Porting jRate (RT-Java) to a POSIX Real-Time Linux Kernel

Walter Mata, Apolinar González, Gerardo Fuentes, Ricardo Fuentes
University of Colima
Av Universidad No 333, Colima, Mexico
{wmata, apogon, fuentesg, fuentesr}@ucol.com.mx
Alfons Crespo

Polytechnic University of Valence
Camino de Vera S/N, Valence, Spain
acrespo@upv.es
Donald Carr

University of Guadalajara
Apdo. Postal 307, C.P. 45101 Zapopan, Jalisco, Mexico
doncarr@gmail.com

Abstract

This paper illustrates how the Real-Time Specification for Java (RTSJ) can be implemented over a real-time operating system (RTOS). We describe the implementation of a subset of the RTSJ over PaRTiKle, which is a new embedded RTOS designed to be compatible with the POSIX.51 standard.

We evaluate the performance of the implementation according to specific features of the RTSJ like eficiency and predictability, wich are important considerations for the real-time applications design. Specifically the metrics analysed are: real-time thread start latency, context switch, memory allocation latency, thread creation latency and periodic thread test. This evaluation will help to the development and improvement of future RTSJ implementations.

1 Introduction

The strong increasing presence of embedded systems in products and services creates huge opportunities for the future in different areas such as industrial control systems, avionics, health care, environment, security, mechanics, etc. Thus, there is a growing scientific interest on conceptual and practical tools for their development. Embedded systems are becoming more and more popular in a wide range of applications such as industrial control systems, avionics, health care, environment, security, mechanics. The development of the hardware and software for these systems require appropriate design, analysis and development tools. The underlying architecture also plays and important role. Special design related challenges come from the specialisation and customisation of target platforms in their use for embedded systems. The challenge is to maintain some degree of exibility to increase the reuse of software components.

In order to develop embedded systems, languages have played an important role. Basic requirements as object orientation, threads and real-time capabilities have been considered fundamental to develop these systems. Languages as C++, Ada or RT-Java achieve in higher or minor degree these requirements. On the other hand, efficiency is another important aspect to be considered.

In embedded systems dealing with soft real-time requirements, aspects as flexibility, adaptability and runtime support have a higher impact. In this en-

vironment, the benets of making Java real-time are clear. However, despite the advantages of a language as Java, its use for real-time applications presents important limitations. To avoid these limitations the Real-Time for Java Experts Group produced the Real-Time Specification for Java (RTSJ) [3]. It defines a set of extensions to the Java virtual machine and the class libraries that facilitate real-time programming.

Currently, there are several implementations conformant with this specification. Some of them are commercial (JamaicaVM [2], Java RTS [14]) and other are from the academia (jRate [10], FLEX [15]). Also, TimeSys [17] has freely released its version. In [8] a comparison of two of these implementation (Java RTS and JamaicaVM) were evaluated.

In this paper we present a porting of jRate over a POSIX Real-Time Linux Kernel adding some components proposed in Ravenscar Profile for hard real-time software development and evaluation of the porting realized. The implementation one became on top of a real-time operating system as PaRTiKle [16] which is going to substitute the RTLinux-GPL distribution [7]. This implementation offers the possibility to the RT-Linux community to use RTJava for real-time applications.

This paper is organised as follows. Section 2 presents the global architecture and details the layers involved. Section 3 provides a small description of the low level (real-time operating system) on top of which the RTJava has been ported. Section 4 presents the RTSJ Implementation. Section 5 offeres the evaluation results. Finally some conclusions are drawn.

2 Global Architecture

The GNU Compiler for the Java platform (GCJ) compiles Java code to native machine code using the GNU Compiler Collection (GCC) framework. GCJ provides the GCJ runtime, libgcj, which offers the core class libraries, a garbage collector, and a bytecode interpreter. libgcj can dynamically load and interpret class files, resulting in mixed compiled/interpreted applications. In order to port the compiled application on top of a RTOS (PaRTiKle), some native methods have been added. For instance, *getRealtimeClock* method that provides the time in nanoseconds to the user application.

The global architecture is drawn in figure 1. As it can be seen, a compiled application includes the GCJ runtime with the javax.realtime Classes and the native methods.

