A Hypervisor Based Platform to Support Real-Time Safety Critical Embedded Java Applications

Apolinar González *, Walter Mata *, Alfonés Crespo ‡
Miguel Masmano ‡, José Félix *, Alvaro Aburto *
*University of Colima - Mexico
‡Polytechnic University of Valence, Spain

Abstract

A small footprint, low latency platform for real-time embedded safety critical Java applications has been developed. The platform consists of a hypervisor, operating system, and Java compiler. This paper describes the design, implementation and analysis of the platform to support real-time safety critical embedded applications using real-time java. The hypervisor, has been created by the group at Universidad Politécnica de Valencia and others, has been partially funded by FRESCOR European project and the Centre National d’Etudes Spatiales in France (CNES, the French space agency), and will be used on LEON2 processor. The operating system was also developed by the group at Universidad Politécnica de Valencia and others too, and is evolved from the original RTLinux project started at this institution. The Java compiler for Real-Time embedded Java applications was adapted and developed by the group at the University of Colima, Mexico, along with others, it has been partially funded for D2ARS Ibero-American project, using the jRate (A Real-Time Java ahead-of-time compiler) and GCJ (GNU Compiler for Java). The details of the parts of this platform are given, along with performance comparisons and analysis.

Keywords: Real-Time Java, Embedded Systems, Real-Time Hypervisors.

1. Introduction

The increasing presence of embedded systems in products and services creates huge opportunities for the future in different areas. Embedded systems already play an important role not only in consumer electronics but also in many important and safety-critical systems. As a result, there is growing scientific interest in conceptual and practical tools for development of embedded systems. These systems are becoming more and more popular for a wide range of applications; each day, our lives become more dependent on embedded systems, digital information technology that is embedded in our environment. More than 98% of processors used today are in embedded systems, and are no longer visible to the customer as a computer in the ordinary sense. This includes not only safety-critical applications such as controllers for automotive, railway, aircraft, aerospace, health care and medical devices, but also communications, mobile systems, environment, home automation, mobiles phones, PDAs, DVD players, cameras, etc. All of these have wide-ranging impacts on society, including security, privacy, and modes of working and living[1], [2], [3], [4], [5], [6].

An embedded system is a special-purpose computer system that is used for a particular task. The special computer system is usually less powerful than general-purpose systems, although some
exceptions do exist where embedded systems are very powerful and complicated. Some of the usual characteristics of embedded systems include these: contain a processing engine, such as a general-purpose microprocessor, typically designed for specific application, includes a simple (or not) user interface, usually low power consumption CPU with a limited amount of memory systems (for example, it has a small memory footprint without hard drive) and usually is not used as a general-purpose computing platform [4], [7].

Traditionally, the development of embedded software has had to take into account several constraints stemming from either the software or the embedded architecture. The development of the hardware and software for these systems require appropriate design, analysis and development tools. The underlying architecture also plays an important role, special design related challenges come from the specialization and customization of target platform in their use for embedded systems. The challenge is to maintain some degree of flexibility to increase the reuse of software components. The design problem is that many embedded software applications have inherent real-time constraints that need to be supported by the underlying operating system and programming language.

The physical constraints arise through two ways that computational processes interact with the physical world: reaction to a physical environment and execution on a physical platform. Common reaction constraints specify deadlines, throughput, and jitter and originate from behavioral requirements. Common execution constraints bound available processor speeds, power, and hardware failure rates and originate from implementation choices [7]. The operating system is responsible for ensuring a predictable execution behavior of the application to allow an off-line guarantee of the required performance [2]. Regarding the architecture, limited and customized resource features for example several embedded devices are designed under space, weight, and energy constraints imposed by the specific application, which increases the complexity of building an embedded system. In order to make these devices cost-effective, it is mandatory to make very efficient use of the computational resources, gaining control of the interplay between computation and both kinds of constraints to meet a given set of requirements on a given implementation platform.

