Implementation and Evaluation of Real-Time Java Threads*

Akihiko Miyoshi
Keio Research Institute at SFC
Keio University
5322 Endoh Fujisawa Kanagawa, Japan
miyos@sfc.keio.ac.jp

Takuro Kitayama
Keio Research Institute at SFC
Keio University
5322 Endoh Fujisawa Kanagawa, Japan
takuro@sfc.keio.ac.jp

Hideyuki Tokuda
Faculty of Environmental Information
Keio University
5322 Endoh Fujisawa Kanagawa, Japan
hxt@sfc.keio.ac.jp

Abstract

Java[†] *has many benefits such as security in distributed environments, reusability of code, and portability because it is architecture neutral. From those characteristics, Java is beginning to be used in many new environments. Even though Java provides various advantages, it still has problems which must be solved. One issue is that, there are often real-time constraints that should be met in those applications. Current Java execution environment and language specification cannot satisfy those requirements. Our research focuses on the current limits of the Java language and its execution environment, and to seek the possibility for real-time using Java and the technologies we have available today. To investigate issues in real-time Java, we have implemented a prototype real-time Java environment which provides real-time Java threads and real-time synchronization mechanism. Then we evaluated its performance. The results indicate that timing requirements are better met and unbounded priority inversion is avoided in our system.*

1. Introduction

Java[3] was developed by Sun Microsystems and has become increasingly popular in the past few years. It is an object-oriented programming language similar to C++ and was designed to be used in world-wide distributed computing environments such as the World Wide Web.

From its characteristics, it is gaining popularity including some new areas such as embedded systems, Internet appliances and so on. These applications interact with the real-world by controlling or handling inputs from devices. They typically require real-time constraints to some degree. Some many need guarantees to avoid total system failure, and some can negotiate to balance its load. Hence we need some mechanism in Java to ensure those requirements and the flexibility to dynamically adjust to a feasible state. In current Java environment, however, there are many unpredictable behavior in the system such as unbounded priority inversion in the system, dynamic loading and compilation (for virtual machines supporting just in time compilation) of objects, and activities for garbage collection.

We designed a real-time Java environment and implemented real-time thread as the first step. We have also changed the synchronization mechanism in the virtual machine to provide real-time features. The effectiveness of these functionality in the Java environment has been evaluated.

In this paper, we will first discuss the benefits of using Java and the limitations that arise when implementing a real-time Java system. We then describe our Java environment which is designed to solve those issues. In Section 4, we will explain our implementation. Then we will evaluate the performance and timing accuracy of real-time Java threads, and the effectiveness of real-time synchronization in Section 5 and conclude.

*This research is conducted under the fund of Information-technology Promotion Agency, Japan (IPA).

†Java is a trademark of Sun Microsystems, Inc

2. Pros. and Cons. of Java

2.1. Benefits of Java

Java has many software engineering benefits such as security features in distributed environments, reusability of code, and portability to other platforms.

If we compare portability, C and C++ were portable only in source code level. Once compiled into an executable, it will be linked to libraries which are machine or host OS dependent. To port programs to new platforms, programmers must recompile programs to each target host. If the programmer wishes to send a program to a remote target, programmer must know the architecture beforehand. It was impossible to write a program for a target not known in advance. Java on the other hand, defines a platform independent instruction set that can be executed on any Java virtual machine. Thus Java programs can be portable in compiled form.

Java also supports multi-threading in the language. C and C++ programmers did not have a standard threading mechanism, thus often had to rewrite their programs for each target platform and threading interface. Many engineers for embedded systems needs multi-threading because their system is often required to handle random and periodic external events coming from various devices. To handle those events, the system must interrupt whatever it was doing to handle those events. Threads are used to establish this efficiently. Support for threads in the language can make the programs more portable and readable.

2.2. Limitations in Current Java

To support the real-time requirements from new application domains, current Java execution environment and language specification is not sufficient. We have categorized some of the problems and system elements that needs to be refined.

- Thread

  Current Java thread cannot fully express many of the new requirements needed by the application. For example, Java thread has a method called **`sleep(t)`** where a running thread will suspend at least `t` milliseconds. This method alone is insufficient for time critical requirements because it does not give any guarantee about the time at which it will actually resume. Thus programmers cannot rely on this method for time critical activity.

