The Design and Performance of Real-Time Java Middleware

Angelo Corsaro, *Student Member*, *IEEE*, and Douglas C. Schmidt, *Member*, *IEEE*

Abstract—More than 90 percent of all microprocessors are now used for real-time and embedded applications. The behavior of these applications is often constrained by the physical world. It is therefore important to devise higher-level languages and middleware that meet conventional functional requirements, as well as dependably and productively enforce real-time constraints. This paper provides two contributions to the study of languages and middleware for real-time and embedded applications. We first describe the architecture of jRate, which is an open-source ahead-of-time-compiled implementation of the RTSJ middleware. We then show performance results obtained using RTJPerf, which is an open-source benchmarking suite that systematically compares the performance of RTSJ middleware implementations. This paper shows that, while research remains to be done to make RTSJ a bullet-proof technology, the initial results are promising. The performance and predictability of jRate provides a baseline for what can be achieved by using ahead-of-time compilation. Likewise, RTJPerf enables researchers and practitioners to evaluate the pros and cons of RTSJ middleware systematically as implementations mature.

Index Terms—Real-time middleware, real-time Java, QoS-enabled middleware platforms, object-oriented languages, real-time resource management, performance evaluation.

1 Introduction

1.1 Current Challenges

The vast majority of all microprocessors are now used for embedded systems, in which computer processors control physical, chemical, or biological processes or devices in real-time. Examples of such systems include telecommunication networks (e.g., wireless phone services), telemedicine (e.g., remote surgery), manufacturing process automation (e.g., hot rolling mills), and defense applications (e.g., avionics mission computing systems). These real-time embedded systems are increasingly being connected via wireless and wireline networks.

Designing real-time embedded systems that implement their required capabilities are dependable and predictable, and are parsimonious in their use of limited computing resources is hard; building them on time and within budget is even harder. Moreover, due to global competition for marketshare and engineering talent, many companies are now also faced with the problem of developing and delivering new products in short timeframes. It is therefore essential that the production of real-time embedded systems can take advantage of languages, middleware, tools, and methods that enable higher software productivity, without unduly degrading the quality of service (QoS).

1.2 The State of the Art

Many real-time embedded systems are still developed in C, and increasingly in C++. While writing in C and C++ is more productive than assembly code, they are not the most productive or error-free programming languages. A key source of errors in C/C++ stems from their *memory management* mechanisms, which require programmers to allocate and deallocate memory manually. Moreover, C++ is a feature rich, complex language with a steep learning curve, which makes it hard to find and retain experienced real-time embedded developers who are trained to use it well.

Real-time embedded software should ultimately be synthesized from high-level specifications expressed with domain-specific modeling tools [1]. Until those tools mature, however, a considerable amount of real-time embedded software still needs to be programmed by software developers. Ideally, these developers should use programming languages and middleware that shield them from many accidental complexities, such as type errors, memory management, real-time scheduling enforcement, and steep learning curves. Java [2] has become an attractive choice because of its rapidly growing programmer base, its simplicity, its safety, and especially for its cheaper maintenance cost when compared to C/C++. Conventional Java implementations are unsuitable for developing real-time embedded systems, however, mostly due to the fact that 1) the scheduling of Java threads is purposely underspecified, 2) Java is a garbage collected language and most of the precise garbage collectors known in literature [3] are not suitable for real-time systems, and 3) Java provides coarse-grained control over memory allocation and access, i.e., it allows applications to allocate objects on the heap, but provides no control over the type of memory in which objects are allocated.

To address these problems, the Real-Time Java Experts Group has defined the RTSJ [4], which provides the following capabilities:

- *A. Corsaro is with the Department of Computer Science and Engineering, Washington University, 1 Brookings Drive, Box 1045, St. Louis, MO 63130. E-mail: corsaro@cse.wustl.edu.*
- *D.C. Schmidt is with the Institute for Software Integrated Systems, Vanderbilt University, Nashville, 1829, Station B, Vanderbilt University, Nashville, TN 37235. E-mail: d.schmidt@vanderbilt.edu.*

Manuscript received 8 Dec. 2002; revised 5 Aug. 2003; accepted 5 Aug. 2003. For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 118752.

- New memory management models that can be used in lieu of garbage collection.
- Access to raw physical memory.
- A higher resolution time granularity suitable for real-time systems.
- Stronger guarantees on thread semantics when compared to regular Java, i.e., the most eligible runnable thread is always run.

Until recently, there were no implementations of the RTSJ, which hampered the adoption of Java in real-time embedded systems. It also hampered systematic empirical analysis of the pros and cons of the RTSJ programming model. Several implementations of RTSJ are now available, however, including the RTSJ RI from TimeSys [5] and jRate from Washington University.

This paper significantly extends our earlier work [6], [7] by providing:

- More extensive coverage of jRate capabilities, such as its memory regions implementation and its support for custom allocators.
- A detailed analysis and comparison of scoped memory that quantifies the trade offs associated with different types of scoped memory.
- Extensive new test results based on the commercial version of Linux/RT from TimeSys, which provides many features (such as higher resolution timer, support for priority inversion control via priority inheritance and priority ceiling protocols, and resource reservation) not supported by the GPL version of the TimeSys Linux/RT.

The main contribution of this paper is to describe the techniques used by jRate to implement the RTSJ middleware, as well as to illustrate empirically the performance pitfalls and trade offs that certain RTSJ design decisions incur.

1.3 Paper Organization

The remainder of the paper is organized as follows: Section 2 provides a brief overview of the RTSJ; Section 3 describes the architecture and design rationale of jRate; Section 4 analyzes the empirical results obtained by benchmarking jRate and the TimeSys RTSJ RI using our RTJPerf [6] benchmarking suite; Section 5 compares our work on jRate with related research; and Section 6 summarizes our work and outlines future plans for improving the next generation of RTSJ middleware for real-time embedded applications.

2 The Real-Time Specification for Java

The RTSJ extends the Java API and refines the semantics of certain constructs to support the development of real-time applications. In the reminder of this section, we will provide an overview of the RTSJ key features.

2.1 Memory

The RTSJ extends the Java memory model by providing memory areas other than the heap. These memory areas are characterized by the anticipated lifetime of the contained objects (immortal, scoped) as well as the time taken for allocation (linear, variable).

Objects allocated within the (singleton) *Immortal Memory* have the same lifetime as the application: They are never collected. Each *Scoped memory* area is equipped with a reference count of the number of threads active in its area. The lifetime of objects allocated in such an area is keyed to the reference count.

Additionally, scoped memory areas provide bounds on the allocation time; currently, variable (`VTMemory`) and linear-time (`LTMemory`) allocators are accommodated. For linear allocation time, the RTSJ requires that the time needed to allocate the $n > 0$ bytes to hold the class instance must be bounded by a polynomial function $f(n) \leq Cn$ for some constant $C > 0$.[1]

For Java Virtual Machine (JVM) and application developers alike, scoped memory is one of the more interesting features added to Java by the RTSJ. Objects allocated within a scoped memory are not garbage collected individually; instead, a reference-counting mechanism detects when all objects in a scope should be collected. Safety of scoped memory areas is ensured by reliance upon 1) a set of rules imposed on entrance of scoped memories, and 2) a set of rules that govern the legality of reference between objects allocated in different memory areas.

2.2 Threads

The RTSJ extends the existing Java threading model with two new types of real-time threads: `RealtimeThread` and `NoHeapRealtimeThread`. The `NoHeapRealtimeThread` can have an execution eligibility higher than the garbage collector. A `NoHeapRealtimeThread` can therefore neither allocate nor reference any heap objects. The scheduler controls the *execution eligibility* of the instances of this class by using the `SchedulingParameters` associated with it.

2.3 Scheduling

The RTSJ introduces the concept of a `Schedulable` object. The execution of `Schedulable` entities is managed by the scheduler that holds a reference to them. The RTSJ provide a scheduling API that is sufficiently general to implement commonly used scheduling algorithms, such as Rate Monotonic (RM), Earliest Deadline First (EDF), Least Laxity First (LLF), Maximum Urgency First (MAU), etc. However, the only required scheduler for a RTSJ-compliant implementation is a priority preemptive scheduler that can distinguish 28 different priorities.