In secure applications domains, for example in aeronautics or current avionics systems, where applications are generally composed of several levels of criticality and should be isolated from each other, so that their high integrity should not be involved because errors or failures in other applications. A common approach to isolation has been based on the Integrated Modular Avionics (IMA) concept, which proposes an integrated architecture with application software portable across an assembly of common hardware modules. IMA architecture demands multiple requirements on the underlying operating system. IMA design brings in the notion of time, space and resource partitioning. IMA architecture contains: A partitioning kernel that runs in supervisor mode and provides Time and Space Partitioning (TSP) and a set of services. Within each partition, the applications execute in user mode completely isolated from other applications. The operating system makes each application behave as if it had exclusive use of the platform when, in fact, it is sharing the platform with many other applications. A common approach to isolation has been based on using a federated architecture, i.e. on allocating different applications to different computers. However, the development of embedded applications actually is entering into a new domain with the availability of new high-speed processors and low cost on-chip memory. As the result of these new developments in hardware, there is an interest in enabling multiple applications to share a single processor and memory. To facilitate such a model the execution time and memory space of each application must be protected from other applications in the system. Partitioning operating
systems represent the future of secure systems. They have evolved to fulfill security requirements where predictability is extremely important. In a partitioning operating system, memory is divided between statically allocated partitions in a fixed manner. The idea is to take a processor and make it pretend to be several processors by completely isolating the subsystems. This concept is well known as Virtualisation. The importance of Virtualisation in the embedded computing area is currently emerging. The industry is making rapid advances in system Virtualisation, for server consolidation and for improving system maintenance and management.

This paper introduces a small footprint, low latency platform for real-time embedded safety critical Java application. The platform consists of a hypervisor, operating system, and Java compiler. The global architecture is shown in figure 1, where the XtratuM hypervisor is executed directly on the hardware architecture. There, a currently support exists for x86, LEON2, and PPC and there is working currently being done on the migration to ARM and XScale. XtratuM completely isolates the different domains, permitting communication via message queues, hypercalls, and by shared memory. The different operating systems can be consequently be executed in an independent manner and XtratuM in real-time controls the execution of each one of them. The current operating systems supported by XtratuM are PaRTiKle, Linux, LithOS and RTEMS; and in the future there will be more real-time operating systems supported. On the PaRTiKle operating system, it is possible to execute different applications in languages such as C, C++, ADA and Java, and especially for Java, the Real-Time Java (RTJ) porting has been done which supports Real-Time Specification for Java (RTSJ). To support RTJ over the PaRTiKle OS, we have modified some of the GNU Compiler for the Java platform libraries and adapted the form of calls to the operating system [8].

![Figure 1: Global Architecture](image)

The remainder of the paper is organized as follows; in sections 2 we describe the related works, 3 and 4 we describe the XtratuM and PaRTiKle’s architecture respectively; in section 4 we present the principal details of the implementation of JR ate on PaRTiKle; in section 5 we show the analysis of the performance advantages, and the minimal memory requirement for this platform. Finally, we conclude summarizing the major points of the paper and outlining future research directions.
2. Related Works

Virtualisation has been introduced to computing architecture by IBM in the 1960s; the advances in the processing power of the desktop processors in the middle of the 90’s opened the positivity to use it in the PC market. Most of the recent advances on Virtualisation have been done in the desktop systems. Xen [9], VMware [10], Sun VirtualBox and Innotek [11] are just four examples out of many more, developing the technologies. In [12] show the implementation and simulation benchmark of HyperMonitor that is a multi-platform lightweight monitor based on hardware virtualisation which allows a high-privileged and transparent execution environment to monitor several behaviors in different operating systems. Virtualisation has recently become popular in the embedded-systems space. The embedded market is now ready to take advantage of this promising technology; however, transferring results from the desktop systems to embedded systems is not as direct as it may seem. Virtualisation is a term used to provide abstractions of computer resources. Hypervisor is the term to refer to the software layer providing this virtualisation. This software layer (in combination with hardware mechanisms) allows to run several independent (spatial and temporal isolated) operating systems in a single computer. In the literature several names can be found to refer to the operating system running on top of a hypervisor: partition, domain or guest operating systems. When the virtualisation technique is used in real-time embedded systems, the performance plays the most important role. One of the most important features of a hypervisor is that it should introduce a low overhead; therefore the performance of applications executed in the virtualised system should be close to the same as for applications running on the native system.

The underlying architecture also plays and important role. Special design related challenges come from the specialization and customization of target platforms in their use for embedded systems. The challenge is to maintain some degree of flexibility 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 environment, the benefits 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) [13], [14], [15]. It defines a set of extensions to the Java virtual machine and the class libraries that facilitate real-time programming.

