The Design and Performance of the jRate Real-time Java Implementation

Angelo Corsaro and Douglas C. Schmidt

Electrical and Computer Engineering Department
University of California, Irvine, CA 92697*
{corsaro, schmidt}@ece.uci.edu

This paper has been submitted to the 4th International Symposium on Distributed Objects and Applications, Irvine, CA, October-November, 2002.

Abstract

Over 90 percent of all microprocessors are now used for real-time and embedded applications. Since the behavior of these applications is often constrained by the physical world, it is important to devise higher-level programming languages and middleware that robustly and productively enforce real-time constraints, as well as meeting conventional functional requirements. This paper provides two contributions to the study of programming languages and middleware for real-time and embedded applications. We first present how we are applying generative programming techniques to develop jRate, which is an open-source ahead-of-time-compiled implementation of the Real-time Specification for Java (RTSJ). The goal of jRate is to provide developers the ability to generate RTSJ implementations that are customized for their needs. We then show performance results of jRate that illustrate how well it performs compared to the TimeSys RTSJ Reference Implementation (RI).

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), tele-medicine (*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, companies are now also faced with the problem of developing and delivering new products in short time frames. It is therefore essential that the production of real-time embedded systems can take advantage of languages, tools, and methods that enable higher software productivity.

1.2 The State of the Art

Many real-time embedded systems are still developed in C, and increasingly also in C++. While writing in C/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 [20]. 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 a programming language that shields them from many accidental complexities, such as type errors, memory management, and steep learning curves. The Java [1] programming language has become an attractive choice for the following reasons:

- It has a large and rapidly growing programmer base and is taught in many universities.

- It is simpler than C++, yet programmers experienced in

C++ can learn it easily.

- It has a virtual machine architecture—the Java Virtual Machine (JVM)—that allows Java applications to run on any platform that supports a JVM.

- It has a powerful, portable standard library that can reduce programming time and costs.

- It offloads many tedious and error-prone programming details, particularly memory management, from developers into the language runtime system.

- It has desirable language features, such as strong typing, dynamic class loading, and reflection/introspection.

- It defines portable support for concurrency and synchronization.

- Its bytecode representation is more compact than native code, which can reduce memory usage for embedded systems.

Conventional Java implementations are unsuitable for developing real-time embedded systems, however, due to the following problems:

- The scheduling of Java threads is purposely underspecified to make it easy to develop JVMs for new platforms.

- The Java Garbage Collector (GC) has higher execution eligibility that any other Java thread, which means that a thread could experience unbounded preemption latency while waiting for the GC to run.

- 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.

- Due to its interpreted origins, the performance of JVMs has historically lagged that of equivalent C/C++ programs by an order of magnitude or more.

To address these problems, the Real-time Java Experts Group has defined the Real-Time Specification for Java (RTSJ) [2], which provides the following capabilities:

- 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 was no implementation 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 Reference Implementation (RI) from TimeSys [23].

1.3 The Road Ahead

While the RTSJ represents an ambitious step toward improving the state of the art in embedded and real-time system development, there are a number of open issues. In particular, the RTSJ was designed with generality in mind. While this is a laudible goal, generality is often at odds with the resource constraints of embedded systems. Moreover, providing developers with an overly general API can actually increase the learning curve and introduce accidental complexity in the API itself.

For example, the scheduling API in RTSJ was designed to match any scheduling algorithm, including RMS, EDF, LLF, RED, MUF, etc. While this generality covers a broad range of alternatives, it may be overly complicated for an application that simply needs a priority preemptive scheduler. But can we do any better then this? Can we provide the needed flexibility and extensibility, without putting undue burden on developers?

We believe that the answer is affirmative, based on our experience to date using Generative Programming (GP) [7] techniques, such as Aspect-Oriented Programming (AOP) [12], Meta-Programming (MP) [11], Component-Oriented Programming (COP) [10], and Model-Integrated Computing (MIC) [20]. Generative programming makes it possible to develop middleware systems that are amenable to customization of behavior and protocols (*e.g.*, APIs), via automatic code generation and composition.

Using a GP approach, the development of middleware, such as RTSJ or Real-time CORBA [19], need not lead to a single implementation. Instead, it can provide a set of components and configuration knowledge that can be used to generated a specific implementation based on user-defined specifications. If we consider the RTSJ scheduling API example, for instance, application developers that need a simple priority preemptive scheduler could use generative programming to specify this as a requirement. The outcome of the generation process would then be a Real-time Java platform that exposed only the API needed for a priority-based scheduler and whose implementation was also optimized for priority-based schedulers.

1.4 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 presents empirical results obtained by benchmarking jRate and the TimeSys RTSJ Reference Implementation (RI) using our RTJPerf [4] benchmarking suite; Section 5 compares our work on jRate with related research; and Section 6 summarizes the results we obtained and outlines how they can be used to improve the support of next-generation implementations of RTSJ for real-time embedded software.

2 Overview of 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 systems. The guiding principles followed by the expert group who created the RTSJ specification included [2]:

- Backward compatibility with the Java 2 platform
- No syntactic extension to the Java language, *i.e.* no new keywords
- Write once carefully, run anywhere conditionally
- Enable predictable execution and
- Balance between current practice and advanced features.

Below, we present an overview of the extensions provided by the RTSJ.