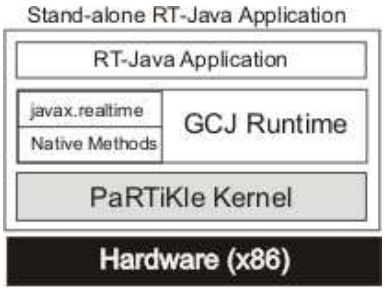


FIGURE 1: *PaRTiKle + GCJ Architecture*

To build a PaRTiKle's application running on stand alone or linux process mode the figure **??** sketches how it can do it, PaRTiKle provides a bash script, named mkkernel, for ease of building process, basically the steps performed by this script are: links the application against the user "c" library and the GCJ runtime then the script links the resulting object file together with the kernel object file to create the executable file containing all the components (.prtk).

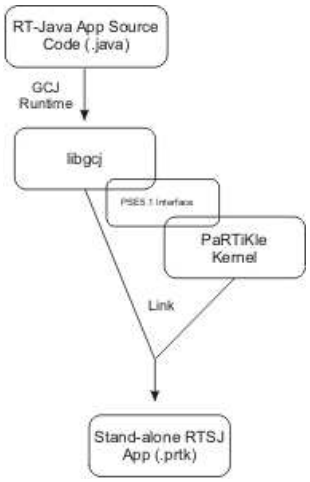


FIGURE 2: *Application Build Process*

3 PaRTiKle overview

PaRTiKle [16] is a new embedded real-time operating system designed to be as compatible with the POSIX.51 standard as possible. The native API is "C" POSIX threads. But also, it provides support for C++, Ada and Java (tasking, synchronisation, protected objects, exception handling, etc.). Besides POSIX compatibility, PaRTiKle also provides the RTLinux/GPL non-portable POSIX extensions; therefore, it should be possible to compile RTLinux/GPL applications on PaRTiKle to get all its benefits.

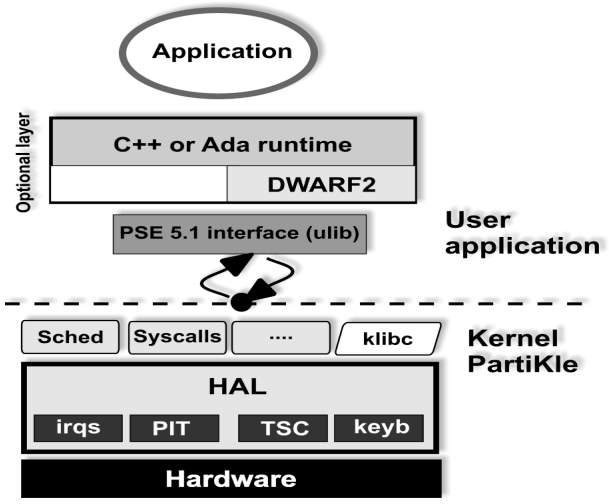
PaRTiKle has been designed bearing the following ideas in mind:

- being as portable, configurable and maintainable as possible.
- support for multiple execution environments, allowing, thus, to execute the same application code (without any modification) to be executed under different environments (so far): in a bare machine, a Linux regular process and as a hypervisor domain.
- support for multiple programming languages, currently PaRTiKle supports Ada, C, C++, Java (the current support of this last language is only supported when GCC compiler version 3.4 is used).

PaRTiKle has been designed to support applications with real-time requirements, providing features such as full preemptability, minimal interrupt latencies, and all the necessary synchronization primitives, scheduling policies, and interrupt handling mechanisms needed for this type of applications.

3.1 PaRTiKle architecture

Figure 3 sketches the PaRTiKle architecture. Contrarily to other small embedded RTOS (which are implemented as a library which is linked with the application), PaRTiKle has been designed as a real kernel with a clean and well defined separation between kernel and application execution spaces. All kernel services are provided via a single *entry_point*, which improves the robustness and also greatly simplifies the work to port PaRTiKle to other architectures and environments.



[1]PSE is the abbreviation of "Generic Environment Profile"
[2]PSE51: POSIX 1003.13^{TM}Minimal Realtime System Profile

FIGURE 3: *PaRTiKle's architecture*

PaRTiKle has eight functional blocks:

Kernel space

Core hardware drivers: Interrupt manager, and clock and timer drivers and virtual memory. This drivers are needed on all execution environments.

Peripheral drivers: Among others, keyboard and screen drivers.