There are new technologies to meet the demands of high performance and low power consumption in all computing domains such as multi-cores, which are not feasible to be used in embedded safety critical environments with hard real-time constraints; however [16] propose a solution for the RTOS support of parallel hard-real time applications on the MERASA multi-core architecture, which fulfills the requirements for time-bound execution of parallel hard real-time tasks focused on thread control with synchronisation mechanisms, memory management and resource management requirements. On the other hand to facilitate the development of real time embedded system for safety critical systems, we address two different challenges not met in existing embedded systems. Our first focus is on virtualisation. Completely isolating the subsystem allows the ability to run several independent operating systems. To support safety critical real-time applications, the hypervisor must execute the different subdomains with real-time characteristics. Our second challenge is to enable the use of Java for real-time applications, porting and adapting the
implementation of real-time Java based on JRate [17] running on PaRTiKle [18], which is executed directly on the XtratuM hypervisor [19][20]. Some characteristics of JRate have been adapted to obtain small footprint applications on the order of hundreds of kilobytes, which allow us to use this stack on limited hardware platforms. Functions to facilitate the development of periodic threads without time deviations have also implemented. Characteristics to do operations on times have been added, such as time granularity, task scheduling, and others, have been moved so that they can be administered through the PaRTiKle OS.

3. XtratuM Architecture Overview

XtratuM is a hypervisor which provides a framework to run several operating systems (or real-time executives) in a robust partitioned environment. XtratuM can be used to build MILS (Multiple Independent Levels of Security) architecture. Usually, operating systems are internally structured as a set of building blocks, user interface layer, a set of drivers, memory manager, virtual file systems infrastructure, network stack, etc. In this scenario, XtratuM can be considered as a small part of the lowest operating system layers. The most exciting feature is the capability to share the same hardware among several operating systems running concurrently. XtratuM makes it possible to run two or more operating systems at the same time in the same computer.

In order to run several domains concurrently, each domain has to be ported to the XtratuM infrastructure. In particular, the boot sequence of each operating system has to be removed because XtratuM will take care of it; and the interrupt and timer drivers have to be replaced by calls to the XtratuM API.

XtratuM has been implemented following a monolithic approach, running all its services in processor privileged mode and in a single memory space; being all the services directly reachable from any part of the hypervisor, the figure 2 shows the internal architecture of XtratuM.
The main idea behind the design of the architecture of XtratuM is to virtualize the minimal possible parts of the hardware to achieve the execution, in a concurrent way, of several Operating Systems. However, unlike some existing nanokernels (for example, the L4-kernels family) XtratuM does not virtualize the whole hardware architecture, but it just multiplexes the most essential parts of the hardware to execute in a concurrent ways several Operating Systems. Each OS should be aware how to use the parts of the hardware which have not been virtualized. XtratuM basically offers the following virtualizations for the guest OS:

- **Interrupts**: Taking over interrupts on a computer is a synonymous with controlling the whole machine. Once XtratuM is started up, it is the only entity which really controls hardware interrupts and, of course, the only one that is able to disable/enable real interrupts. An API is offered, which enables a guest OS to deal with virtual interrupts which basically allow enabling/disabling virtual interrupts, installing interrupt and exception handlers, and so on.

- **Timer**: Providing a timer is not necessary to execute concurrently several operating systems. However, to simplify the porting of an OS, XtratuM provides at least one virtual timer. The exact number of timers implemented by XtratuM depends on the available number of hardware timers. For example, when XtratuM is executed in the Intel x86 architecture, supposing the APIC timer is available, it will offer two different virtual timers: the classic PIT and the APIC timer. Besides, in order to work with these virtual timers, XtratuM also provides a high-level API to deal with them.

The code of XtratuM is structured up into three layers:

- **Hardware-dependent layer** implements the set of drivers required to manage the strictly necessary hardware: processor, interrupts, hardware clocks, hardware timers, paging, etc. This layer is isolated from the rest through the Hardware Abstraction Layer (HAL). Thus, the HAL hides the complexity of the underlying hardware by offering a high-level abstraction of it (for instance, a ktimer is the common abstraction to all the hardware timers).

- **Internal-service layer** internal supporting services. Those services are not available to the partitions. This layer includes a minimal C library (KLibC) which provides the strictly required set of standard C functions (e.g. strcpy, memcpy, sprintf) and a bundle of data structures (e.g. a list, a heap and a queue). The boot code is also part of the internal services. On limited memory systems, once the hypervisor has completed the boot this memory can be freed and reused.

- **Virtualization-service layer** provides the services required to support the virtualization and the paravirtualization services, which are provided via the hypercall mechanism to partitions. Some of these services are also used from other XtratuM modules. For example, the physical memory manager is in charge of allocating free physical memory pages to both, XtratuM and partitions.