2.4 Asynchrony

The RTSJ defines mechanisms to bind the execution of program logic to the occurrence of internal and/or external events. In particular, the RTSJ provides a way to associate an asynchronous event handler to some application-specific or external events. There are two types of asynchronous event handlers defined in RTSJ:

- The `AsyncEventHandler` class, which does not have a thread permanently bound to it—nor is it

1. This bound does not include the time taken by an object's constructor or a class's static initializers.

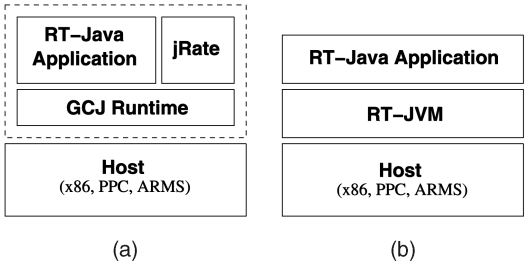


Fig. 1. The jRate architecture.

guaranteed that there will be a separate thread for each `AsyncEventHandler`. The RTSJ simply requires that, after an event is fired, the execution of all its associated `AsyncEventHandlers` will be dispatched.

- The `BoundAsyncEventHandler` class, which has a real-time thread associated with it permanently. The associated real-time thread is used throughout its lifetime to handle event firings.

Event handlers can also be specified a *no-heap*, which means that the thread used to handle the event must be a `NoHeapRealtimeThread`.

The RTSJ also introduces the concept of *Asynchronous Transfer of Control* (ATC), which allows a thread to asynchronously transfer the control from a locus of execution to another.

2.5 Time and Timers

Real-time embedded systems often use timers to perform certain actions at a given time in the future, as well as at periodic future intervals. For example, timers can be used to sample data, play music, transmit video frames, etc. The RTSJ provides two types of timers:

- `OneShotTimer`, which generates an event at the expiration of its associated time interval and
- `PeriodicTimer`, which generates events periodically.

`OneShotTimers` and `PeriodicTimers` events are handled by `AsyncEventHandlers`. The RTSJ also supports high resolution timers and high resolution clocks.

3 jRate Overview

jRate is an open-source RTSJ-based real-time Java implementation that we are developing at Washington University. jRate extends the open-source GNU Compiler for Java (GCJ) front-end and runtime system [8] to provide an ahead-of-time compiled middleware platform for developing RTSJ-compliant applications. One of jRate's key research goals is that of using Generative Programming (GP) [9] in order to make it possible to have a configurable, customizable, and yet efficient RTSJ implementation. The jRate architecture shown in Fig. 1a differs from the JVM model shown in Fig. 1b since there is no JVM interpreting the Java bytecode. Instead, jRate compiles RTSJ applications into native code. The Java and RTSJ services, such as garbage collection, real-time threads, and scheduling, are accessible via the GCJ and jRate runtime systems, respectively.

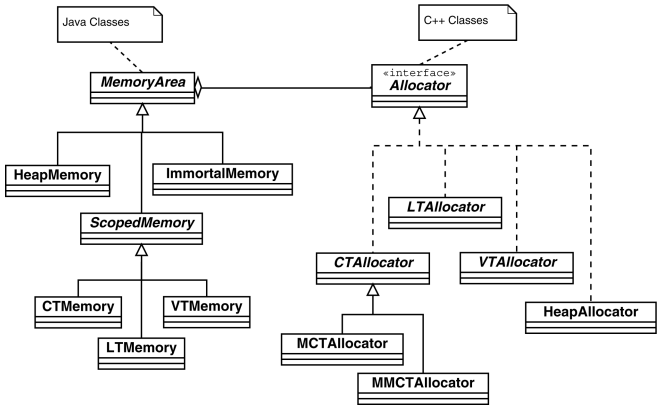


Fig. 2. The jRate memory region structure.

jRate supports most of the RTSJ features described in Section 2. We describe these features below and indicate where the design and performance of these features is discussed in subsequent sections of this paper.

3.1 Memory Areas

jRate supports scoped memory and immortal memory. Fig. 2 shows how these memory areas are implemented in jRate. This diagram shows how each memory area is associated with an allocator. There are two parallel class hierarchies: one set of Java classes for the memory areas and one set of C++ classes for the allocators. The memory management strategy is delegated to native allocators and the binding between memory area and type of allocator can be deferred until creation time. This design provides users with the flexibility to experiment with different allocators strategies and to choose the type of allocator that best fits their application usage patterns.

jRate also provides a strategy that enables users to decide which type of memory (such as linear time memory or variable time memory) should implement immortal memory. Although the RTSJ does not mandate how immortal memory is implemented, we believe it is important to allow users to specify which type of implementation to configure. jRate's immortal memory implementation can be configured when an application is launched. Its scoped memory implementation also exposes a nonstandard extension that uses nonthread safe allocators to avoid the overhead of unnecessary locks if a memory area is always accessed by a single thread.

Fig. 2 illustrates a new type of scoped memory—called `CTMemory`—provided by jRate. `CTMemory` trades off allocation time for the memory area creation time. This memory area is zeroed at initialization time and the amount used is also zeroed each time the memory reference count drops to zero.[2] This feature provides constant time allocation for objects created within the `CTMemory`, which is useful for real-time applications. The structure of `CTMemory` is depicted in Fig. 4. The *type* field distinguishes different types of objects. Different types of objects must be treated differently, e.g., some must be finalized, whereas

2. The reference count associated with a scoped memory is represented by the number of real-time threads in it that are currently *active*, i.e., have entered the scoped memory but have not yet exited.

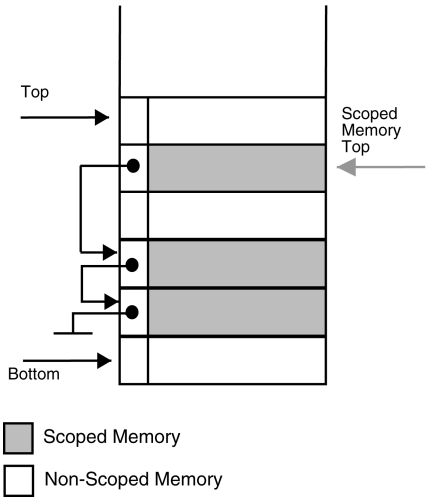


Fig. 3. The jRate scope stack structure.

others need not be finalized. jRate, as described in [10], is currently the only RTSJ implementation which implements a constant time algorithm for checking the validity of memory references. The performance of jRate scoped memory implementation is analyzed in Section 4.3.1.

3.2 Real-Time Threads and Scheduling

jRate supports real-time threads of type `RealtimeThread` using a priority preemptive scheduler based on the underlying real-time OS priority preemptive scheduler—jRate's threads are directly mapped to native OS threads. An interesting characteristic of the RTSJ is that each `RealtimeThread` is associated with a scope stack that 1) keeps track of the set of memory regions that have been entered by the thread and 2) can detect cycles in the entered scopes. jRate's scope stack implementation uses data structures which allow to perform all scope stack operations in constant time.

For example, the `findFirstScope()` operation defined in the RTSJ, scans the scope stack from top to bottom and returns the first scoped memory area found. If implemented as suggested by the RTSJ specification, this operation would have a time complexity of $O(n)$, where n is the length of the stack. jRate enhances the stack data structure suggested in the RTSJ by maintaining a linked list of the scoped memory in the stack, along with the index of the topmost scoped memory, as shown in Fig. 3. This design allows a constant time implementation of both `findFirstScope()`, `push()`, and `pop()`. In fact, `findFirstScope()` simply has to return the value of the pointer to the top scoped memory, while `push` and `pop` have, respectively, to detect if the memory area being pushed or popped is a scoped memory and, if so, update the top pointer for the scoped memory and the pointer in the linked list. Something else that is worth noticing is that jRate avoids using *instanceof* in order to determine the type of a the memory area being pushed or popped since this is usually implemented as a linear time operation in the height of the class hierarchy. To have a constant time type identification, jRate uses its own type encoding for all those classes that often require an *instanceof*.