Kernel C library: This is a small library (called klibc) of C functions used by kernel code.

POSIX functionality: Threads, mutexes, signals, timers, I/O, etc.

System call interface: Implements the system call mechanism.

User space

POSIX C library: See [16] for a more complete description of the available API.

DWARF2 support: DWARF is a debugging mechanism which is also used to manage high level exception handling (e.i. `try` and `catch` blocks in C++).

Languages support: Runtime library of the supported languages.

The Core hardware drivers jointly with the peripheral drivers form the hardware abstraction layer (HAL).

It is important to note that the PaRTiKle kernel implements a minimal C library. This minimal C library does not shared any line of code with the libraries used by the application.

The application space is composed by a C library, PSE51[12] Compliant, which relays on the services provided by the kernel via the system call interface.

Also, the support for Ada, C++ and Java applications are provided at user space level. Additionally, to support all these run-times, PaRTiKle implements the Debugging Information Format DWARF2, which will be also linked with the application if required. The application is linked on the top of all these support libraries.

3.2 Execution Environments

PaRTiKle has been designed to be run under several different execution environments. So far, three different execution environments are available, all of them for the x86 architecture: 1) on a bare machine, 2) as a Linux regular process and 3) as a domain of XtratuM [12], [13], giving this last alternative the possibility of executing PaRTiKle jointly with another general purpose operating system (Linux so far).

1. On a bare machine: PaRTiKle is the only system executed in the system, it is in charge of managing the whole hardware. This environment is the best option for application with only hard-real time constraints, and small footprint.

2. As a Linux regular process: This environment is intented for testing purposes. The generated code is executed as a regular Linux process. PaRTiKle still has direct access to the hardware, however, real-time constraints are not guaranteed whatsoever.

3. As a XtratuM domain: XtratuM is an hypervisor that provides hardware virtualisation and enables the execution of several kernels (or runtimes) concurrently. PaRTiKle can be built to be XtratuM aware and then loaded using the XtratuM domain's loader `xmctl`:

4 RTSJ Implementations Details

The implementation supports most of the RTSJ features such as real-time threads, periodic threads, asynchronous event handlers, timers, POSIX signals, etc. By default all memory allocations are done in the ImmortalMemory area, which means that object allocated are never freed, the garbage collector is not supported. This attribute is concerned with ensuring that the software will not access unintended or disallowed memory locations, and ensuring that the use of memory space will be predictable and bounded.

Three kinds of threads are supported periodic, sporadic and realtime *Realtimethread*. Periodic threads are implemented by means of the PeriodicThread Class. This implementation defines time-triggered and, transparently, invokes the *waitForNextPeriod* method of the RealtimeThread class to delay until its next periods. The constructor of *periodicThreads Class* is shown below, Figure 4:

```
public PeriodicThread (PriorityParameters(
        prioParams,
        PeriodicParameters periodicParams,
        java.lang.Runnable logic)
        )
```

FIGURE 4: *Constructor of PeriodicThread*

Threads are dispatched usign the scheduling policy provided by the RTOS (PartiKle). The scheduling policy is based on fixed priorities. Section 5 shows the performance of this policy overhead. Figure 5 shows an example for using periodicsThreads:

```
new PeriodicThread(new PriorityParameters(ph1),
        new PeriodicParameters(init, period), h1).start();
```

FIGURE 5: *Periodic Threads Create*

From the point of view of time management, it is needed a clock with high resolution and granularity. This need is achieved with the CLOCK_REALTIME data structure provided by PaRTiKle. The timespec variable takes the clock and returns to time value in milliseconds and nanoseconds. In order to be used in a coherent way on Java, the time value is normalised. Figure 6 shows the native code of the getRealtimeClock method.

```
struct timespec tp;
clock_gettime (CLOCK_REALTIME, &tp);
jlong millis = tp.tv_sec * 1000;
jint nanos = tp. tv_nsec;
dest->set(millis, nanos);
```

FIGURE 6: *getRealtimeClock implementation*

The delay function (nanosleep) is also implemented by a method shown in figure 7

```
_Jv_nanosleep (jint secs, jlong nanos)
{
    struct timespec delay = {secs, nanos};
    nanosleep(&delay, (timespec*)NULL);
}
```

FIGURE 7: *Nanosleep Method*

On the other hand, PaRTikle implements a priority pre-emptive scheduler. We translate the RTJava priorities to PaRTiKle OS. PaRTiKle (Posix) has an scheme for priorities in the range of 0 to 1023 (in the PaRTiKle implementation). Priorities are moved to priority=28-newPriority+1.