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 lifetime the objects created in the given memory area and/or by their allocation time. *Scoped memory areas* provide guarantees on allocation time. Each real-time thread is associated with a *scope stack* that defines its allocation context and the *history* of the memory areas it has entered. Figure 1 shows how the scope stack for threads T_1 and T_2 evolve while moving from one memory area to another. As shown in Figure 2, the RTSJ specification pro-
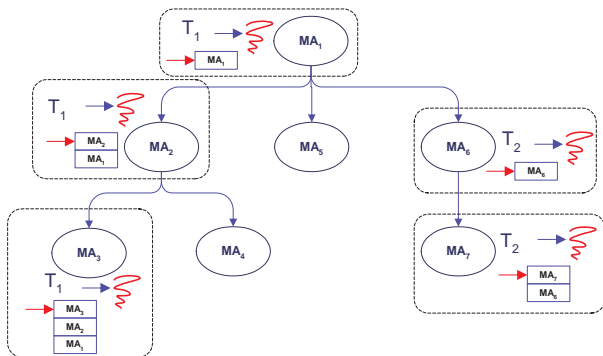

Figure 1: **Thread Scope Stack in the RTSJ Memory Model**

vides scoped memories with linear and variable allocation times (LTMemory, LTPhysicalMemory and VTMemory, VTPhysicalMemory, respectively). For linear allocation time scoped memory, 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] The RTSJ also introduces the concept of

---

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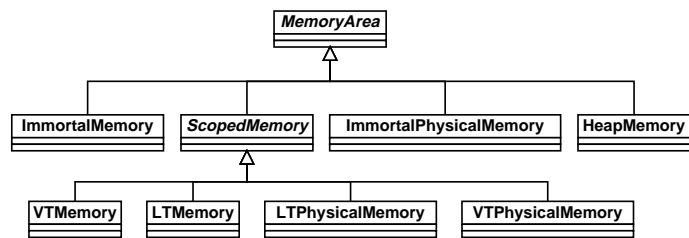

Figure 2: **Hierarchy of Classes in the RTSJ Memory Model**

Immortal Memory. Objects allocated within this memory area have the same lifetime of the JVM, *i.e.* are never collected. Another addition to the Java memory model provided by the RTSJ allows direct access to raw memory, as well as to allocate Java objects at specific memory locations.

2.2 Threads

The RTSJ extends the existing Java threading model with two new types of real-time threads: RealtimeThread and NoHeapRealtimeThread. The relation of these new classes with respect to the regular Java thread class is depicted in Figure 3.
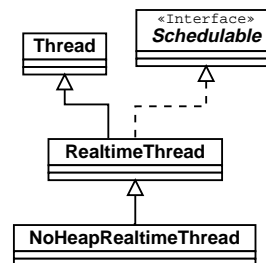

Figure 3: **RTSJ Real-time Thread class Hierarchy**

The NoHeapRealtimeThread can have execution eligibility higher than the garbage collector.[2] Therefore, a NoHeapRealtimeThread can neither allocate nor reference any heap objects. The scheduler controls the *execution eligibility*[3] of the instances of this class by using the SchedulingParameters associated with it.

---

[2]The RTSJ v1.0 specification states that the NoHeapRealtimeThread always has execution eligibility higher than the GC, but this has been changed in the v1.01.

[3]Execution eligibility is defined as the position of a schedulable entity in a total ordering established by a scheduler over the available entities [6]. The total order depends on the scheduling policy. The only scheduler required by the RTSJ is a priority scheduler, which uses the PriorityParameters to determine the execution eligibility of a Schedulable entity, such as threads or event handlers.

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 RMS, EDF, LLF, RED, MUF, 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. As shown in Figure 4, there are two types of asyn-
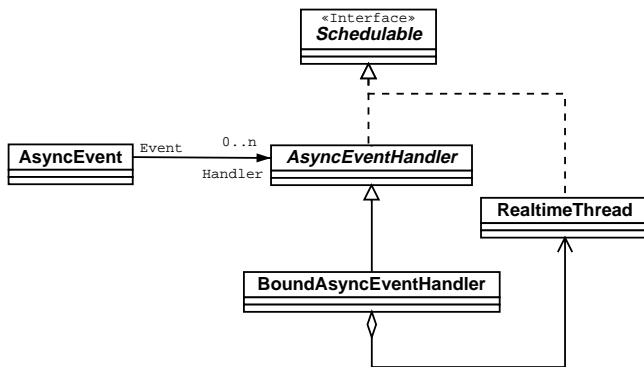

Figure 4: **RTSJ Asynchronous Event Class Hierarchy**

chronous event handlers defined in RTSJ:

- The `AsyncEventHandler` class, which does not have a thread permanently bound to it – nor is it 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. An `BoundAsyncEventHandler`'s 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. As shown in Fig-
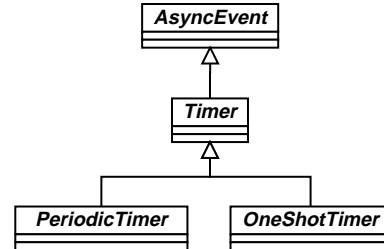

Figure 5: **RTSJ Timer Class Hierarchy**

ure 5, 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 the University of California, Irvine (UCI). jRate extends the open-source the GNU Compiler for Java (GCJ) runtime system [9] to provide an ahead-of-time compiled platform for the development of RTSJ-compliant applications. The jRate architecture shown in Fig-
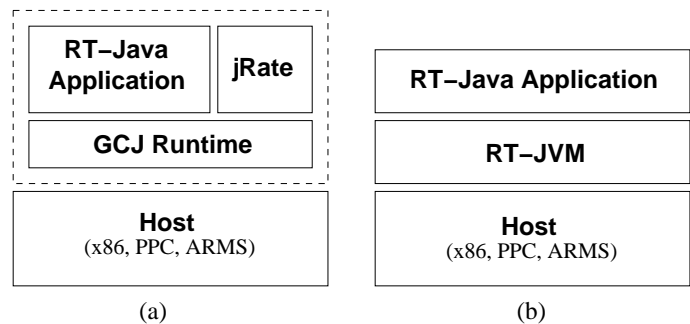

Figure 6: **The jRate Architecture**

ure 6(a) differs from the JVM model shown in Figure 6(b) since there is no JVM interpreting the Java bytecode. Instead, jRate ahead-of-time 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.