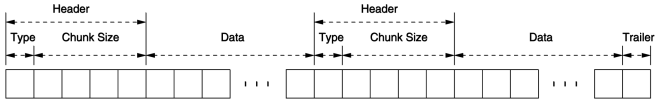


Fig. 4. The jRate `CTMemory` structure.

When a scope stack is destroyed and the reference counts of all scoped memory areas still on the stack must be decremented, jRate knows exactly which entries are scoped memory and which are not. This knowledge enables it to elide a test on the type of memory area and avoids blindly searching for scoped memories on the stack.

The size of the jRate scope stack is fixed after the real-time thread is created. This design makes jRate more efficient by avoiding the use of pointers to implement the linked list. The performance of jRate thread implementation is analyzed in Section 4.3.2.

3.3 Asynchrony

jRate provides a robust and efficient asynchronous event handling implementation that avoids priority inversion and provides lock free dispatch on most platforms.[3] jRate uses priority queues ordered by the execution eligibility of the handlers to dispatch asynchronous events. Execution eligibility is the ordering mechanism used throughout jRate, e.g., it is used to achieve total ordering of schedulable entities whose QoS are expressed in various ways. The performance of jRate asynchrony implementation is analyzed in Section 4.3.3.

3.4 High Resolution Time and Clock

jRate implements the RTSJ high resolution time API. Different implementations of real-time clocks are provided. Depending on the underlying hardware and OS platform, resolution from nanoseconds up to microseconds can be obtained. In particular, on Pentium platforms, high resolution time with a resolution close to the processor frequency is obtained by using the read time stamp counter (RDTSC) register.

3.5 Timers

jRate implements periodic and one-shot timers in accordance to the RTSJ. A thread is associated with each timer (the RI takes a similar approach). Periodic timers are implemented by relying on the behavior provided by periodic threads, where as one-shot timers use a real-time thread with custom logic to generate the event at the right time. The priority of the thread is inherited by the priority of the most eligible handler registered with the timer. An analysis of the performance of the jRate timers implementation appears in [6].

4 Performance Results and Analysis

This section first describes our real-time Java testbed and outlines the various Java implementations used for the tests. We then present and analyze the results obtained running most of the RTJPerf tests [6] in our testbed.

3. On certain platforms, such as Compaq Alpha, the assumptions that we rely upon to avoid locking do not hold, so for those platforms jRate must use locks.

4.1 Overview of the Hardware and Software Testbed

The test results reported in this section were obtained on an Intel Pentium III 733 MHz with 256 MB RAM, running Linux RedHat 7.3 with the TimeSys Linux/RT 3.1 kernel [11]. This is the TimeSys Linux/RT NET commercial version that implements priority inheritance protocols, high resolution timers, and a resource kernel that supports resource reservation. The Java platforms used to test the RTSJ features are described below:

1. *TimeSys RTSJ RI:* TimeSys has developed the official RTSJ RI [5], which is a fully compliant implementation of Java [2], [12] that implements all the mandatory features in the RTSJ. The RI is based on a Java 2 Micro Edition (J2ME) (JVM) and supports an interpreted execution mode, i.e., there is no just-in-time (JIT) compilation. The RI runs on any Linux platform and its threading model directly maps Java threads onto Linux POSIX threads.
2. jRate: As described in Section 3, jRate is an open-source RTSJ-based extension of GCJ front-end and runtime systems that we are developing at Washington University.

4.2 Compiler and Runtime Options

The following options were used when compiling and running the RTJPerf benchmarks.

3. *TimeSys RTSJ RI*:The Java code for the tests was compiled with **jikes** [13] using the `-O` option. The TimeSys RTSJRI Java Virtual Machine (JVM) was always run using the `-Xverify:none` option. The environment variable that controls the size of the immortal memory was set as `IMMORTAL_SIZE=9000000`.
4. jRate: The Java code for the test was compiled with GCJ with the `-O3` flag and statically linked with the GCJ and jRate runtime libraries. The immortal memory size was set to the same value as the RI. jRate v0.3 was used along with GCJ 3.2.1.

4.3 RTJPerf Benchmarking Results

This section presents a description and the results obtained for the RTJPerf tests we ran. We analyze the results and explain why the TimeSys RI and jRate RTSJ implementations performed differently.[4]

We provide average and worst-case behavior, along with dispersion indexes, for all the RTSJ Java features we measured. For certain tests, we provide sample traces that are representative of all the measured data. The measurements performed in the tests reported in this section are based on *steady state* observations, where the system is run to a point at which the transitory behavior effects of *cold starts* are negligible before executing the tests.

4.3.1 Memory Benchmark Results

Below we present and analyze the results of the RTJPerf memory benchmarks that we ran.

Allocation Time Test: Dynamic memory allocation is forbidden or strongly discouraged in many real-time embedded systems to minimize memory leaks, latency, and nonpredictability. The scoped memory specified by the RTSJ is designed to provide a relatively fast and safe way to allocate memory that has nearly the flexibility of dynamic memory allocation, but the efficiency and predictability of stack allocation. The measure of the allocation time and its dependency on the size of the allocated memory is a good measure of the efficiency of various types of scoped memory implementations.

To measure the allocation time and its dependency on the size of the memory allocation request, RTJPerf provides a test that allocates fixed-sized objects repeatedly from a scoped memory region whose type is specified by a command-line argument. To control the size of the object allocated, the test allocates an array of bytes. It is possible to determine the allocation time associated with each type of scoped memory by running this test with different allocation sizes.

1. *Test Settings*: To measure the average allocation time incurred by the RI implementation of `LTMemory` and `VTMemory`, we ran the RTJPerf allocation time test for allocation sizes ranging from 32 to 16,384 bytes. Each test samples 1,000 values of the allocation time for the given allocation size. This test also measured the average allocation time of jRate's `CTMemory` implementation described in Section 3.1.
2. *Test Results*: The data obtained by running the allocation time tests were processed to obtain an average, dispersion, and worst-case measure of the allocation time. We compute both the average and dispersion indexes since they indicate the following information:
   - how predictable is the behavior of a scope memory implementation,
   - how much variation in allocation time can occur, and
   - how the worst-case behavior compares to the average-case and to the case that provides a 99 percent upper bound.[5]

   Fig. 5 shows the resulting average allocation time for the different test runs, and it also shows the standard deviation of the allocation time measured in the various test settings.
3. *Results Analysis*: We now analyze the results of the tests that measured the average and worst-case allocation times, along with the dispersion for the different test settings:
   - **Average Measures**—As shown in Fig. 5, both `LTMemory` and `VTMemory` provide linear time allocation with respect to the allocated memory size. Since similar results were found for other measured statistical parameters, we infer that the RI implementation of `LTMemory` and `VTMemory` are similar, so we focus primarily

4. Explaining certain behaviors requires inspection of the source code of a particular Java Virtual Machine (JVM) feature, which is not always feasible for Java implementations that are not open-source.

5. By "99 percent upper bound," we mean that value that represents an upper bound for the measured values in the 99th percentile of the cases.

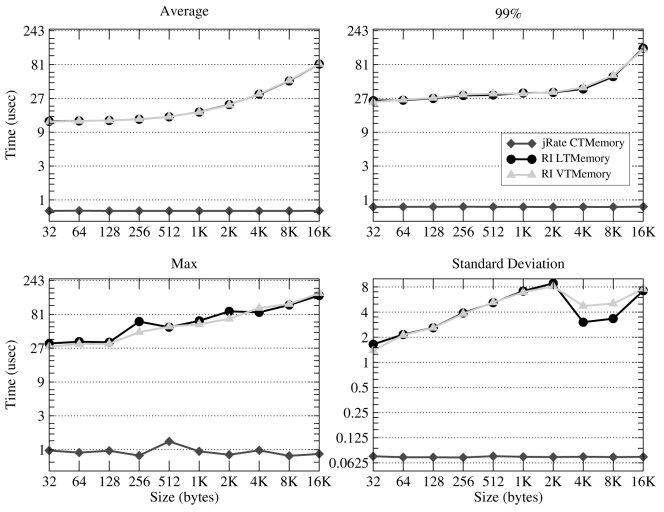


Fig. 5. Scoped memory allocation time statistics.