Partition support: The library libxm contains the wrappers for the XtratuM services. There are two basic communication mechanisms between the partition and the hypervisor: hypercalls and shared
4. PaRTiKle OS Overview

PaRTiKle [18], [21] is an embedded real-time operating system designed to be as compatible with the POSIX 5.1 standard as possible. The native API is C POSIX threads. But, it also provides support for C++, Ada and Java (tasking, synchronization, 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++ and Java

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.

4.1. PaRTiKle Architecture Overview

Figure 3 shows 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.

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 share any line of code with the libraries used by the application.

The application space is composed of a C library, PSE51 (PSE is the abbreviation of “Generic Environment Profile” PSE51: POSIX 1003.13 Minimal Real-time System Profile) Compliant, which relies 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. PaRTiKle has eight functional blocks, divided between kernel space and user space.
4.2. Kernel and User Space

The Kernel Space is divided basically in five kinds of libraries or drivers:

1. Core hardware drivers: Interrupt manager, and clock and timer drivers and virtual memory. These drivers are needed on all execution environments.
2. Peripheral drivers: Among others, keyboard and screen drivers.
3. Kernel C library: This is a small library (called klibc) of C functions used by kernel code.
4. POSIX functionality: Threads, mutexes, signals, timers, I/O, etc.
   - System call interface which implements the system call mechanism.

The User Space consists of:

1. POSIX C library: See [22] for a more complete description of the available API.
2. DWARF2 support: DWARF is a debugging mechanism which is also used to manage high level exception handling (i.e. try and catch blocks in C++).
3. Languages support: Runtime library of the supported languages.

4.3. 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 [23], 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 intended 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 provides hardware virtualization and enables the execution of several kernels (or run-times) concurrently. PaRTiKle can be built to be XtratuM aware.

5. Porting Real-Time Java to ParTiKle OS

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. LibGCJ has also been modified to obtain a footprint that is a number of kilobytes smaller, permitting the possibility to export to architectures and applications with restricted computational and memory resources.

5.1. Global Architecture

The global architecture is drawn in figure 4. As it can be seen, a compiled application includes the GCJ runtime with the javax.realtime classes and the native methods.

To build a PaRTiKle’s application running on XtratuM mode, the figure 5 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 “c” user 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).

![Image of RTJ Application Architecture]

Figure 4. RTJ Application Architecture
5.2. LibGCJ Minimum

GCJ, which is the GCC Java compiler, links with the LibGCJ by default, which includes a complete java runtime on the order of some megabytes with a set of characteristics that are not necessary for small platforms. The idea is to remove the extra functions that are not necessary for safety critical systems and also not necessary for real-time systems with hard timing restrictions. We started from scratch, copying the source code necessary to the directories where we had installed GCG on PaRTiKle. The code was compiled generating a new, reduced, version of LibGCJ. Later, doing the link with PaRTiKle and XtratuM, and finally, executing. The size obtained is in the range of hundreds of kilobytes for the complete application, which is composed of the application code (code developed for the end user), the PaRTiKle operating system, and the XtratuM hypervisor.

The reduced version of LibGCJ is a subset of the runtime support and the CLASSPATH. Finally, this version of the library has all of the support to execute applications using the porting and adaption of jRate on the PaRTiKle OS, a subset of java.lang and java.io, support for thread execution, and OS interfaces. The garbage collector, support for graphical interfaces, runtime classloading, bytecode interpretation, reflection, finalization, serialization, file and network I/O, and many parts of java.lang, java.util, and java.io, that were not considered essential, have been removed, for real-time and safety critical applications.

5.3. Some RTSJ Implementations Details

Three kinds of threads are supported periodic, sporadic and real-time (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:
periodicThread PriorityParameters(
    PrioParams prioParams,
    PeriodicParameters periodicParams,
    Java.lang.Runnable logic
)

Threads are dispatched using the scheduling policy provided by the RTOS (PaRTiKle). The scheduling policy is based on fixed priorities. Section 6 shows the performance of this policy overhead.

From the point of view of time management, a clock with high resolution and granularity is needed. 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 seconds and nanoseconds.

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

In order to be used in a coherent way on Java, the time value is normalized. The code list shows the native code of the getRealtimeClock method.

The delay function (nanosleep) is also implemented by a method shown:

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

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

### 6. 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 and we have opted for the indirect measurements approach as it is simpler to implement and does not require additional hardware and software over the hypervisor. We mainly focus the test on efficiency and predictability measures which are important considerations when designing 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 XtratuM and Linux 2.6.