- Synchronization

  Java deals with synchronization among different threads of control with the keyword **`synchronized`**. Every object in the virtual machine has a lock associated with it. By calling a method or an arbitrary block of code synchronized, the virtual machine acquires the lock associated with the specified object and ensure that no other thread is executing the same code at the same time. If no care is taken, unbonded priority inversion will occur. Then it becomes impossible to guarantee timing requirements, because a job may be delayed by an arbitrary amount of time. This cannot be easily detected and solved by Java programmers. Thus the problem should be solved within the virtual machine.

- Resource Management

  One of the subtle issue is how we handle resources for Java to support real-time activities. The ability to provide resource abstraction is often needed. If real-time applications are to co-exist with other activities, reservation of resources[12] should be done to ensure it so it can continue to run.

  Unfortunately in current Java language, it is difficult to express resources or to reserve resources such as CPU time, memory, network, and I/O bandwidth. Without the notion of resources, it is difficult for programmers to maintain a certain level of quality of service (QOS) when a system is in an overloaded condition, or reserve resources to make guaranteements for real-time activities.

- Window Management

  Various media types such as text, audio, and video will be handled in Java. To support the handling of continuous media types in real-time system, timing requirements must be guaranteed throughout the system including window management. We need some mechanism to ensure timing requirements in the window system and it also should ensure prioritized work so the program will be served in the same priority by the window server too.

  If we take X11 which is used in many UNIX systems, the action of moving one window can dominate or halt all other activities. For example, if we were monitoring real-time data from a radar, the data will not be updated until the user finishes the movement of the window.

- Execution Environment

  In the execution environment for Java, there are activities undesirable for real-time. For example, if the virtual machine supports JIT (Just In Time) compilation, the execution time of a method will be very long if it is being JIT compiled. Although a previously JIT compiled code has predictable execution time in an controlled situation, it is extremely difficult to predict when a JIT compilation activity occurs. For example, a code which need to be JIT compiled may be executed on a event notification that occurs randomly. Care must be taken to prevent this effect. An approach would be to precompile the classes needed for real-time activity prior to execution and save it in a file. Then use that class file for actual execution.

Another source of unpredictability is the garbage collector. In current implementation of many virtual machines, when the garbage collector runs, a thread may be blocked for an arbitrary amount of time. This prevents programmers to analize worst case execution time. As the requirements for the garbage collector, it is desirable to satisfy the resource needs of the real-time activity while still making guarantees over the timing requirements.

3. Design of Real-time Java Environment

To investigate issues in Java as we have discussed previously, we are developing a prototype real-time Java environment. In this section we describe our design of real-time Java environment.

Our real-time Java environment is implemented as an extension package to the Java virtual machine. The intention is to keep changes small as possible to the original virtual machine and the language itself. We did not wish to customize the byte code interpreter or the compiler and reimplement a real-time system which is Java-like.

We have divided our work into three phases, Roast, Grind, and Brew.

3.1. Roast Phase

In this phase we support applications which requires the ability of expressing explicit timing requirements. To establish the requirements, the following extensions are provided.

- real-time Java thread
- real-time synchronization in the virtual machine
- timing fault handler mechanism

Soft real-time applications such as data acquisition and multimedia systems can benefit from this extension. The goal of this phase is to introduce real-time programming APIs and evaluate its usefulness.

3.2. Grind Phase

Grind phase will tune the performance of the system and extend it to support various soft real-time requirements throughout the system. The following functions will be provided.

- user-level real-time thread and synchronization mechanism
- network support
- real-time window support

To implement user-level thread and synchronization mechanism, we will extend the RTC-thread package[15, 16] and add real-time synchronization functions to satisfy the requirements of real-time Java threads. To extend real-time support throughout the system, real-time window management, and prioritized network support will be provided. As a real-time window server, we are currently considering ARTIFACT[17] which was developed at Carnegie Mellon University. ARTIFACT has the capability of prioritizing client requests and managing preemption points to avoid unbounded priority inversion in the window server. We are planning to provide network support using a user-level socket library[9]. Extensions and evaluations are currently being made to provide real-time support[7].