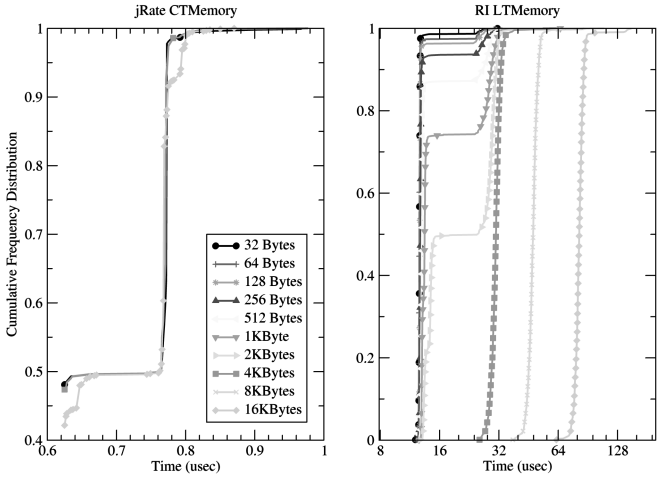


Fig. 6. Allocation time cumulative relative frequency distribution.

on the `LTMemory` since our results also apply to `VTMemory`. jRate has an average allocation time that is independent of the allocated chunk, which helps analyze the timing of RTSJ code, even without knowing the amount of memory that will be needed. From Fig. 5, it can be easily seen that for small memory chunks the jRate memory allocator is nearly 20 times faster than RI's `LTMemory`. For the largest chunk we tested, jRate's `CTMemory` is ~ 120 times faster RI's LTMemory.

- **Dispersion Measures**—The standard deviation of the different allocation time cases is shown in Fig. 5. This deviation increases with the chunk size allocated for both `LTMemory` and `VTMemory` until it reaches 4 Kbytes, where it suddenly drops and then it starts growing again. On Linux, a virtual memory page is exactly 4 Kbytes, but when an array of 4 Kbytes is allocated, the actual memory is slightly larger in order to store freelist management information. In contrast, the `CTMemory` implementation has the smallest variance and the flattest trend.

  The plots in Fig. 6 show the cumulative relative frequency distribution of the allocation time for some of the different cases discussed above. These plots illustrate how the allocation time is distributed for different types of memory and different allocation sizes. For any given point t on the x axis, the value on the y axis indicates the relative frequency of allocation time for which $AllocationTime \leq t$. The standard deviations shown in Figs. 6 and 5 provide insights on how the measured allocation time is dispersed and distributed. It is interesting to note that the cumulative frequency distribution for jRate is S-shaped. Moreover, the values are mostly concentrated in the two flat regions of the "S." The shape of distribution in Fig. 6 reveals a characteristic of jRate's allocator implementation. The GCJ runtime requires all the objects to be allocated at the 8-byte boundaries.[6] As a result, padding may need to be computed, depending whether the size of the object plus the size of the header is a multiple of 8 or not (see Fig. 4). The two regions in the cumulative distribution show the two different cases.

- **Worst-Case Measures**—Fig. 5 shows the bounds on the allocation time for jRate's `CTMemory` and the RI `LTMemory`. Each of these graphs depicts the average and worst-case allocation times, along with the 99 percent upper bound of the allocation time. Fig. 5 illustrates how the worst-case execution time for jRate's `CTMemory` is at most ~ 1.8 times larger than its average execution time. Fig. 5 shows how the maximum, average, and the 99 percent case for the RI `LTMemory` converge as the size of the allocated chunk increases. The minimum ratio between the worst and average-case allocation times is ~ 1.8 for a chunk size of 16K. Figs. 5 and 6 also characterize the distribution of the allocation time. Fig. 6 shows how, for some allocation sizes, the allocation time for the RI `LTMemory` is centered around two points.

Scoped Memory Lifetime Test: Scoped memory is one of the key features introduced by the RTSJ. It enables applications to circumvent the garbage collector, yet still use automatic memory management, by 1) associating with each memory scope a reference count that depends on the number of real-time threads within the memory area (i.e., that have entered the scope but yet not exited it), and 2) ensuring that all the objects allocated in the scope are finalized and the space reclaimed, as soon as the reference count associated with the memory area drops to zero. Since most RTSJ applications use scoped memory heavily, it is essential to characterization its performance precisely.

RTJPerf provides a test that measures 1) the time needed to create a memory scope, 2) the time needed to enter it, and 3) the time needed to exit it. The time needed to create a scoped memory area depends on the following factors:

6. This is done because the lowest three bits are used to store locking information.

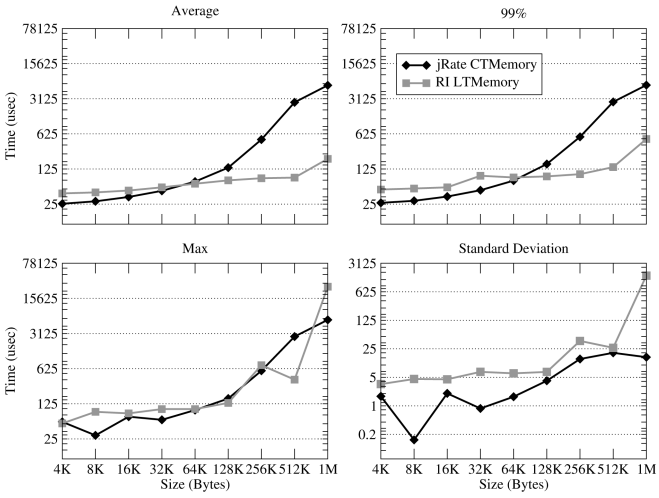


Fig. 7. Average, 99 percent, maximum, and standard deviation of the creation time (μsec).

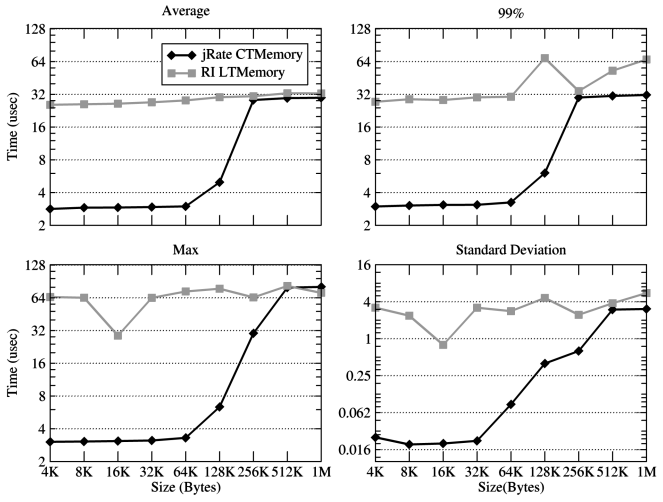


Fig. 8. Average, 99 percent, maximum, and standard deviation of the enter time (μsec).

- The allocation context of the thread that creates the memory scope. The Allocation Time Test measures this aspect of memory scope creation time.
- The native C/C++ implementation of scoped memory. The Scoped Memory Lifetime Test measures the efficiency and predictability of the native C/C++ implementation of scoped memory.[7]

The time needed to exit a memory scope is measured by the case in which its reference count drops to zero as a result of the thread exiting the scope. In this case, the memory scope must finalize all the objects allocated within it and reclaim the used storage.

To determine the time needed to enter, exit, and create a memory scope—and to determine how efficient the implementation is—this test creates a memory scope, enters it, fills it with objects, and then exits the scope. The test can be run by configuring the type of scoped memory to be used and by having the object allocated selectively override the default `finalize` method. Measuring this latter point is important since some Java implementations are smarter than others in handling the case where an object does not override the finalizer.