One downside of ahead-of-time compiled RTSJ implementations like jRate is that they can hinder portability since applications must be recompiled each time they are ported to a new architecture. In practice, however, embedded and real-time software developers should find this a small price to pay for the substantial performance benefits, compared to Java implementations based on interpreters or just-in-time (JIT) compilers.

3.1 Current jRate Capabilities

The RTSJ features currently supported by jRate are described next.

3.1.1 Memory Areas

jRate supports scoped memory and immortal memory. It provides a strategy to decide which type of memory, such as linear time memory or variable time memory, should be used as immortal memory. The RTSJ does not specify the immortal memory implementation, but since we believe it is important to specify the type of memory used, jRate can configure it at application launch time. The scoped memory implementation exposes an additional non-standard extension that allows the use of non-thread safe allocators. This extension allows threads to avoid unnecessary locks if a memory area will always be accessed by one thread.

jRate also provides a new type of scoped memory called CTMemory, which 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[4] drops to zero. This feature provides constant time allocation for objects created within the CTMemory.

The internal organization of the CTMemory is depicted in Figure 7. The *type* field distinguishes different types of ob-
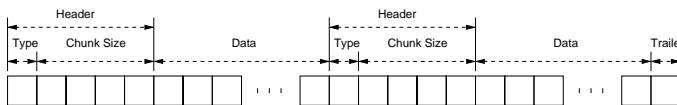

Figure 7: **The jRate CTMemory Structure**

jects. In fact, there are different types of objects that must be treated slightly differently, *e.g.*, some must be finalized, whereas others need not be finalized.

3.1.2 Real-Time Threads and Scheduling

jRate currently supports real-time threads of the type RealtimeThread (*i.e.* it does not yet support NoHeapRealtimeThread), using a basic priority preemptive scheduler. This implementation simply relies upon the underlying real-time operating system priority preemptive scheduler.

3.1.3 Asynchrony

jRate provides a robust and efficient asynchronous event handling implementation, as shown by the empirical results in Section 4.2. This implementation avoids any source of priority inversion and provides lock free dispatch on most platforms.[5] jRate uses the priority queues ordered by the execution eligibility of the handlers for the event dispatching. 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. This approach is an application of the formalisms presented in [6].

3.1.4 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.

3.2 Next Steps: A Chameleonic Real-Time Java Implementation

jRate is intended to be a "chameleonic" Real-time Java implementation. We use the analogy since jRate is designed to its target environment, just as a chameleon adapts to its surrounding environment. In jRate, the adaptation process is obtained via generative programming techniques. Our ultimate goal is to provide a set of core reusable components, along with the appropriate configuration tools, so application developers can automatically generate a customized Real-time Java implementation that precisely meets their needs. Our work to date has focused on manually generating high performance and small footprint implementations of the RTSJ specification.

Figure 8 shows a typical generative programming approach. In this approach, the *feature profile* provided by application developers is used to select the set of features that must be present in the generated Real-time Java implementation. Generative tools are used to compose the different parts, check dependencies, and optimize the generated system.

---

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

[5]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.
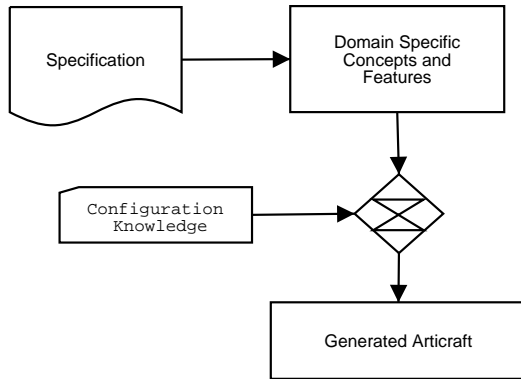
Figure 8: **Generative Programming Approach**

For instance, in the context of jRate, a given *feature profile* might designate the following:

- A priority-based scheduler with no support for feasibility analysis
- Raw memory access is not needed, and
- A particular style of scoped memory must be used.

This information is then used by generators to provide an instance of jRate that is optimized for this particular use case. The generated API could be simplified since the user only wants to use priority-based scheduler and does not need any feasibility analysis support.

The tools we are applying to make jRate a *generative* real-time Java implementations include AspectJ [22] and AspectC++ [21], along with other techniques that are commonly used in generative programming, such as:

- **Static crosscutting** to compose APIs. Static crosscutting is an AOP technique that allows "meta-programmers" to modify the static structure of a class, *e.g.*, by adding methods or changing an inheritance hierarchy.
- **Dynamic crosscutting** is used to customize and compose run-time behaviors. Dynamic crosscutting is another AOP technique that allows the execution of aspect code at specific points in the application, known as *join-points*, such as method invocations, data member assignments, etc.