3.3. Brew Phase

In the Brew phase, we will make reliable guarantees to resources such as CPU time, memory, network, and I/O bandwidth available to the programs. To provide this feature, resource management functionality such as garbage collectors will be redesigned and support for online worst case execution time analysis and admission control mechanism should be added. Programmers will be able to express resources and reserve them using new classes supported in the extension. Our goal in the Brew phase is to support hard real-time application as well as soft real-time.

4. Implementation

We have finished implementing the Roast phase and we are currently implementing the grind phase and designing the mechanisms of brew phase. In this section we explain the implementation of Roast phase.

4.1. System Architecture

Our real-time Java environment in Roast phase is implemented as a user-level server on RT-Mach[22]

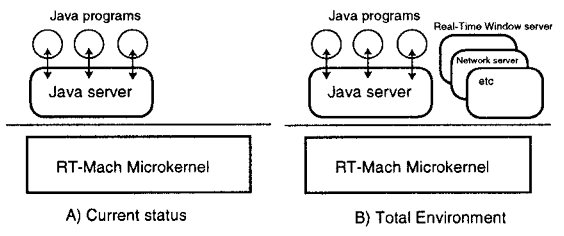

Figure 1. System Architecture

as shown in Figure 1. RT-Mach is an extension of Mach 3.0 microkernel developed at Carnegie Mellon University. It provides distributed real-time computing environment for a wide range of target machines including Pentium, SPARC, MIPS, and Power PC architectures. RT-Mach provides real-time features such as real-time threads, real-time scheduler, real-time synchronization[21], real-time IPC primitives[8], and processor capacity reservation[12]

Java server is based on RTS (Real-Time Server) [13] which is also a user-level server on RT-Mach. RTS is a simple object-based server which provides task management, file management, name service, and exception handling. Java server extends RTS and enables the server to interpret Java byte code as well as native binaries. It has an in-memory file system and different types of file system can be mounted from various media such as floppy, hard disk or RAM disk. Files can be copied to in-memory file system as continuous memory blocks. In our current implementation, Java server mounts Unix file system and copy necessary files (classes) from the local disks into its in-memory file system at initialization. This avoids blocking for disk I/O while executing Java methods. But devices with less memory can choose to read from a mounted file system rather than to keep it in memory or can request data from another server in the same machine or a remote machine. For the Java byte code interpreter, we used a virtual machine called kaffe[23] which supports interpreting as well as JIT compilation. By default, our server runs in JIT mode.

4.2. Real-Time Java Threads

We implemented real-time Java threads which extends Java threads. Real-time threads have been very effective in preserving timing constraints when dealing with continuous media data and thus resulting in decrease of jitters and noise in multimedia applications. Programmers can easily detect whether a thread missed a deadline or not and change the QOS (quality

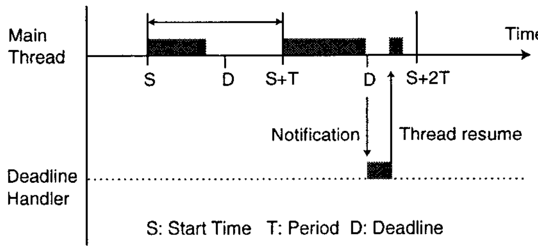

Figure 2. Real-Time Thread model

of service) of continuous media dynamically[20].

RT-Mach has two implementation of real-time threads. One is kernel-level threads called RT-Threads and the other implemented at the user-level, called RTC-Thread. RT-Thread is scheduled by the kernel-level scheduler and timing management is done at the kernel using a clock device which interrupts the kernel at short intervals. RTC-Thread separated timing management and thread management and put both functions at the user-level for flexibility and efficiency. In both thread models, a thread becomes a real-time thread by specifying its timing attributes.

A real-time thread is defined with its attributes `f`, `S`, `T`, and `D`. `f` indicates the thread's function. `S`, `T`, and `D` indicate the thread's start time, period, and deadline respectively. Note that if a thread is periodic then it will automatically restart, or *reincarnate*, when it reaches the end of its function body.

We inherited the model of real-time threads of RT-Mach shown in Figure 2. Main thread will start at time `S` with a period of time `T`. If it misses its deadline `D` and a deadline handler is specified, main thread will be suspended and thread of control will be handed off to the deadline handler which is another thread. Deadline handler can specify a forward or backward recovery action as well as change the main threads attributes such as priorities, deadline, and period or some application specific attributes. For example, if a thread in a movie player application misses a deadline, the deadline handler can choose to reduce the frame rate or resolution of the movie.