4. *Test Settings*: To measure scoped memory creation, enter, and exit time, we ran the RTJPerf scoped memory timing test for memory sizes ranging from 4,096 to 1,048,576 bytes. The test was designed to ensure that the allocated objects overrode the finalizer, which enabled a worst-case measurement of the exit time. For each test, 500 values were sampled for each of the measured variables. This test was run to get the relevant measures for the RI's `LTMemory` and jRate's `CTMemory`.
5. *Test Results*: Figs. 7, 8, and 9 contain the average, 99 percent, maximum, and standard deviation trend for memory scope creation, enter, and exit time. Fig. 10 shows the execution time, measured as the time needed to fill the memory with the objects.

7. The memory used by the scoped memory to allocate an object is not retrieved by the current allocation context, but is allocated in a platform-specific way, e.g., using `malloc()` or `mmap()`.

6. *Results Analysis*: Below, we analyze the results of the test that measure the creation, enter, and exit time for a scoped memory area.

- **Average Measures**—From Fig. 7, we can see how the jRate `CTMemory` has better creation times, on average, than the RTSJ RI for scoped memory with size less or equal than 64 KBytes. After this point, jRate starts performing worse than the RI. The explanation of this behavior stems from the fact that the `CTMemory` maps the Linux `/dev/zero` device and then locks the allocated chunk. Simply locking the memory does not reserve physical memory for the calling process; however, since the pages might be copy-on-write. To improve the predictability of an application, therefore, it is usually a good habit to make sure that at least a byte each page of the locked chunk of memory is accessed. The time taken to write at least one byte per each

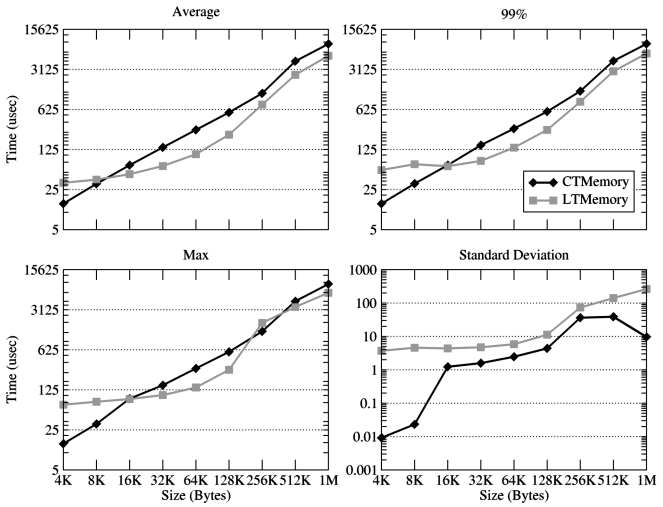


Fig. 9. Average, 99 percent, maximum, and standard deviation of the exit time (μsec).

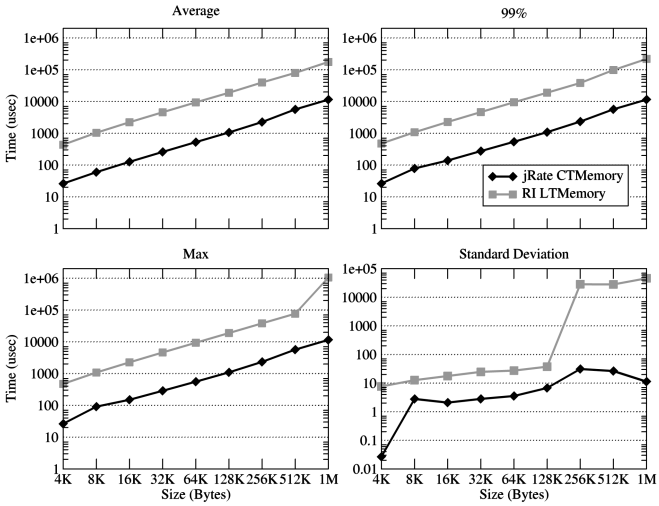


Fig. 10. Average, 99 percent, maximum, and standard deviation of the execution time (μsec).

allocated page, has a time complexity of $O(n)$ in the size of the scoped memory allocated, which makes creation time depend linearly on memory size.

Fig. 8 shows the enter time trend for jRate's `CTMemory` and for the RI's `LTMemory`. From this diagram, we can see that, while the RI enter time varies only slightly with the scoped memory size, for jRate, the enter time depends more on the size. This behavior stems from the fact that jRate's scoped memory puts pressure on the cache and induces cache misses in succeeding instructions, which degrades the execution time of the first enter (which is the one measured by the test).

Fig. 9 shows the results for the exit time. From this diagram, we see that, even if jRate's `CTMemory` has to zero the allocated memory other than finalizing the allocated objects, its performance is close to the RI. Exit time depends primarily on the efficiency of JNI and on the efficiency of the native implementation that manages the scope stack, the finalization, and the cleanup of the memory, if any is needed. In this case, jRate's ahead-of-time compilation model plays a minor role, especially as the size of the memory grows.

- **Dispersion Measures**—Figs. 7, 8, 9, and 10 show how jRate and the RI have very low dispersion values for small memory sizes. These dispersion values grow with the size of the scoped memory for both implementations. The RI becomes less predictable as the size of the memory scope increases, culminating in pathological cases shown in Figs. 7 and 10.
- **Worst-Case Measures**—The results for all the measured variables show that the worst-case measures are very close to the average-case for jRate. In contrast, the RI's worst-case values can be quite large compared to its average-case values. The largest difference between average and worst-case measures appeared in the creation time and in the execution time.

We cannot give a precise answer to the reason of this behavior since the code of the RI was not available for inspection. A reasonable guess, however, is that the RI allocators rely directly on the system provided `malloc()` for each of the allocated objects. This explanation justifies both the relatively small creation time, and also the degradation of the predictability when the allocator creates many objects to fill the scoped memory.

TABLE 1
Yield Context Switch Statistics

	Average	Std. Dev.	Max	99%
jRate	1.449 μs	0.002 μs	1.475 μs	1.459 μs
RI	2.858 μs	0.0198 μs	3.052 μs	2.900 μs
C++	1.30 μs	0.002 μs	N/A	N/A

4.3.2 Thread Benchmark Results

Below, we present and analyze the results from the RTJPerf yield and thread creation latency benchmarks.

Yield Context Switch Test: High levels of thread context switching overhead can significantly degrade application responsiveness and predictability. Minimizing this overhead is therefore an important goal of any runtime environment for real-time embedded systems. To measure context switching overhead, RTJPerf provides a test in which two real-time threads characterized by the same execution eligibility yield to each other. Since there are just two real-time threads, whenever one thread yields, the other thread will have the highest execution eligibility, so it will be chosen to run.

1. *Test Settings:* For each RTSJ platform we evaluated, we collected 1,000 samples of the the context switch time, which we forced by explicitly yielding the CPU. Real-time threads were used for the RI and jRate, and immortal memory was used as allocation context for the threads.
2. *Test Results:* Table 1 reports the average and standard deviation for the measured context switch in the various platforms.
3. *Results Analysis:* Below, we analyze the results of the tests that measure the average context switch time, its dispersion, and its worst-case behavior for the different test settings:

   - **Average Measures**—Table 1 shows how the RI performs fairly well in this test, i.e., its context switch time is only ~ 2 μs larger than jRate's. The main reason for jRate's better performance stems from its use of ahead-of-time compilation. The last row of Table 1 reports the results of a C++-based context switch test described in [14]. The table shows how the context switch time measured for the RI and jRate is similar to that for C++ programs on TimeSys Linux/RT. The context switching time for the RI is less than three times larger than that found for C++,

TABLE 2
Thread Creation Time Statistics

	Average	Std. Dev.	Max	99%
jRate	17.690 μs	3.768 μs	56.526 μs	33.815 μs
RI	890.190 μs	227.260 μs	1306.550 μs	1276.15 μs

TABLE 3
Thread Startup Time Statistical Indexes

	Average	Std. Dev.	Max	99%
jRate	118.671 μs	5.005 μs	169.552 μs	145.878 μs
RI	800.149 μs	29.310 μs	894.578 μs	867.250 μs

whereas the times for jRate are roughly the same as those for C++.