4 jRate Performance

This section presents jRate's performance results for the primary RTSJ features. All experiments were conducted using RTJPerf, which is an open-source benchmarking suite for RTSJ available at `http://tao.doc.wustl.edu/~corsaro/periscope.html`. see [4, 5] for in-depth coverage of the RTJPerf benchmark suite and a comparison of

jRate's performance with a range of Java implementations, including CVM, the TimeSys RTSJ Reference Implementation (RI), and Sun's JDK 1.4.

The test results reported in this section were obtained on an Intel Pentium III 733 MHz with 256 MB RAM, running Linux RedHat 7.2 with the TimeSys Linux/RT 3.0 GPL[6] kernel [24]. jRate was compared against the TimeSys RTSJ RI [23], to provide a baseline to compare jRate against. The RI is based on a Java 2 Micro Edition (J2ME) JVM and supports only an interpreted execution mode *i.e.*, there is no just-in-time (JIT) compilation. The *efficiency* of the RI was intentionally not optimized since its main goal was *predictable* real-time behavior and RTSJ-compliance. The RI runs on all Linux platforms, but the priority inversion control mechanisms are available to the RI only when running under TimeSys Linux/RT [24], *i.e.*, the commercial version.

4.1 jRate Memory Subsystem Performance

Scoped Memory Allocation Time Test. RTJPerf provides a test that measures the allocation time for different types of scoped memory. The results obtained for the jRate's and RI implementation of scoped memory are presented and analyzed below.

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. Figure 9 shows how jRate's `CTMemory` implementation relates to the memory areas defined by the RTSJ, which are depicted in Figure 2.
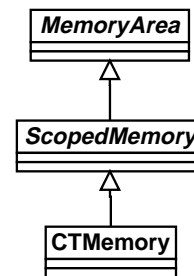

Figure 9: `CTMemory` **Class Hierarchy**

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 indices since they indicate the following information:

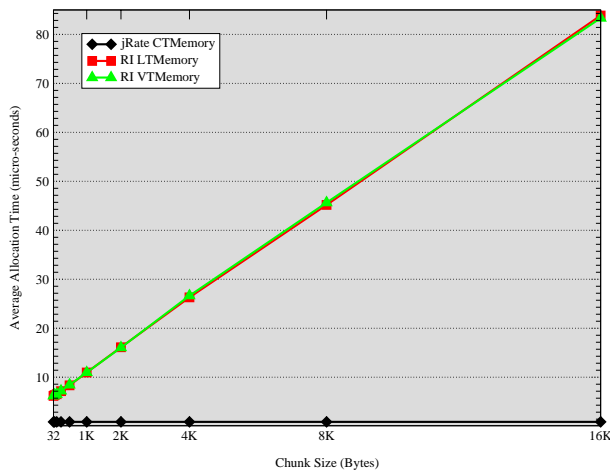[6]This OS is the freely available version of TimeSys Linux/RT and is available under the GNU Public License (GPL).
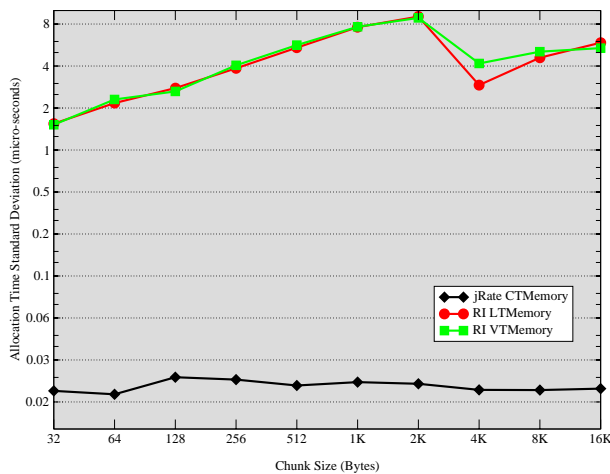
Figure 10: **Average Allocation Time.**

Figure 11: **Allocation Time Standard Deviation.**

- How predictable is the 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% upper bound.[7]

Figure 10 shows the resulting average allocation time for the different test runs and Figure 11 shows the standard deviation of the allocation time measured in the various test settings. Figure 12 shows the performance ratio between jRate's CTMemory and the RI LTMemory. This ratio indicates how many times smaller the CTMemory average allocation time is compared to the average allocation time for the RI LTMemory.

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:

[7]By "99% upper bound" we mean that value that represents an upper bound for the measured values in the 99th percentile of the cases.
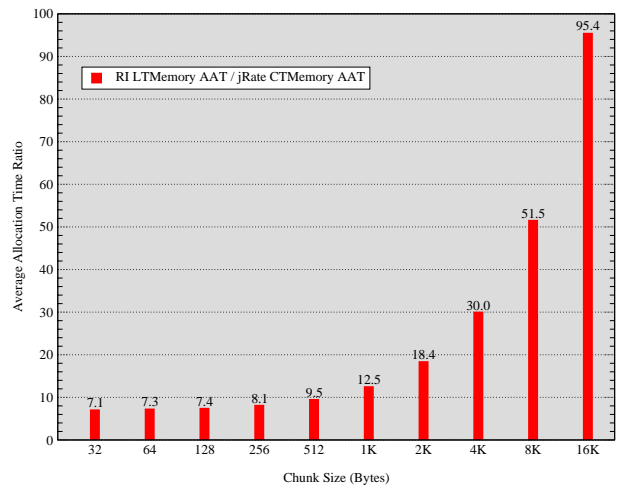

Figure 12: **Speedup of the** CTMemory **Average Allocation Time Over the LTMemory Average Allocation Time.**

- **Average Measures**—As shown in Figure 10, both LTMemory and VTMemory provide linear time allocation with respect to the allocated memory size. Matching results were found for the other measured statistical parameter, based on this, we infer that the RI implementation of LTMemory and VTMemory are similar, so we mostly focus 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 real-time Java code, even without knowing the amount of memory that will be needed. Figure 12 shows that for small memory chunks the jRate memory allocator is nearly ten times faster than RI's LTMemory. For the biggest chunk we tested, jRate's CTMemory is ∼95 times faster RI's LTMemory.

