi processors and 2GB of RAM.

7.1 Latency and Latency Jitter

In order to assess the determinism of cyclic activities, we measure the difference between the theoretical release time and the actual time where the thread starts executing. Because there is a single NHRT at the highest priority, this

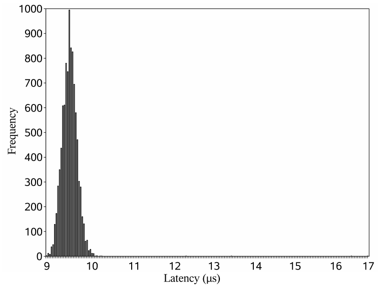

Figure 3. Periodic NHRT release latency(1)

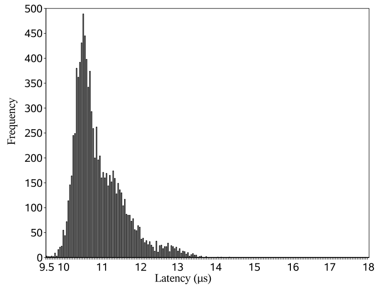

Figure 4. Periodic NHRT release latency(2)

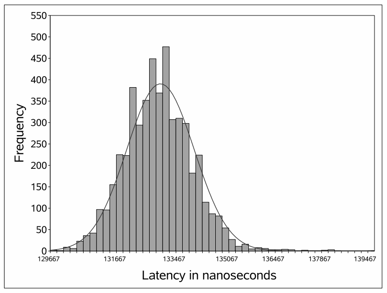

Figure 5. Handler's release latency

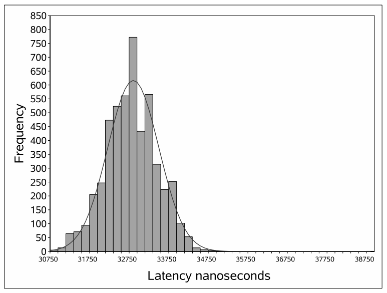

Figure 6. Bound Handler's release latency

procedure measures the latency. We represent, figure 3, the distribution of latencies. Those results show that we satisfy our requirement 4 which states that the latency jitter be in the tens of μs) (below 16.8 μs).

To further test our implementation, we perform the same test with concurrent activities that generate a large amount of garbage memory. The results,which are represented figure 4, show that we also satisfy the requirement that NHRT stay unaffected by the GC(Requirement 2).

The GC load, has a visible effect on the release latency, our investigations showed that the interactions between the GC and NHRTs come from low-level hardware mechanisms such as cross-CPU interrupts and memory hierarchy access management. Still, the release latency remains below 20 μs.

Those tests are run with the processor configured as described section 4.3.

7.2 Asynchronous Event Handler Latency and Latency Jitter

To measure the determinism of a handler's release, we use an NHRT which periodically takes the current time and then fires an event. This event is associated with a single handler which takes the current time at the beginning of its execution. The difference between the two times gives the release latency. We only focus on the release latency in the steady state, first releases are not taken into account so as to eliminate the jitter induced by symbol resolution and server thread creation.

The figure 5 shows release latencies for instances of *AsyncEventHandlers* and *BoundAsyncEventHandlers*. During the measurements considered here, no server thread creation has occurred, thus the binding process shows a deterministic behavior. With unbound handlers, the release latency is under 140 microseconds and the maximum jitter is 20 microseconds. With instances of *BoundAsyncEventHandler*, the maximum latency is less than 39 microseconds. It's only about twice the release latency of a periodic NHRT and it includes all of the additional work required by the specification: management of a fire count to deal with release bursts and verification of aperiodic and sporadic parameters which regulate the acceptance of new releases of handlers.

7.3 Performance of Executed Code

We provide some results on throughput performance of regular Java code and some results on the execution predictability of code compiled for NHRTs.

7.3.1 Throughput of Regular Java Code

The performance criterion for the throughput of **Threads** is given by the requirement 7. Our goal is to maintain the overhead of the changes to Mackinac to below 20% as compared to unmodified HotSpot.

What we seek to understand here is how the throughput performance of non real-time Java applications, executed by

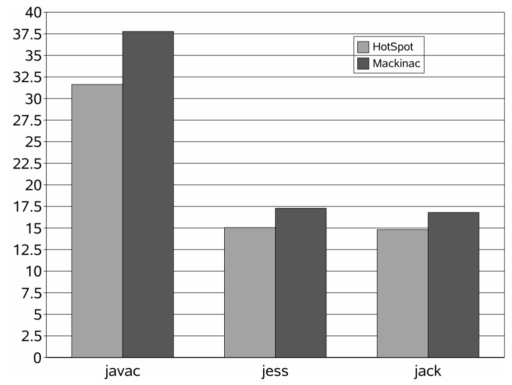

Figure 7. Benchmark Results

a Thread, in Mackinac has been degraded.

We use the some publicly available and other well-known industry standard Java application benchmarks to compare the throughput performance of Mackinac with the throughput performance of unmodifed HotSpot.

The benchmarks have been run on the same version of Hotspot on which the current version of Mackinac is based. This results are graphically represented figure 7. The execution times are reported in seconds.

The first remark is that we managed to stay below the 20% of degradation of throughput for all the benchmarks we ran, including those not shown in figure 7. The overhead of the executed benchmarks varies from a low of 2% to a high of 19.4%. The reasons are multiple. We manage to maintain the overhead very low for computation-oriented benchmarks but the overhead increases as benchmarks manipulate many data structures and strings, e.g., for parsing and internal representations of programs as in `javac`. Also, the `javac` benchmark causes the GC to execute often as it creates much garbage in its operation. We have identified that problems arise from the GC, probably because our version of HotSpot has to be extended to fully support an N generation model instead of only 2 generations. Most of the immortal memory parsing could be avoided for the young collections. We are currently investigating GC extensions to take into account RTSJ specifics.