- **Dispersion Measures**—The third column of Table 1 reports the standard deviation for the context switch time. Both jRate and the RI exhibit tight dispersion indexes, indicating that context switching overhead is predictable for these implementations. In general, the context switch time for jRate is as predictable as C++ on our Linux testbed platform.
- **Worst-Case Measures**—The fourth and fifth column of Table 1 represent the maximum and the 99 percent bound for the context switch time, respectively. jRate and the RI have 99 percent bound and worst-case context switching times that are close to their average-case values.

4. *Thread Creation Latency Test:* This test measures the time needed to create a thread, which consists of the time to create the thread instance itself and the time to start it. For this test, RTJPerf provides two variants; here, we refer to the test that creates and starts a real-time thread from another real-time thread. The results we obtained are presented and analyzed below.
5. *Test Settings:* For each platform in our test suite, we collected 1,000 samples of the thread creation time and thread start time. Real-time threads were used for the RI and jRate. Since we are interested in measuring the time taken to create and start a thread—while limiting the effects of other delays—the threads had immortal memory as their allocation context, so to avoid garbage collection overhead.
6. *Test Results:* Tables 2 and 3 report the average, standard deviation, maximum, and 99 percent bound for the thread creation time and thread start time, respectively.
7. *Results Analysis:* Below, we analyze the results of the tests that measure the average-case, the dispersion, and the worst-case for thread creation time and thread start time.

   - **Average Measures**—The second column of Tables 2 and 3 show that jRate has the best average thread creation time and thread start time. The RI's creation time grows linearly in the RT test, which is unusual and may reveal a problem in how the RI manages the scope stack. The smallest creation time experienced by the RI is around 500 μs (which is much higher than jRate) and one reason may be the RI's scope stack implementation.
   - **Dispersion Measures**—The third column of Tables 2 and 3 present the standard deviation for thread creation time and thread start time, respectively. jRate has the smallest dispersion of values for the thread startup time and creation time.

     The standard deviation associated with the thread creation time is influenced by the predictability of the time the memory allocator takes to provide the memory needed to create the thread object. In contrast, the standard deviation of the thread start time depends largely on the OS, whereas the rest of thread start time depends on the details of the thread startup method implementation.
   - **Worst-Case Measures**—The fourth and fifth column of Tables 2 and 3 present the maximum and the 99 percent bound for the thread creation time, and the thread startup time, respectively. These tables clearly show how jRate has a more predictable startup time than creation time. The jitter introduced in the creation time likely stems from the fact that jRate allocates several data structure via `malloc()`, and does not take yet advantage of custom allocators. The RI also has fairly good startup time, though we cannot make any comparison with its creation time due to an RI memory leak exposed by this test. The problem is related to RI's management of the scope stack since it appears only when a `RealtimeThread` is created by another `RealtimeThread`.

Periodic Thread Test: Real-time embedded systems often have activities (such as data sampling and control law evaluation) that must be performed periodically. The RTSJ provides programmatic support for these activities via its APIs for scheduling real-time thread execution periodically. To program this RTSJ feature, an application specifies the proper release parameters and uses the `waitForNextPeriod()` method to schedule thread execution at the beginning of the next period (the period of the thread is specified at thread creation time via `PeriodicParameters`). The accuracy with which successive periodic computation are executed is important since excessive jitter is detrimental to most real-time applications.

RTJPerf provides a test that measures the precision at which the periodic execution of real-time thread logic is managed. This test measures the actual time that elapses from one execution period to the next.

8. *Test Settings:* This test runs a `RealtimeThread` that does nothing but reschedule its execution for the next period. The actual time between each activation was measured and 1,000 of these measurements were made for the periods 1*ms*, 5*ms*, 10*ms*, 50*ms*, 100*ms*, and 500*ms*.
9. *Test Results:* Fig. 11 shows average and dispersion values that we measured for this test.[8]

8. Whenever the plot for jRate and the RI overlap, the values for jRate are shown above the graph and the value for the RI are shown below the graph.

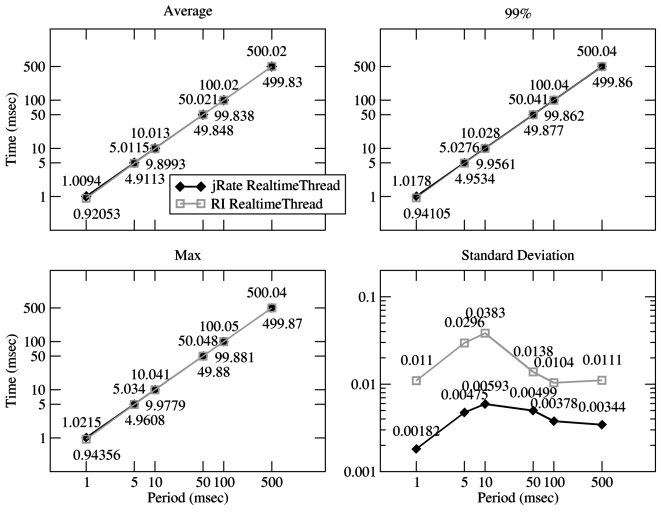


Fig. 11. Measured period statistics.

10. *Results Analysis:* Below, we analyze the results of the test that measures the accuracy with which periodic a thread's logic is activated:

    - **Average Measures**—Fig. 11 shows that both jRate and the RI have an average period that is quite close to the target period. Whereas RI is always at least several hundreds of microseconds early, however, jRate is at most several tens of microseconds late. To understand the reason for this behavior, we inspected the RI implementation of periodic threads, (i.e., at the implementation of `waitForNextPeriod()`), and found that a Java Native Interface (JNI) method call is used to wait for the next period. Without the source for the RI's JVM, it is hard to tell exactly how the native method is implemented. On the TimeSyS Linux/RT kernel, jRate relies on the `nanosleep()` system call to implement periodic thread behavior. To produce more accurate periods, a calibration test can be run at configuration time to obtain a slack time that should be considered as an approximation of the overhead of calling the `waitForNextPeriod()` and then getting the control back.

      As stated near the beginning of Section 4.3, the measure for each test are made in stationary conditions. The Periodic Thread Test was interesting since the RI took a relatively long time to reach the steady state, particularly for small periods.
    - **Dispersion Measures**—Fig. 11 shows the dispersion of the measured period for both jRate and the RI has the same trend. While jRate generally has less dispersed values than the RI, both implementations are quite predictable.
    - **Worst-Case Measures**—As shown in Fig. 11, both jRate and the RI have worst-case behavior that is close to the average-case values and the 99 percent bound. In general, jRate's worst-case values are closer to the average, but the RI values are not much further away.

4.3.3 Asynchrony Benchmark Results

Below, we present and analyze the results of the RTJPerf asynchrony tests that we ran.

Asynchronous Event Handler Dispatch Delay Test: Several performance parameters are associated with asynchronous event handlers. One of the most important is the *dispatch latency*, which is the time from when an event is fired to when its handler is invoked. Events are often associated with alarms or other critical actions that must be handled within a short time and with high predictability. This RTJPerf test measures the dispatch latency for the different types of asynchronous event handlers prescribed by the RTSJ.

1. *Test Settings:* To measure the dispatch latency provided by different types of asynchronous event handlers defined by the RTSJ, we ran the test described above with a fire count of 1,000 for both RI and jRate. To ensure that each event firing causes a complete execution cycle, we ran the test in "lockstep mode," where one thread fires an event and only after the thread that handles the event is done is the event fired again. To avoid the interference of the Garbage Collector (GC) while performing the test, the real-time thread that fires and handles the event uses scoped memory as its current memory area.
2. *Test Results:* Fig. 12 shows the trend of the dispatch latency for successive event firings (since the RI's `AsyncEventHandler` trend is completely off the scale, it is reported in a separate plot in Fig. 12). The data obtained by running the dispatch delay tests were processed to obtain average and worst-case behavior, and the dispersion measure of the dispatch latency. Tables 4 and 5 show the results found for jRate and the RI, respectively.
3. *Results Analysis:* Below, we analyze the results of the tests that measure the average-case and worst-case dispatch latency, as well as its dispersion, for the different test settings:

    - **Average Measures**—Table 5 illustrates the large average dispatch latency incurred by the RTSJ RI `AsyncEventHandler`. The results in Fig. 12 show how the actual dispatch latency increases as the event count increases. By tracing the memory used when running the test using heap memory, we found that not only did memory usage increased steadily, but even invoking the GC explicitly did not free any memory.