- **Dispersion Measures**—The standard deviation of the different allocation time cases is shown in Figure 11. 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 to store freelist management information. In contrast, the CTMemory implementation has the smallest variance and the flattest trend.

The plots in Figure 13 show the cumulative relative frequency distribution of the allocation time for some of the different cases discussed above. These graphs 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
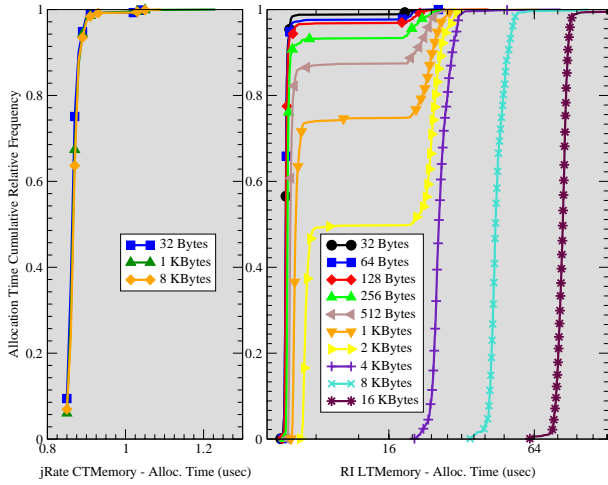
7

Figure 13: **Allocation Time Cumulative Relative Frequency Distribution.**

Figure 14: **CTMemory Worst, Best, Average and 99% Allocation Time.**
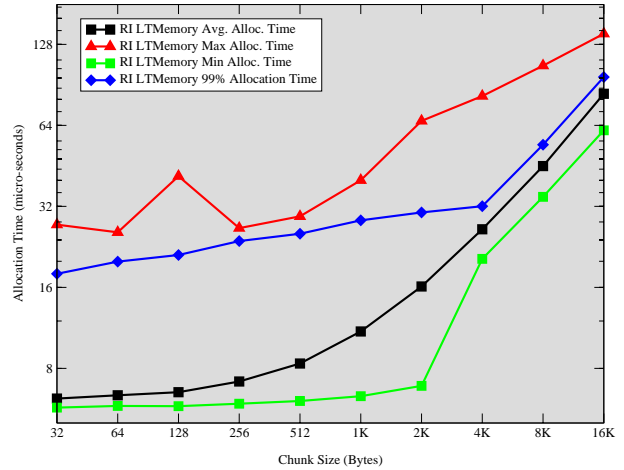
Figure 15: **LTMemory Worst, Best, Average and 99% Allocation Time.**

$AllocationTime \leq t$. This graph, along with Figure 11 that shows the standard deviation, provides insights on how the measured allocation time is dispersed and distributed.

- **Worst-case Measures**—Figure 14 and Figure 15 show the bounds on the allocation time for jRate's CTMemory and the RI LTMemory. Each of these graphs depicts the worst, best, and average allocation times, along with the 99% upper bound of the allocation time. Figure 14 illustrates how the worst-case execution time for jRate's CTMemory is at most ∼1.4 times larger than its average execution time.

Figure 15 shows how the maximum, average, and the 99% case, for the RI LTMemory, converge as the size of the allocated chunk increases. The minimum ratio between the worst-case allocation time and the average-case is ∼1.6 for a chunk size of 16K. Figure 14, Figure 15 and Figure 13 also characterize the distribution of the allocation time. Figure 13 shows how for some allocation sizes, the allocation time for the RI LTMemory is centered around two points.

4.2 jRate's Asynchronous Event Handler Performances.

Asynchronous Event Handler Dispatch Delay Test. RTJPerf provides a test that measures the dispatch latency of the two types of RTSJ asynchronous event handlers, which are the BoundAsyncEventHandler and the AsyncEventHandler. The results we obtained are presented and analyzed below.

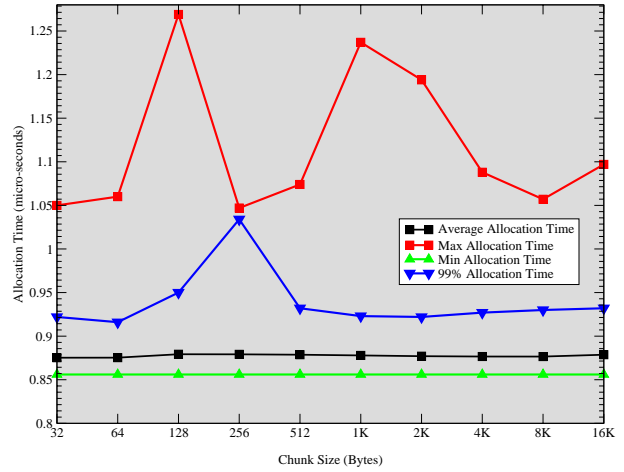**Test Settings.** To measure the dispatch latency provided by different types of asynchronous event handlers defined by the RTSJ, we ran the asynchrony tests provided by RTJPerf, and described in [4, 5], with a fire count of 2,000 for both RI and jRate. To ensure that each event firing causes a complete 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 GC while performing the test, the real-time thread that fires and handles the event uses scoped memory as its current memory area.