The biggest issue we faced in preserving the performance of Threads was support for the `synchronized` statement. HotSpot uses a fast-lock mechanism to minimize the lock/unlock cost on uncontended monitors. Unfortunately, because of requirement 3, Java locks must support the Priority inheritance protocol (PIP)[11]. Hence, we were forced to reimplement an enhanced version of PIP.

Value	Thread	RTT	NHRT
Max (ms)	13.6	8.976	9.050
Min (ms)	8.3	7.120	8.913
Average (ms)	8.7	7.125	8.931

Table 1. Jitter on execution times

7.3.2 Predictability of Execution Time for Linear Code Sections

In order to measure the variability of execution time, we use a benchmark that has a workload based on the object oriented features of the Java programming language. It consists of a small interpreter that evaluates expressions of which the internal representation is a tree of Java objects. The benchmark mimics a control law in that it takes into account the results of the previous computations (sequential computations). There is dynamic memory allocation in the computation phase. The benchmark is highly object oriented in the sense that it uses polymorphism and data structures consisting of objects.

In order to assess the variability of the execution time of linear code sections, we compare the execution times of thousand runs of the same logic. The conditions under which the code is executed are stressful for the VM because we artificially trigger GC cycles every 10 milliseconds.

This experiment shows that under such stressful circumstances, the variability of the execution time of the logic of NHRT's is small even in the presence of GC cycles.

The interesting facts are first that the maximum difference of execution time for NHRTs is 137μseconds which represents 1.5% of the minimum execution time. The same figure for regular Threads is as high as 61%. It is actually possible to observe arbitrarily large differences for Threads because jitter is not bounded in that thread context.

RTTs suffer from the same jitter as Threads in Mackinac because we do not, yet, have a real-time GC. The reason why the score of RTT is better than the score of Threads is that RTTs run at real-time priorities with the consequence that they don't give up the processor as easily as Threads. This effect is artifactual and doesn't represent actual variability in the execution time of Threads.

In order to assess the code produced by the compiler for NHRT's, we also measured the exact number of instructions executed during each run. We measured this using the hardware counters available in the SPARC processor. The NHRTs the largest difference that we observed is 12 instructions with a minimum of 6,585,189 instructions executed. This represents only 0.0002% of variation. The difference between execution time and instruction executed is mainly due to cache effects and other hardware effects.

8 Conclusion

Mackinac is the code name for Sun Microsystems' project to implement the Real-Time Specification for Java. We based the Mackinac VM on a production version of Sun's HotSpot Java Virtual Machine and made modifications indicated by the requirements of the RTSJ and also imposed our own requirements.

The main design goal of Mackinac is meant to preserve throughput performance of HotSpot while providing the best real-time performance as possible given the hardware (latency and latency jitter).

An interesting point is that HotSpot has been designed in order to optimize the average execution time (throughput) at the expense of maximum execution time which is exactly the value we have to bound. This fact, together with the requirements imposed on the RTSJ, delineate the design space for Mackinac where tradeoffs are made between predictability on one side and throughput and memory usage on the other side. This includes, for instance, the treatment of *ScopedMemory* concurrent cleaning where the solution which provides the best predictability consumes twice as much memory as the solution with a lower predictability.

We achieved our goal in the sense that we keep release latencies to the few tems of microseconds as well as limit the degradation of average-case throughput to less than 20% for NHRTs.

The original intent of Scoped Memory and NoHeapRealtimeThread was for only those pieces of logic for which absolute minimum jitter, overhead, and latency would suffice. The RTSJ thus expects that real-time GC algorithms will be implemented in various offerings so that the GC-induced jitter on RTTs can be bounded. This will further ease the design of real-time application by allowing a broader use of RTTs and, as such, increase the adoption of the RTSJ.

References

[1] Bowen Alpern, Anthony Cocchi, Stephen Fink, and David Grove. Efficient implementation of Java interfaces: Invokeinterface considered harmless. *ACM SIGPLAN Notices*, 36(11):108–124, November 2001. Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01).

[2] David F. Bacon, Perry Cheng, and V. T. Rajan. Controlling fragmentation and space consumption in the metronome, a real-time garbage collector for java. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 81–92. ACM Press, 2003.

[3] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[4] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 37–48. ACM Press, 2004.

[5] R. Griesemer and S. Mitrovic. *The School of Niklaus Wirth: The Art of Simplicity*, chapter A Compiler for the Java HotSpot Virtual Machine. Morgan Kaufmann Publishers, 2000.

[6] Tuning garbage collection with the 1.4.2 java[tm] virtual machine. http://java.sun.com/docs/hotspot/gc1.4.2/.

[7] Sun Microsystems Inc. Real-time programming and administration. *Solaris 10 Programming Interface Guide*, 2005.

[8] Junpyo Lee, Byung-SunYang, Suhyun Kim, Kemal Ebcioglu, SeungIl Lee, Yoo C. Chung, Heungbok Lee, Erik Altman, Je Hyung Lee, and Soo-Mook Moon. Reducing virtual call overheads in a java vm just-in-time compiler. *ACM SIGARCH Computer Architecture News*, 28(1):21–33, march 2000.

[9] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, January 1997.

[10] Eric Schrock. Inside the cyclic subsystem. http://blogs.sun.com/roller/page/eschrock/20041127, nov 2004.

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

[12] Fridtjof Siebert. Hard real-time garbage collection in the Jamaica Virtual Machine. In *Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, Hong Kong, 1999.

[13] Fridtjof Siebert and Andy Walter. Deterministic execution of Java's primitive bytecode operations. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '01)*, Monterey, CA, April 2001.