We have defined the priorities of real-time and regular Java threads as shown in Figure 3. Non real-time threads have 10 levels of priorities starting from 1 and up to 10. (In Java, higher number means higher priority.) All real-time threads have a higher priority than a non real-time thread. In between non real-time and real-time threads, there are house keeping activities such as garbage collecting threads and final-

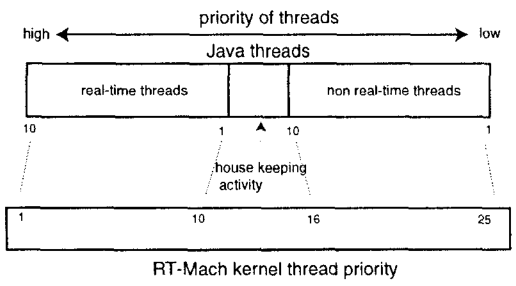

Figure 3. Thread Priorities

izer threads. Internally, priorities of Java threads are mapped to threads provided by RT-Mach.

After a thread object has been created, programmer calls the `start()` method of the object to start the thread. When the `start()` method is called for both Java thread and real-time Java thread, the virtual machine actually creates a kernel thread with the attributes of the Java threads. For non real-time threads, priority information and stack information are passed. For real-time threads, additional timing attributes and information to support deadline handler are passed. Next it starts interpreting from the `run()` method of this thread object. After completion of executing the `run()` method, it exits by interpreting the `exit()` method only if it is not a real-time periodic thread. For periodic threads, it will reincarnate and execute the `run()` method periodically.

By mapping Java thread to RT-Thread, Java threads are scheduled by the real-time scheduler in the kernel. The real-time scheduler in the kernel separates policy and mechanism. By selecting the policy module we can change the scheduling policies. Current version of RT-Mach has round robin, fixed priority, earliest deadline first, rate monotonic, and deadline monotonic scheduling modules.

4.3. Synchronization

To support real-time synchronization policies, we used `rt_mutex_lock` and `rt_mutex_unlock` primitives provided in RT-Mach. By using these primitives, we can specify the synchronization policies to be used such as basic priority inheritance protocols[19] to avoid unbounded priority inversion problem.

Java allows nested locking of object by using the `synchronized` primitive, but the semantics of `rt_mutex_lock` did not allow it. We have made the virtual machine keep track of the owner of the lock and count the times it has been locked. If the owner tries to get the same lock, it just increments the count.

One of the problems we encountered while implementing the mechanism of deadline-handler is that the program may deadlock if a real-time thread misses a deadline inside a synchronized method. The thread of control will go to the deadline-handler. If the deadline-handler wishes to lock the same object as the original main thread, a deadlock occurs because the main thread has not released the lock.

To deal with this problem, we choose to let the main thread execute even if it has missed the deadline until it exits the synchronized block of code. Then let the deadline-handler do its job.

4.4. Application Programming Interface

To use real-time Java threads, we introduced a new class which extends `Thread.class` called `RtThread.class`. Programmers can use the same methods in `Thread.class` as well as new methods provided to support real-time.

To use Java threads, a programmer would construct an object derived from `Thread` class and call `start()` as we show below. By invoking `start()` method, the thread starts by executing the `run()` method of this `MyThread` object.

```
MyThread th = new MyThread( . . . );
th.start();
```

Real-time Java threads are created by instantiating the class `RtThread` as we show below. `P`, `S`, `T`, and `D` indicate the threads priority, start time, period, and deadline respectively. By specifying a deadline hander, method `meth` will be invoked when it has missed its deadline. This can enable programmers to dynamically adjust attributes of its threads to provide better quality of service for continuous media applications.

```
RtThread rtth = new RtThread(P, S, T, D);
rtth.setHandler(RtHandler handler, meth);
```

These are some of the new methods added in RtThread.class.

- `RtThread.setAttr(int priority, Time start, Time period, Time deadline)`
  This method sets the priority and timing attributes of the real-time thread.
- `Time RtThread.getAttrStart()`
  This method gets the start time of the real-time thread.
- `Time RtThread.getAttrPeriod()`
  This method gets the period of the real-time thread.
- `Time RtThread.getAttrDeadline()`
  This method gets the deadline of the real-time thread.