Test Results. Figure 16 shows the trend of the dispatch latency for successive event firings.[8] The data obtained by

---

[8]Since The RI's AsyncEventHandler trend is completely off the scale, it is omitted in this figure and depicted separately in Figure 17.
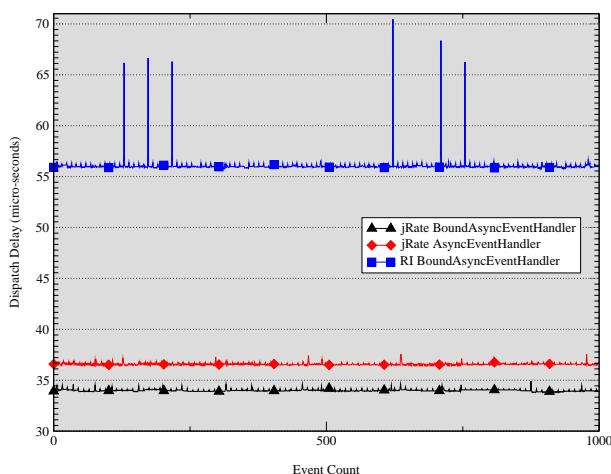
Figure 16: **Dispatch Latency Trend for Successive Event Firing.**

	AsycnEventHandler	**BoundAsycnEventHandler**
Avg.	36.574 μs	34.004 μs
Std. Dev.	0.113 μs	0.148 μs
Max	39.400 μs	35.555 μs
99%	36.945 μs	34.472 μs

Table 1: **jRate Event Handler's Dispatch Latency statistics for the Different Settings**

running the dispatch delay tests were processed to obtain average worst-case and dispersion measure of the dispatch latency. Table 1 and Table 2 shows the results found for jRate and the RI respectively.
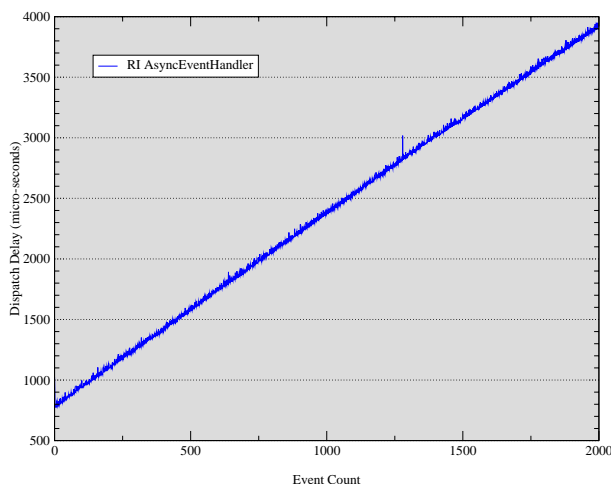

Figure 17: `AsyncEventHandler` **Dispatch Latency Trend.**

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:

	AsycnEventHandler	**BoundAsycnEventHandler**
Avg.	2373.0 μs	56.100 μs
Std. Dev.	909.92 μs	0.848 μs
Max	3950.8 μs	70.462 μs
99%	3892.5 μs	56.692 μs

Table 2: **RI Event Handler's Dispatch Latency Statistics for the Different Settings**

- **Average Measures**—Table 2 illustrates the large average dispatch latency incurred by the RTSJ RI `AsyncEventHandler`. The results in Figure 17 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.

  Figure 16 and Table 1 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* [14] 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 Table 2, Table 1, Figure 16, and Figure 18 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 this locking overhead, 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
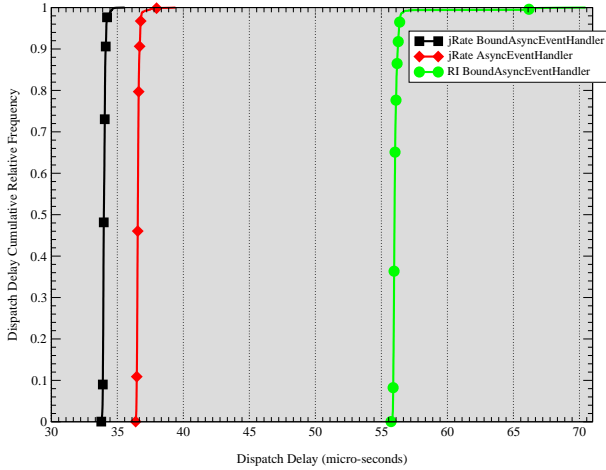
Figure 18: **Cumulative Dispatch Latency Distribution.**

read without acquiring/releasing a lock. Only modifications to the data structure 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 [17].

- **Worst-case Measures**—Table 1 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 provided by the RI's BoundAsyncEventHandler is not as good as the one provided by jRate, due to differences in how their event dispatching mechanisms are implemented. The 99% bound differs only on the first decimal digit for both jRate and the RI (clearly we do not consider the RI's AsyncEventHandler since no bound can be put on its behavior).

Asynchronous Event Handler Priority Inversion Test. This test measures how the dispatch latency of an asynchronous event handler H is influenced by the presence of N others event handlers, characterized by a lower execution eligibility than H. In the ideal case, H's dispatch latency should be independent of N, and any delay introduced by the presence of other handlers represents some degree of priority inversion. The results we obtained are presented and analyzed below.