5 Experimental Evaluation

Several benchmarks have been defined and implemented to measure the performance of the proposed implementation. The evaluated features in the comparative study are based on the main aspects of the RTSJ. We mainly focus the test on efficiency and predictability measures which are important considerations when designing a real-time applications. All benchmarks in this section were run on an INTEL Pentium IV running at 2.6GHz, with 512 MB of memory. The operating system is PaRTiKle running on stand-alone (bare-machine). There is a small but growing body of work on measuring performance characteristics of Real-time Java [1], [4], [5], [6], [9]. Unfortunately the comparison of different implementations is difficult due to the proprietary nature of many systems. We only have used of jRate (RTJPerf benchmark) some characteristics. Specifically, the evaluated aspects are: RealtimeThread startup latency, context switch using **yield()** method and context switching measured on a varying number of threads, latency memory allocation and latency creating RealtimeThread.

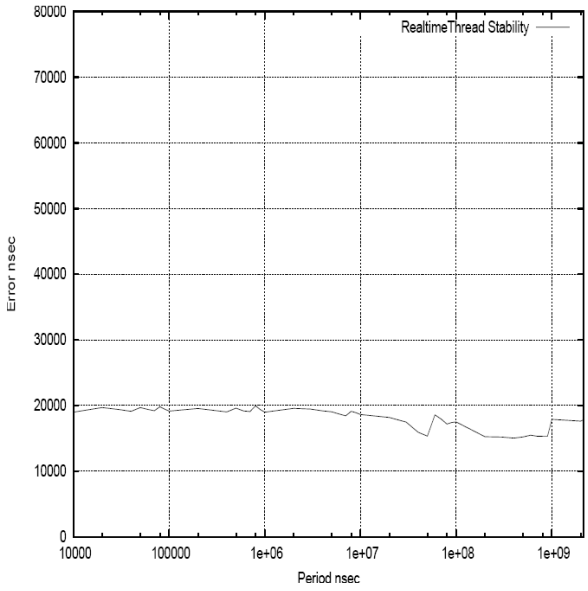


FIGURE 8: *RealtimeThread Stability*

5.1 RealtimeThread Startup Latency

In order to evaluate the stability of the implementation, we have evaluated the latency. This evaluation has consisted in a periodic task which program a delay and waits for it. As soon as the task is executed, the task reads the clock value and calculates the difference between the programmed delay and the real clock, then the task period is increased in order to evaluate the implementation latency from 100 nanoseconds to 1 seconds. With this measure, we show the stability of the response to a delay independently of the frequency.

As can be seen in the plot (figure 8), there is a constant delay of 20 microseconds due to the timer arm. Nonetheless, lower periods show higher latencies because of the time taken to program the system timer.

5.2 Context Switch

High levels of thread context switching overhead can significantly degrade application responsiveness and determinism. Minimizing this overhead is therefore an important goal of any runtime environment for real-time embedded systems. To measure context switching overhead, we provided two tests.

5.3 Yield Latency

Two threads with the same priority are started. The first one repeatedly gets the current time and yields. The second thread gets the current time once it is scheduled. We measure the interval between the first thread yields and the second thread is scheduled. The results of the figure 9 show an average of approximately 0.9 microseconds. The maximum time to switch between threads was approximately 1.9 microseconds, which is better than the obtained in most of the RTSJ implemented.