      These results reveal a problem with how the RI manages the resources associated to threads. The RI's `AsyncEventHandler` creates a new thread to handle a new event, and the problem appears to be a memory leak in the underlying RI memory manager associated with threads, rather than a limitation with the model used to handle the events. In contrast, the RI's `BoundAsyncEventHandler` performs quite well, i.e., its average dispatch latency is slightly less than twice as large as the average dispatch latency for jRate.

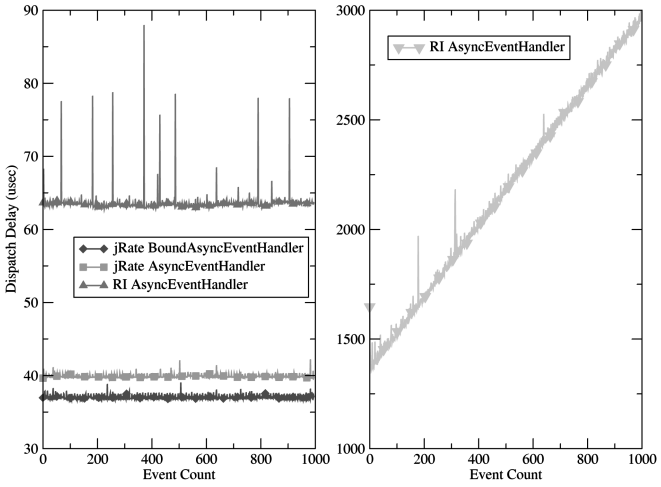


Fig. 12. Dispatch latency trend for successive event firing.

Fig. 12 and Table 4 show that the average dispatch latency of jRate's `AsyncEventHandler` is the same order of magnitude as its `BoundAsyncEventHandler`. The difference between the two average dispatch latency stems from jRate's `AsyncEventHandler` implementation, which uses an *executor* [15] thread from a pool of threads to perform the event firing, rather than having a thread permanently bound to the handler.

- **Dispersion Measures**—The results in Tables 5 and 4, Fig. 12, and Fig. 13 illustrate how jRate's `BoundAsyncEventHandler` dispatch latency incurs the least jitter. The dispatch latency value dispersion for the RTSJ RI `BoundAsyncEventHandler` is also quite good, though its jitter is higher than jRate's `AsyncEventHandler` and `BoundAsyncEventHandler`. The higher jitter in RI may stem from the fact that the RI stores the event handlers in a `java.util.Vector`. This data structure achieves thread-safety by synchronizing all method that `get()`, `add()`, or `remove()` elements from it, which acquires and releases a lock associated with the vector for each method.

  To avoid the locking overhead incurred by the RI, jRate uses a data structure that associates the event handler list with a given event and allows the contents of the data structure to be read without acquiring/releasing a lock. Only modifications to the data structure itself must be serialized. As a result, jRate's `AsyncEventHandler` dispatch latency is relatively predictable, even though the handler has no thread bound to it permanently. The jRate thread pool implementation uses LIFO queues for its executor, i.e., the last executor that has completed executing is the first one reused. This technique is often applied in thread pool implementations to leverage cache affinity benefits [16].
- **Worst-Case Measures**—Table 4 illustrates how the jRate's `BoundAsyncEventHandler` and `AsyncEventHandler` have worst-case execution time that is close to its average-case. The worst-case dispatch delay of the RI's `BoundAsyncEventHandler` is not as low as the one provided by jRate due to differences in how their event dispatching mechanisms are implemented.

TABLE 4
jRate Event Handler's Dispatch Latency Statistics

	AsycnEventHandler	BoundAsycnEventHandler
Avg.	39.884 μs	36.809 μs
Std. Dev.	0.2115 μs	0.231 μs
Max	42.211 μs	39.376 μs
99%	40.665 μs	37.504 μs

TABLE 5
RI Event Handler's Dispatch Latency Statistics

	AsycnEventHandler	BoundAsycnEventHandler
Avg.	2177.3 μs	63.600 μs
Std. Dev.	463.59 μs	1.4813 μs
Max	2997.8 μs	87.968 μs
99%	2953.7 μs	68.297 μs

4. *Asynchronous Event Handler Priority Inversion Test:* If the right data structure is not used to maintain the list of event handlers associated with an event, an unbounded priority inversion can occur during the dispatching of the event. This test therefore measures the degree of priority inversion that occurs when multiple handlers with different SchedulingParameters are registered for the same event. This test registers N handlers with an event in order of increasing importance. The time between the firing and the handling of the event is then measured for the highest priority event handler.

   By comparing the results for this test with the result of the test described above, RTJPerf can determine the degree of priority inversion present in the underlying RTSJ event dispatching implementation. Section 4.3.3 provides an analysis of the implementation of the current RI and presents how jRate overcomes some RI shortcomings.

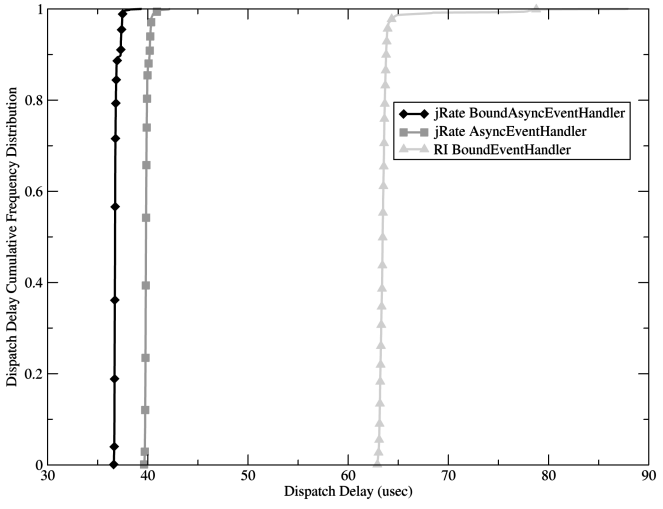


Fig. 13. Cumulative dispatch latency distribution.

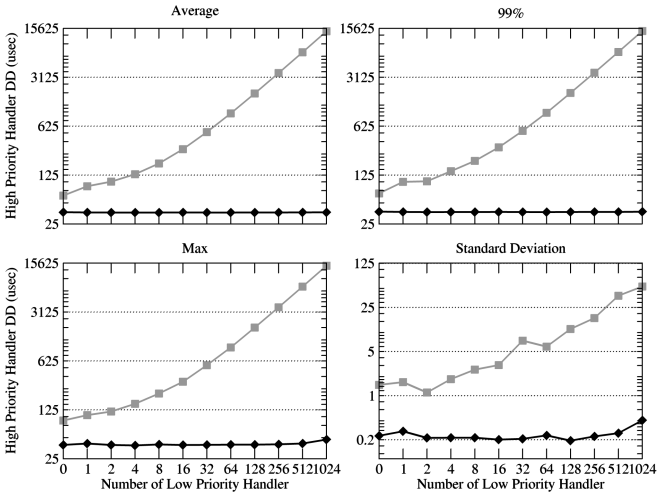


Fig. 14. H's dispatch latency statistics.

5. *Test Settings:* This test uses the same settings as the asynchronous event handler dispatch delay test. Only the `BoundAsyncEventHandler` performance is measured, however, because the RI's `AsyncEventHandlers` are essentially unusable since their dispatch latency grows linearly with the number of events handled (see Fig. 12), which masks any priority inversions. Moreover, jRate's `AsyncEventHandler` performance is similar to its `BoundAsyncEventHandler` performance, so the results obtained from testing one applies to the other. The current test uses the following two types of asynchronous event handlers:
   - The first is identical to the one used in the previous test, i.e., it gets a time stamp after the handler is called and measures the dispatch latency. This logic is associated with H.
   - The second does nothing and is used for the lower priority handlers.
6. *Test Results:* Fig. 14 shows how the average, standard deviation, maximum, and 99 percent bound of the dispatch delay changes for H as the number of low-priority handlers increase.
7. *Results Analysis:* Below, we analyze the results of the tests that measure average-case and worst-case dispatch latency, as well as its dispersion, for jRate and the RI.
   - **Average Measures**—Fig. 14 illustrates that the average dispatch latency experienced by H is essentially constant for jRate, regardless of the number of low-priority handlers. It grows rapidly; however, as the number of low-priority handlers increase for the RI. The RI's event dispatching priority inversion is problematic for real-time applications and stems from the fact that its queue of handlers is implemented with a `java.util.Vector`, which is not ordered by the *execution eligibility*. In contrast, the priority queues in jRate's event dispatching are ordered by the execution eligibility of the handlers.