- `void RtThread.setHandler(RtHandler handler, String meth)`
  This method specifies the deadline handler and its method. When the thread misses a deadline, method called `meth` is invoked by the handler thread.

A class added to support `RtThread.class` is `RtHandler.class` and `Time.class`. `Time.class` is a class to express seconds and nanoseconds. To specify the start time, period, and deadline, the programmer will use this class.

`RtHandler.class` is implemented using real-time Java thread by extending `RtThread.class`. It detects the deadline misses of real-time threads and take appropriate actions specified by the programmer. A single handler can manage deadline misses of multiple real-time threads.

Using these methods, programmers can express periodic threads and aperiodic threads. Periodic threads can be expressed using priority, start time, period, and deadlines. A new instantiation of the periodic thread will be scheduled at the start time specified. Aperiodic threads can be expressed by specifying priority, start time, and deadlines.

Next, we illustrate an example Java program creating a periodic thread with a deadline handler.

```
public class myProgram {
  public static void main(String args[]) {

     MyHandler handler = new MyHandler();
      // set handlers priority
     handler.setPriority(10);
      // start deadline handler
     handler.start();
      // timing attributes
      // period is 1 sec
     Time period = new Time(1, 0);
      // deadline is 1 sec
     Time deadline = new Time(1, 0);
      // start immediately
     Time start = new Time(0, 0);

     MyRtThread rtThread = new myRtThread();
      // set thread attribute
     rtThread.setAttr(1, start, period, deadline);
      // set deadline handler
     rtThread.setHandler(handler, "Handler");

      // start RT-thread
     rtThread.start();
      // wait a while
     try {
             Thread.sleep(60000);
      } catch (catch (Exception e) {
     }
      // stop RT-thread
      rtThread.stop();
      // stop deadline handler
      handler.stop();
   }
}

class MyHandler extends RtHandler {
   public void  Handler(RtThread Th) {
        // deadline miss is detected

        // describe recovery job or fault
   }
}

class MyRtThread extends RtThread {
   public void  run() {
          // Thread body

          // Do some Job
   }
}
```

operations	RT-Java	kernel thread	kaffe
lock object	33.4μs	17.7μs	5.9μs
new Thread	851.7μs	N/A	101.9μs
set priority	1.8(16.5†)μs	15.5μs	1.8μs
get priority	1.0μs	20.5μs	1.0μs
start Thread	299.7μs	146.9μs	140.3μs
context switch	20.5μs	9.7μs	5.5μs

†set priority to a running thread

Table 1. Basic Performance of Java Threads

5. Evaluation

We have measured all performance on an Intel Pentium 166 MHz machine with 32MB of memory. To measure the time elapsed to execute a block of code, we used RDTSC (read time stamp counter) instruction[10] on the Pentium processor.

5.1. Basic Performance

We have evaluated the basic performance of Java server compared with RT-Mach's kernel threads and the original virtual machine in Table 1. Performance of Java server was evaluated with itself as the only server on RT-Mach. Since the original virtual machine kaffe was an application program running on Unix, we evaluated its performance on 4.4 BSD Lite Server on RT-Mach.

In kaffe, the cost of locking an object is approximately 30 μsec faster than our real-time Java server. This cost difference comes from the locking mechanism of the virtual machine. Our server uses kernel-level locks that supports real-time synchronization which costs 17.7 μsec. On the other hand, the locking mechanism and thread support are implemented at the user-level in kaffe. Creating a new thread object in real-time Java server takes 851.7 μsec which is slower than kaffe.

operations	
new RtThread object	877.1μs
start aperiodic thread	440.3μs
start periodic thread	519.1μs
start periodic thread with deadline	586.2μs
delay from absolute start time	81.8μs

Table 2. Performance of Real-Time Java Threads

To create a new thread object, the virtual machine will allocate the locks for various objects. Then it needs to lock and unlock various shared data such as garbage collection information. Hence locking performance difference accumulates to the difference shown here. In our current virtual machine, an object was locked 11 times to create a new thread object.