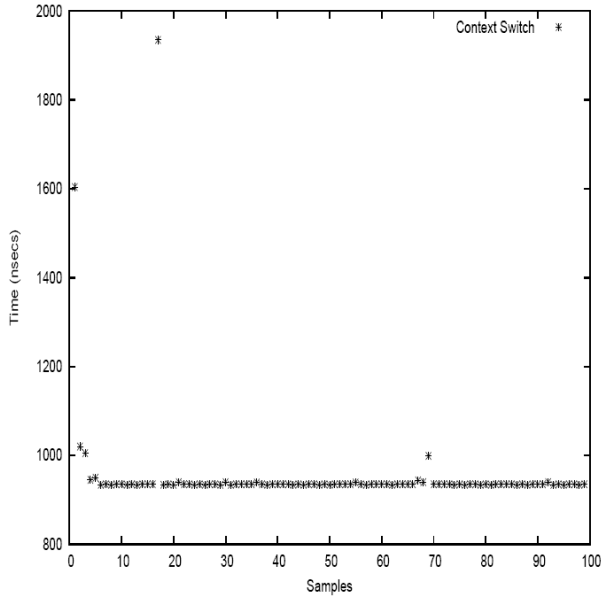


FIGURE 9: *Context Switch*

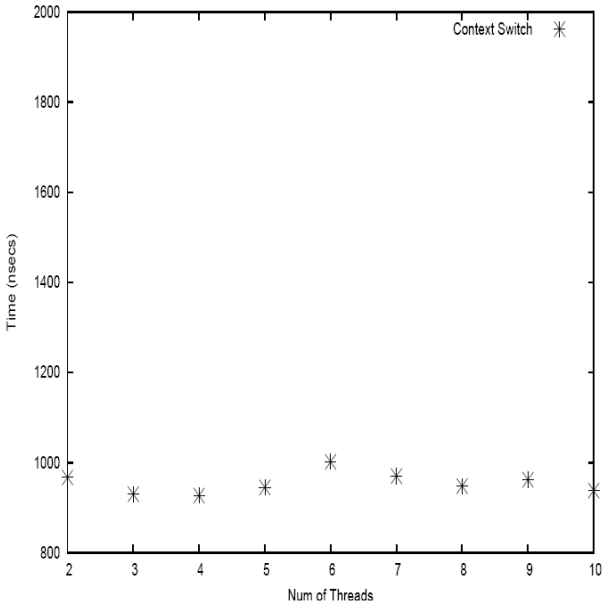


FIGURE 10: *Context Switch between Threads*

5.3.1 Context Switching Between Number of RealtimeThreads

The measurements are taken by running a number of threads each of which call their yield() method 100 times. The number of context switches is 10 times the number of RealtimeThreads. In this test, the time for context switching was measured on a time interval with a significant number of threads. The goal is to check if the number of threads leads to an additional overhead in context switching. Figure **??** shows these results. As it can be seen, the context switch times remain almost constant as the number of threads increase. This shows that both implementations perform context switching efficiently, with our implementation.

5.4 Latency Memory Allocation

Implicit dynamic memory allocation is strongly discouraged in many real-time embedded systems to minimize memory leaks, latency, and nonpredictability due to garbage collection. Explicit memory allocation is supported by PaRTiKle OS. The measure of the allocation time and its dependency on the size of the allocated memory is a good measure of the efficiency of allocated memory implementations. To measure the allocation time and its dependency on the size of the memory allocation request, we provide a test that allocates fixed-sized objects repeatedly. To control the size of the object allocated, the test allocates an array of bytes of different sizes.

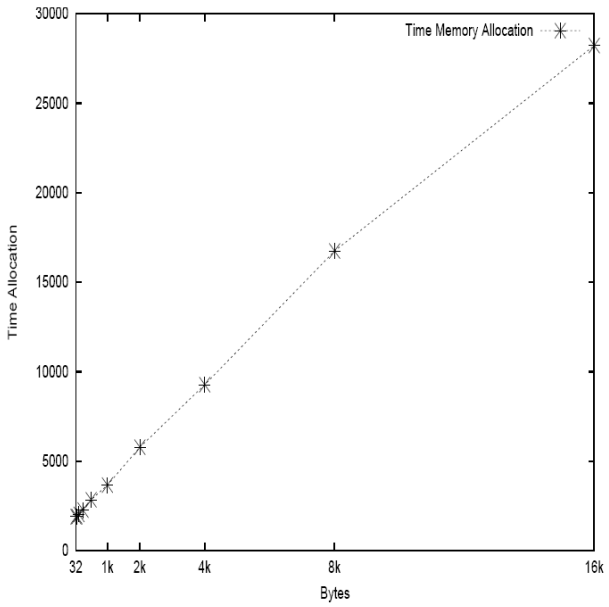


FIGURE 11: *Latency Memory Allocation*

It is possible to determine the allocation time associated with the supported dynamic memory allocation provided by PaRTiKle. The test measures the temporal cost of the allocation by measuring the clock before and after the memory allocation code. This test is run for object sizes from 32 to 128 Kbytes (figure 11).