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 event handled (see Figure 17), 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.

Test Results. Table 3 and Table 4 report how the average, standard deviation, maximum and 99% bound of the dispatch delay changes for H as the number of low-priority handlers increase. Figure 19 and Figure 20 provide a graphical representation for the average and dispersion measures.

	Avg.	Std. Dev.	Max	99%
0 LP	33.375 μs	0.124 μs	34.877 μs	34.116 μs
10 LP	33.154 μs	0.134 μs	34.903 μs	33.797 μs
50 LP	33.205 μs	0.161 μs	36.063 μs	33.825 μs
100 LP	33.264 μs	0.147 μs	35.959 μs	33.851 μs
500 LP	33.632 μs	0.180 μs	37.149 μs	34.283 μs
1000 LP	33.739 μs	0.199 μs	37.565 μs	34.458 μs

Table 3: jRate**'s Dispatch Delay Statistics.**

	Avg.	Std. Dev.	Max	99%
0 LP	56.106 μs	0.887 μs	70.462 μs	56.706 μs
10 LP	112.33 μs	1.346 μs	133.90 μs	122.18 μs
50 LP	332.41 μs	2.396 μs	353.17 μs	344.86 μs
100 LP	609.92 μs	3.410 μs	631.51 μs	624.96 μs
500 LP	2826.4 μs	12.005 μs	2884.0 μs	2862.1 μs
1000 LP	5587.0 μs	23.768 μs	5672.7 μs	5650.3 μs

Table 4: **RI's Dispatch Delay Statistics.**

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**—Figure 19 and Tables 3 and 4 illustrate 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 systems 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.
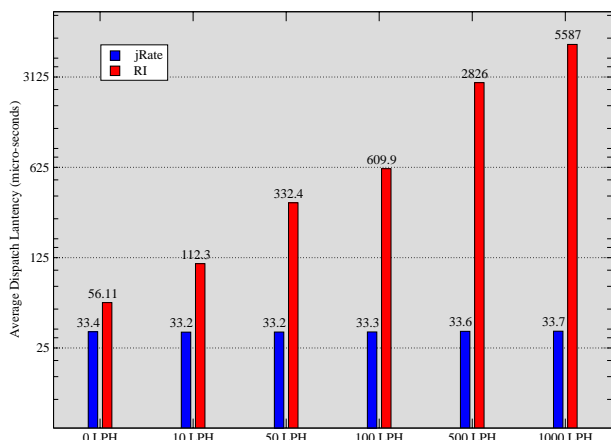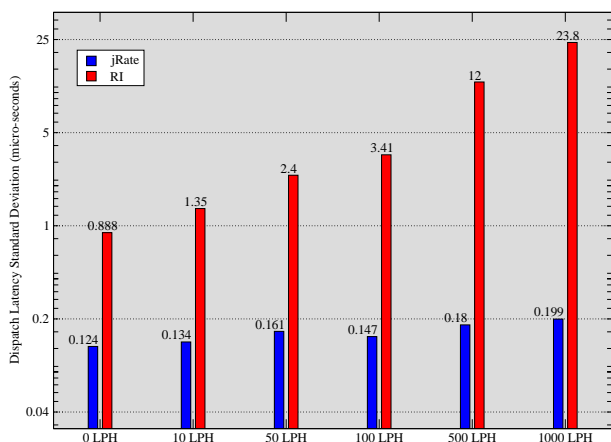
Figure 19: H's **Average Dispatch Latency.**

Figure 20: H **Dispatch Latency's Standard Deviation.**

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. This approach is an application of the formalisms presented in [6].

- **Dispersion Measures**—Figure 20 and Tables 3 and 4 illustrate 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**—Tables 3 and 4 illustrate how the worst-case dispatch delay is largely independent of the number of low-priority handlers for jRate. In con-

trast, worst-case dispatch delay for the RI increases as the number of low-priority handlers grows. The 99% bound is close to the average for jRate and relatively close for the RI.

5 Related Work

Although the RTSJ was adopted fairly recently [2], there are already a number of research projects related to our work on jRate and RTJPerf. The following projects are particularly interesting:

- The **FLEX** [15] 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 OVM [16] project is developing an open-source JVM framework for research on the RTSJ and programming languages. The OVM virtual machine is written entirely in Java and its architecture emphasizes customizability and pluggable components. Its implementation strives to maintain a balance between performance and flexibility, allowing users to customize the implementation of operations such as message dispatch, synchronization, field access, and speed. OVM allows dynamic updates of the implementation of instructions on a running VM.

- Work on real-time storage allocation and collection [8] is being conducted at Washington University, St. Louis. The main goal of this effort is to develop new algorithms and architectures for memory allocation and garbage collection that provide worst-case execution bounds suitable for real-time embedded systems.

There are several ways in which we plan to leverage our work on jRate and the work being done in the FLEX, OVM, and real-time allocator projects outlined above. For instance, the jRate RTSJ library implementation could become the library used by the OVM. This is possible because jRate has been designed to port easily from one Java platform to another. jRate could be used as the RTSJ library on which FLEX relies. Likewise, the work on real-time allocators and garbage collectors could be to implement jRate's scoped memory with different characteristics than its current CTMemory design.