To set priorities in real-time Java server, it takes 1.8 μsec for a thread which has not yet been started. This cost is equal to that of kaffe because setting a priority is just changing the value at the user-level. To set the priority of a running thread, it costs 16.5 μsec. This is because the virtual machine must propagate the priority to the kernel which costs 15.5 μsec. To get priority information, both virtual machines will just read the value in the thread object. Thus it will only take 1.0 μsec.

It takes 299.7 μsec to start a thread in real-time Java server compared to 140.3 μsec of the original virtual machine. This is because our real-time Java thread creates a native thread in the kernel when it starts, while kaffe creates a user-level thread. To create a native thread, it takes 146.9 μsec.

In Table 2, we have measured the performance of real-time Java thread related actions. Starting of an aperiodic thread costs 440 μsec. Starting of a periodic thread costs 519 μsec. The difference is from the setting of timing attributes and the cost of creating periodic timer in the kernel. Starting of a periodic thread with a deadline handler specified costs 586 μsec. This comes from the overhead of setting the deadline notification mechanism. Delay from absolute start time is the time for the thread to actually execute the first Java instruction from the clock tick of the specified absolute time.

5.2. Periodic Activity

In Figure 4, we have compared the timing accuracy of Java threads and our real-time Java threads with interfering threads running at the same priority. For regular Java threads, to emulate periodic activity we

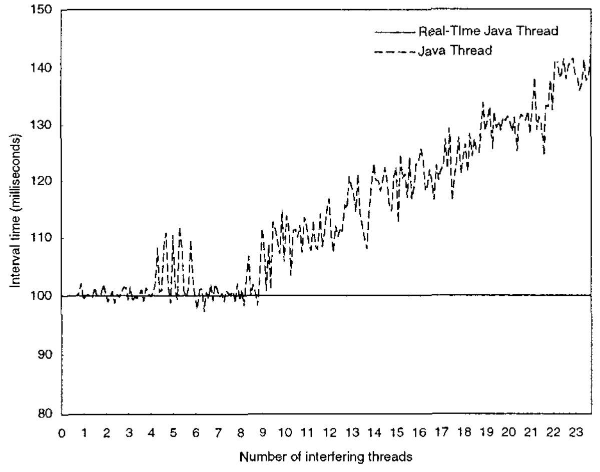

Figure 4. Timing Accuracy with Interfering Threads

created a thread using `sleep` method. Inside a `while` loop the thread will sleep for 100 milliseconds. The interval time of the thread re-entering the head of the loop was measured. For real-time Java threads, we created a periodic thread with a period of 100 milliseconds and measured its period. In both cases, the threads do not consume any workload. We added a number of interfering thread which computes for a random amount of time (with no I/O). All of these threads have the same priority.

We can observe from Figure 4 that as interfering threads increases, regular Java thread behaves more unpredictable. Scheduling delay occurs caused from competing threads. When there are 20 other threads, the delay will increase up to more than 20 milliseconds. On the other hand, our real-time Java thread will keep its period even when there are 20 other competing threads. This shows real-time Java threads are more reliable than regular Java threads with overhead in the system.

5.3. Effect of Synchronization Policy

We have created a benchmark program to evaluate the synchronization cost with different policies. This benchmark is similar to the Distribute Hartstone(DHS)[11] benchmark test. Since DHS is designed to evaluate the communication performance in a distributed environment, it is necessary to create clients and servers running on different machines. To test the effect of various synchronization policy, we created a synchronized method instead of running a server.

Table 3 shows the task set of the benchmark. All threads are periodic threads, and their priorities are assigned rate-monotonic basis. The benchmark uses the kilo-Whetstone as a unit of computation derived from the popular Whetstone benchmark. Each thread enters the synchronized method and computes for a variable amount of time. Then it will exit the synchronized method and consume its specified local workload. The computation time in the synchronization method is variable, and it is increased until any of the thread misses its deadline. The computation time of the local workload is static. Figure 5 shows the timing requirements of each threads. We have calculated the CPU utilization when any one of the thread missed its deadline.