   Execution eligibility is the ordering mechanism used throughout jRate. For example, it is used to achieve total ordering of schedulable entities whose QoS are expressed in different ways.
   - **Dispersion Measures**—Fig. 14 illustrates how H's dispatch latency dispersion grows as the number of low-priority handlers increases in the RI. The dispatch latency incurred by H in the RI therefore not only grows with the number of low-priority handlers, but its variability increases, i.e., its predictability decreases. In contrast, jRate's standard deviation increases very little as the low-priority handlers increase. As mentioned in the discussion of the average measurements above, the difference in performance stems from the proper choice of priority queue.
   - **Worst-Case Measures**—Fig. 14 illustrates how the worst-case dispatch delay is largely independent of the number of low-priority handlers for jRate. In contrast, worst-case dispatch delay for the RI increases as the number of low-priority handlers grows. The 99 percent bound is close to the average for jRate and relatively close for the RI.

5 Related Work

Although the RTSJ was adopted as a standard fairly recently [4], there are already a number of related research projects. The following projects are particularly interesting:

- The **FLEX** [17] provides a Java compiler written in Java, along with an advanced code analysis framework. FLEX generates native code for StrongARM or MIPS processors, and can also generate C code. It uses advanced analysis techniques to automatically detect the portions of a Java application that can take advantage of certain real-time Java features, such as memory areas or real-time threads.
- The Real-Time Java for Embedded Systems (RTJES) program [18] is working to mature and demonstrate real-time Java technology. A key objective of the RTJES program is to assess important real-time capabilities of real-time Java technology via a comprehensive benchmarking effort. This effort is examining the applicability of real-time Java within the context of real-time embedded system requirements derived from Boeing's Bold Stroke avionics mission computing architecture [19].

6 Concluding Remarks

The RTSJ is an important emerging middleware platform that defines a standard, high-level, and productive environment for developing real-time embedded applications. RTSJ encapsulates much of the complexity and platform-specific details in the middleware, and provides a powerful set of programming abstractions to application developers. This paper presented the architecture of jRate, which is an open-source ahead-of-time compiled RTSJ middleware that we have created at Washington University. This paper also used the open-source RTJPerf benchmarking suite to empirically

evaluate the performance of RTSJ middleware features in jRate and the TimeSys RTSJ RI that are crucial to the development of real-time embedded applications.

RTJPerf is one of the first open-source benchmarking suites designed to evaluate RTSJ-compliant Java implementations empirically. We believe it is important to have an open benchmarking suite to measure the quality of service of RTSJ implementations. RTJPerf not only helps guide application developers to select RTSJ features that are suited to their requirements, but also helps developers of RTSJ implementations evaluate and improve the performance of their products.

Although much work remains to ensure predictable and efficient performance under heavy workloads and high contention, our test results indicate that real-time Java is maturing to the point where it can be applied to certain types of real-time applications. In particular, the performance and predictability of jRate is approaching C++ for some tests. The TimeSys RTSJ RI also performed relatively well in some aspects, though it has problems with AsyncEventHandler dispatching delays and priority inversion.

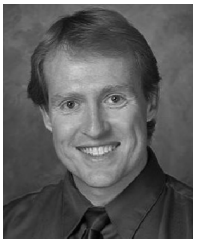
ACKNOWLEDGMENTS

The authors would like to thank Ron Cytron, Peter Dibble, David Holmes, Doug Lea, Doug Locke, Carlos O'Ryan, John Regehr, and Gautam Thaker for their constructive suggestions that helped to improve earlier drafts of this paper.

REFERENCES

[1] J. Sztipanovits and G. Karsai, "Model-Integrated Computing," *Computer*, vol. 30, no. 4, pp. 110-112, Apr. 1997.
[2] K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Language.* Boston: Addison-Wesley, 2000.
[3] R. Jones and R. Lins, *Garbage Collection Algorithms for Automatic Dynamic Memory Management.* New York: Wiley & Sons, 1996.
[4] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull, *The Real-Time Specification for Java.* Addison-Wesley, 2000.
[5] TimeSys, Real-Time Specification for Java Reference Implementation, www.timesys.com/rtj, 2001.
[6] A. Corsaro and D.C. Schmidt, "Evaluating Real-Time Java Features and Performance for Real-time Embedded Systems," *Proc. Eighth IEEE Real-Time Technology and Applications Symp.,* Sept. 2002.
[7] "The Design and Performance of the jRate Real-Time Java Implementation," *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE*, R. Meersman and Z. Tari, eds., pp. 900-921,Springer Verlag, 2002.
[8] GNU is Not Unix, GCJ: The GNU Complier for Java, http://gcc.gnu.org/java, 2002.
[9] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000.
[10] A. Corsaro and R.K. Cytron, "Efficient Memory-Reference Checks for Real-Time Java," *Proc. ACM SIGPLAN Conf. Language, Compiler, and Tool for Embedded Systems,* pp. 51-58, 2003.
[11] TimeSys, TimeSys Linux/RT 3.0, www.timesys.com, 2001.
[12] J. Gosling, B. Joy, and G. Steele, *The Java Programming Language Specification.* Addison-Wesley, 1996.
[13] IBM, Jikes 1.17, http://www.research.ibm.com/jikes/, 2001.
[14] D.C. Schmidt, M. Deshpande, and C. O'Ryan, "Operating System Performance in Support of Real-Time Middleware," *Proc. Seventh Workshop Object-Oriented Real-Time Dependable Systems,* Jan. 2002.
[15] D. Lea, *Concurrent Programming in Java: Design Principles and Patterns,* second ed. Addison-Wesley, 2000.
[16] J.D. Salehi, J.F. Kurose, and D. Towsley, "The Effectiveness of Affinity-Based Scheduling in Multiprocessor Networking," *Proc. IEEE INFOCOM,* Mar. 1996.
[17] M. Rinard et al., "FLEX Compiler Infrastructure," http://www.flex-compiler.lcs.mit.edu/Harpoon/, 2002.
[18] J. Lawson, "Real-Time Java for Embedded Systems (RTJES)," http://www.opengroup.org/rtforum/jan2002/slides/java/lawson.pdf, 2001.
[19] D.C. Sharp, "Reducing Avionics Software Cost through Component Based Product Line Development," *Proc. 10th Ann. Software Technology Conf.,* Apr. 1998.

Angelo Corsaro received the Laurea degree in computer engineering in 1999 from the University of Catania, and the MS degree in computer science from Washington University in 2001. Currently, he is a PhD candidate in the Computer Science Department of Washington University. His research focuses on real-time Java extensions and optimizations, programming languages, generative programming, real-time distributed systems, software patterns, optimization techniques, and empirical analysis of object-oriented frameworks and middleware platforms. He is a student member of the IEEE.



Douglas C. Schmidt is a professor in the Electrical Engineering and Computer Science Department at Vanderbilt University. His research focuses on patterns, optimization techniques, and empirical analysis of object-oriented frameworks that facilitate the development of distributed real-time and embedded (DRE) middleware running over high-speed networks and embedded system interconnects. Dr. Schmidt has served as a Deputy Office Director and a Program Manager at DARPA, where he led the national R&D effort on DRE middleware. Dr. Schmidt has also served as the cochair for the Software Design and Productivity (SDP) Coordinating Group of the US government's multiagency Information Technology Research and Development (IT R&D) Program, which formulated the multiagency research agenda in software design. He is a member of the IEEE and the IEEE Computer Society