The average time to create 32 bytes objects is less than 1904 nanoseconds. Regardless of the RTSJ implementation, the allocator provides linear time allocation with respect to the allocated memory size. These results show a very good time which is better than other RTSJ implementations.

5.5 Thread Creation Latency Test

This test measures the time needed to create a thread. Thread creation in RTSJ platform involves many operation and checks concerning memory areas, memory stack and memory allocation. Thus the thread creation time is affected by the memory area implementation. The results obtained for this test are presented in the figure 12. The same of Latency Memory Allocation, thread creation latency test provide linear time allocation with respect to the allocated memory size. It is important to mention that task allocated in this test are not deallocted.

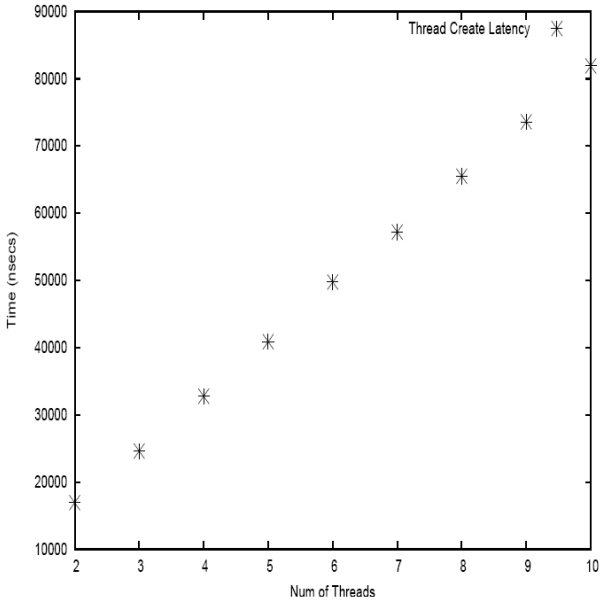


FIGURE 12: *Thread Create Latency*

5.6 Periodics Threads Test

For each periodic thread a period is specified. During the test, the accuracy of the periods are evaluated. This is achieved by executing 100 periods of a single thread with a period of 10ms, 100ms and 1000ms. In order to increase the graph readability, a sample of 100 periods is shown in the plot.

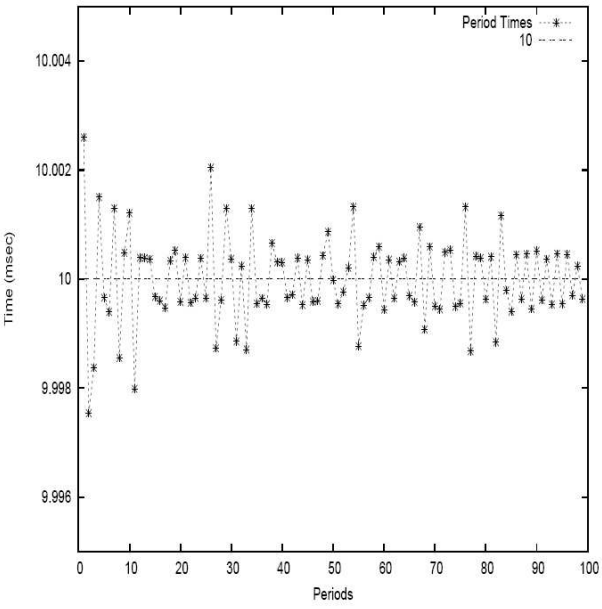


FIGURE 13: *Periodic Thread (10ms)*

Figure 13, 14 and 15 show the results for periods of 10ms, 100ms and 1000ms respectively. Looking at

the graphs, accurate period durations are very constant and the average is close to the nominal period.

6 Conclusions

This paper has presented an implementation of the RTSJ. The implementation is based on a real-time operating system (PartiKle) which provides POSIX 5.1 standard interface.

The RTSJ implementation has been constrained to the functionalities defined in the Ravenscar Profile. To keep the proposed API as close as possible to the RTSJ, minor modifications were provided, for instance the PeriodicThread Class that helps to developers use a Periodic thread directly just with its declaration and start invocation.