6 Concluding Remarks and Future Directions

This paper presented an overview of jRate, which is an ahead-of-time compiled implementation of RTSJ. We analyzed the results of systematic benchmarks of jRate and the TimeSys RTSJ RI based on the RTJPerf benchmarking suite [4]. RTJPerf is one of the first open-source benchmarking suites designed to evaluate RTSJ-compliant Java implementations empirically. The RTJPerf results shown in Section 4 underscore that jRate is both efficient and predictable since its use of ahead-of-time compilation (1) improves its performance and (2) limits the sources of overhead and jitter introduced by interpreted or just-in-time (JIT) compiled execution.

Although jRate implements many core RTSJ features, the following omissions will be addressed in our future work:

- Add support for the remaining RTSJ features, such as timers, POSIX signal handling, periodic and no-heap real-time threads, and physical memory access. Some feature that we don't plan to implement in the first release of jRate is the memory reference checking, and the Asynchronous transfer control. Since jRate will be the primary Real-Time Java platform used by ZEN [13], jRate's implementation is being driven by the features that are most important for real-time ORBs.

- Provide a user-level scheduling framework that leverages the simple priority-based scheduling provided by the underlying real-time operating systems to provide advanced scheduling services.

- Focus on applying generative programming techniques to jRate. This will involve completely partitioning the Java and C++ parts of jRate into sets of aspects that can be woven together at compile-time to configure custom real-time Java implementations that are tailored for specific application needs.

- Provide a meta-object protocol as one of the aspects to support both computational and structural reflection. Reflection is useful for real-time applications that must manage resources dynamically. Moreover, it enables developers to customize the behavior of jRate's implementation at run-time.

Our long-term goal is to create not just a single RTSJ implementation, but a set of components that allow users to generate custom Real-time Java implementation tailored for particular requirements and environments. Developer therefore will not have to pay the cost and accidental complexity for features that they do not use.

The first public version of jRate will be released by the end of June, 2002. Information on its current status and availability can be found at `http://tao.doc.wustl.edu/`
`~corsaro/jRate`. Since jRate is an open-source project, we encourage researchers and developers to provide us feedback and help improve its quality and capabilities. jRate will use the same open-source model we use for ACE [18] and TAO [3], which has proved to be successful to produce high-quality open-source middleware.

References

[1] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, Boston, 2000.

[2] Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, and Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[3] Center for Distributed Object Computing. The ACE ORB (TAO). www.cs.wustl.edu/~schmidt/TAO.html, Washington University.

[4] Angelo Corsaro and Douglas C. Schmidt. Evaluating Real-Time Java Features and Performance for Real-time Embedded Systems. In *Proceedings of the 8th IEEE Real-Time Technology and Applications Symposium*, San Jose, September 2002. IEEE.

[5] Angelo Corsaro and Douglas C. Schmidt. Evaluating Real-Time Java Features and Performance for Real-time Embedded Systems. Technical Report 2002-001, University of Califoria, Irvine, 2002.

[6] Angelo Corsaro, Douglas C. Schmidt, Ron K. Cytron, and Chris Gill. Formalizing Meta-Programming Techniques to Reconcile Heterogeneous Scheduling Disciplines in Open Distributed Real-Time Systems. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications.*, pages 289–299, Rome, Italy, September 2001. OMG.

[7] Krzysztof Czaenwcki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, Massachusetts, 2000.

[8] Steven M. Donahue, Matthew P. Hampton, Morgan Deters, Jonathan M. Nye, Ron K. Cytron, and Krishna M. Kavi. Storage allocation for real-time, embedded systems. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Embedded Software: Proceedings of the First International Workshop*, pages 131–147. Springer Verlag, 2001.

[9] GNU is Not Unix. GCJ: The GNU Complier for Java. `http://gcc.gnu.org/java`, 2002.

[10] George T. Heineman and Bill T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, Reading, Massachusetts, 2001.

[11] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of The Metaobject Protocol*. The MIT Press, Cambridge, Massachusetts, 1991.

[12] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.

[13] Raymond Klefstad, Douglas C. Schmidt, and Carlos O'Ryan. The Design of a Real-time CORBA ORB using Real-time Java. In *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing*. IEEE, April 2002.

[14] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns, Second Edition*. Addison-Wesley, Boston, 2000.

[15] M.Rinard et.al. FLEX Compiler Infrastructure. `http://www.flex-compiler.lcs.mit.edu/Harpoon/`, 2002.

[16] OVM/Consortium. OVM An Open RTSJ Compliant JVM. `http://www.ovmj.org/`, 2002.

[17] James D. Salehi, James F. Kurose, and Don Towsley. The Effectiveness of Affinity-Based Scheduling in Multiprocessor Networking. In *IEEE INFOCOM*, San Francisco, USA, March 1996. IEEE Computer Society Press.

[18] Douglas C. Schmidt. The ADAPTIVE Communication Environment (ACE). www.cs.wustl.edu/∼schmidt/ACE.html, 1997.

[19] Douglas C. Schmidt and Fred Kuhns. An Overview of the Real-time CORBA Specification. *IEEE Computer Magazine, Special Issue on Object-oriented Real-time Computing*, 33(6), June 2000.

[20] Janos Sztipanovits and Gabor Karsai. Model-Integrated Computing. *IEEE Computer*, 30(4):110–112, April 1997.

[21] The AspectC++ Organization. Aspect-Oriented Programming for C++. www.aspectc.org, 2001.

[22] The AspectJ Organization. Aspect-Oriented Programming for Java. www.aspectj.org, 2001.

[23] TimeSys. Real-Time Specification for Java Reference Implementation. www.timesys.com/rtj, 2001.

[24] TimeSys. TimeSys Linux/RT 3.0. www.timesys.com, 2001.