There is a small but growing body of work on measuring performance characteristics of Real-time Java [24], [25], [26], [27]. Real Time Java (RTJ) based on jRate has been evaluated and compared
with other RTSJ implementations over different operating systems [17]. In this work, we make a comparison of our RTJ porting based on JRate running over Linux 2.6 and adapted for PaRTiKle and XtratuM. Specifically, the evaluated aspects are: RealtimeThread startup latency, context switch using the yield() method and context switching measured on a varying number of threads, latency memory allocation and latency creating RealtimeThread.

6.1. 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 in dependently of the frequency. As can be seen in the plot (figure 6), there is a constant delay of 19 microseconds for RTJ over PaRTiKle OS over XtratuM and 56 microseconds for RTJ over Linux 2.6 and similarly for RTJ over Linux/XtratuM. Nonetheless, lower periods show higher latencies because of the time taken to program the system timer. The results show that the use of a dedicated operating system is more efficient than a general purpose and the hypervisor does not introduce significant overhead.

![Figure 6. RealtimeThread Stability](image)

6.2. 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.

6.2.1. 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 7 shows an average of approximately 0.9 microseconds and 1.1 microseconds, which is better than that obtained with RTJ over Linux 2.6 (approximately 1.6 microseconds) and RTJ over Linux 2.6 /XtratuM (1.8 microseconds) respectively. Similar to the previous case, the hypervisor overhead is negligible.

Figure 7: Context Switch

6.2.2. Context Switch between 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 8 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.
6.3. 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 9). For the implementation of RTJ on PaRTiKle/XtratuM, for now, we have only included the immortal memory, in our future works, we will integrate a constant-time dynamic storage allocator memory model TLSF (two-level segregated fit) [22], [23] that is supported directly by the PaRTiKle OS and permits allocation of areas of memory dynamic in constant time.

![Latency Memory Allocation](image)

Figure 9: 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.

The average time to create 32 bytes objects is less than 1600 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.

6.4 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 10. 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 those tasks allocated in this test are not deallocated.

6.5 PeriodicThread Test

For each PeriodicThread a period is specified. During the test, the accuracy of the periods is 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 11 and 12 show the results for periods of 10ms and 100ms respectively. Looking at the graphs, the RTJ implementation over PaRTiKle/XtratuM accurate period durations are very constant
and the average is close to the nominal period and show that the cost of virtualisation is negligible and that the set RTJ + PaRTiKle + XtratuM has a good performance compared to implementations with general operating systems. These results can be considered very satisfactory, taking into account the advantages of having several applications running in an isolated (temporal and spatial) partitioning framework.

7. Conclusions

Virtualisation has many aspects attractive to the embedded world, but on its own is a poor match for modern embedded systems. The main objective of XtratuM is to simplify operating system portability so that a XtratuM aware OS can be easily ported to any environment. XtratuM allows execute several applications (application or real-time operating system plus applications) with temporal and spatial isolation. Porting and adapting of the real-time Java was implemented on PaRTiKle OS, which provides POSIX 5.1 standard interface and is executed directly on the XtratuM hypervisor.

The combination of a compact Java environment, the PaRTiKle OS, and XtratuM hypervisor, has resulted in a very small footprint, low latency, and highly reliable platform for time critical Java applications, allowing running different applications with different levels of criticality on the same platform. XtratuM hypervisor, and PaRTiKle OS implement time and space isolation between partitions and RTSJ provides timing predictability within each partition. In order to validate and evaluate the performance of this implementation, several tests have been designed and performed. All the testing realized in this work shows that the performance of this implementation is very efficient, achieving very good results in object allocation, stability, context switching and scheduling overhead. Maximum latencies of 0.9 microseconds have been achieved for thread context switch, linear characteristic for memory allocation, and we have also obtained a maximum jitter of 2 microseconds for periodic tasks. This evaluation demonstrates the efficiency of our hypervisor-based platform to support Real-Time safety critical embedded Java applications, which can aid research in advancing the use of Java in real-time systems. These results can be considered very satisfactory, taking into account the advantages of having several applications running in an isolated (temporal and spatial) partitioning framework. Similarly, tests show that the cost of
virtualisation is negligible and this hypervisor does not introduce significant overhead, and the set of RTJ + PaRTiKle + XtratuM has good performance.

8. Future Work

In the future we will provide our software architecture for many more hardware architectures such as Arm, XScale, PPC, etc., and we will implement the memory model TLF that is supported by PaRTiKle operating system, for which we must integrate the adaption and implementation of RTSJ that we realized. And, considering measurements, we will also be instrumenting hardware to measure interrupt latency, which we have not yet measured.

9. References