In order to validate and evaluate the performance of this implementation, several test have been designed and implemented. All the test show that the performance of this implementation is very efficent achieving very good results in object allocation, stability, context switching and scheduling overhead. This evaluation demonstrates the efficiency of the RTSJ implementation can aid research in advancing the use of Java in real-time systems.

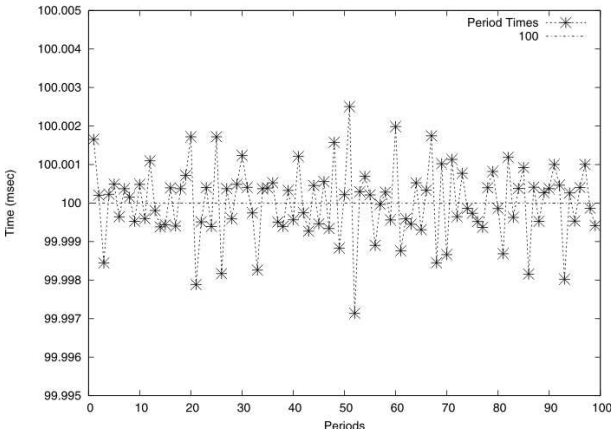


FIGURE 14: *Periodic Thread (100ms)*

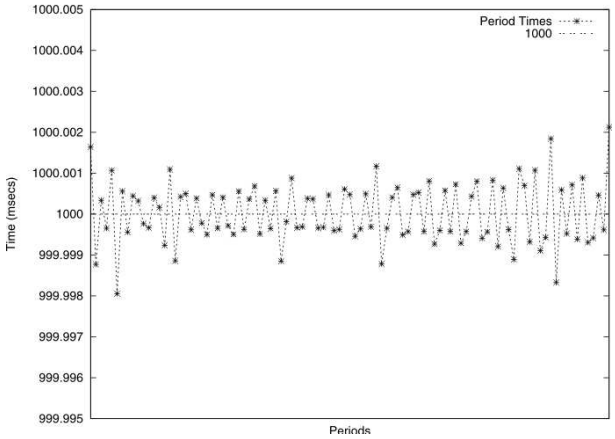


FIGURE 15: *Periodic Thread (1000ms)*

References

[1] N. A. and B. E. , *Rtsj memory areas and their affects on the performance of a flight-like attitude control* . In JTRES, pages 508519, 2003.

[2] Aicas. Jamaicavm 2.6 user documentation.

[3] G. Bollella and J. Gosling, *The real-time specification for java*. IEEE Computer, 33(6):47-54,2000.

[4] M. K. G. Bollella G, Loh k and W. T., *Experiences and benchmarking with jtime*. In JTRES, pages 534549, 2003.

[5] A. Corsaro and S. D., *The Design and Performance of the jRate Real-Time Java Implementation* . On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE

[6] A. Corsaro and S. D., *Evaluating real-time java features and performance for real-time embedded systems*. In RTAS, 2002.

[7] RTLinux, Available at www.rtlinux-gpl.org

[8] J. M. Enery, D. Hickey, and M. Boubekeur, *Empirical evaluation of two main-stream rtsj implementations*. In JTRES, pages 4754, 2007.

[9] T. Higuera-Toledano and I. V., *Analyzing the performance of memory management in rtsj*. Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2002

[10] Irvine, Available at http://jrate.sourceforge.net/

[11] A. W. Jagun Kwon and S. King, *Ravenscar-Java: A High Integrity Profile for Realtime-Java*. Department of Computer Science, University of York, UK, 2002.

[12] M. Masmano, I. Ripoll, and A. Crespo, *Introduction to XtratuM*. Available at http://www.xtratum.org/files/documents /xtratum_intro.pdf, 2005.

[13] M. Masmano, I. Ripoll, and A. Crespo, *An overview of the XtratuM nanokernel*. In Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), 2005.

[14] Sun Microsystems, *Java rts 1.0.0 release notes*

[15] MIT, *Flex*. flex-compiler.lcs.mit.edu

[16] S. Peiro, M. Masmano, I. Ripoll, and A. Crespo,*PaRTiKle OS, a replacement of the core of RTLinux*. in 9th Real-Time Linux Worshop, 2007

[17] TimeSys, *Linux RTOS Standars Edition*. Available at www.timesys.com