Thread	Local Workload	Period	Priority
Thread1	100 KWS	80 ms	5
Thread2	100 KWS	160 ms	4
Thread3	200 KWS	320 ms	3
Thread4	200 KWS	640 ms	2
Thread5	800 KWS	1280 ms	1

Table 3. Benchmark Task Set

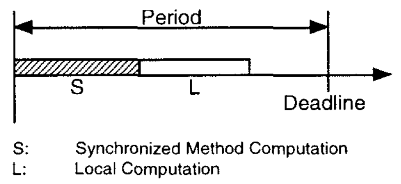

Figure 5. Thread timing requirement

We executed the benchmark with various synchronization policies. Figure 6 shows the result of the benchmark. The result indicates that the CPU utilization using priority inheritance protocol is about 1.6 to 2 times higher than the other two policies. It is demonstrated from the result, that priority inversion is avoided using priority inheritance protocol.

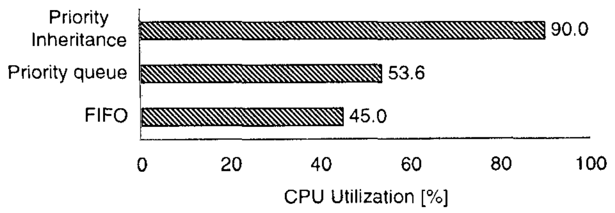

Figure 6. Synchronization Effect

5.4. Self-Stabilization

The self-stabilization scheme was proposed to control a playable QOS-level by the player itself without running any QOS-manager. In this section, we demonstrate the real-time Java version of self-stabilization scheme.

Each thread calculates a suitable period and deadline every iteration and apply it to the next iteration. The period and deadline are the same and obtained as follows:

$$P_{next} = G_n K_n + G_{n-1} K_{n-1} + \cdots + G_1 K_1$$

$$\sum_{i=1}^{n} Ki = 1, \quad 0 \le K_i \le 1$$

where P_{next} is the next period and deadline, and G_n represents ith inter-frame gap.

To simplify the algorithm, we use the following.

$$P_{next} = G_{last} K_1 + G_{last-1} K_2 + G_{last-2} K_3 + G_{last-3} K_4$$

$$K_i = \{\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8}\}$$

To raise the period rapidly when there are enough CPU resources, the latest inter-frame gap multiplied with 0.9 is applied for G_{last} if the thread does not miss the last deadline. To detect deadline misses of threads, we used `RtHandler.class`. The threads are given an equivalent amount of work to playing a movie, since we are still working on the window system.

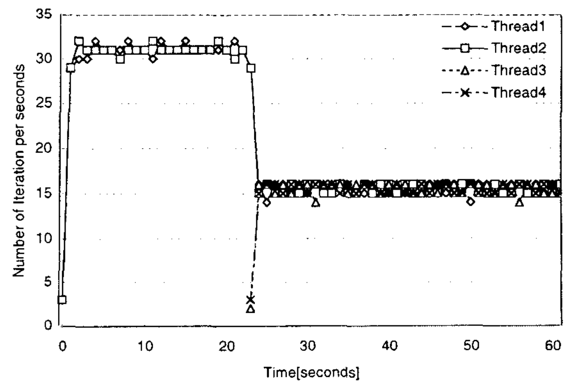

Figure 7. Inter-iteration Gap using Self-Stabilization

Figure 7 shows the number of iterations of each threads. First, two threads start at the same time and tries to get a stable QOS-level. In this case, 30 iterations per second is the stable QOS-level for two threads. 23 seconds after the first two threads start, two more threads start. All four threads are maintained at around 15 iterations per seconds. The result indicates that real-time Java threads with self-stabilization mechanism can maintain a stable QOS-level.

6. Related Work

As an example of real-time thread implementation, there is Spring's real-time thread package[4]. It focuses on dynamic hard real-time environments. In their model, programmers are able to specify attributes such as whether a thread will create another thread or not, and various timing requirements. The scheduler will analize if those threads are schedulable. Spring's real-time thread is scheduled in a reservation-based scheme, where the scheduler will schedule activities only if the necessary resources can be obtained. To handle events, the thread which creates the event-handling thread will negotiate with the scheduler to see if it is schedulable. If it is unschedulable, the thread may request to create the same thread with a longer deadline or negotiate to create a different event-handling thread that requires less resource. While this approach is suitable for hard real-time systems, it may lower the responsiveness and utilization of the system for event oriented soft real-time applications often seen in multimedia domains. There are other real-time thread implementation such as Schwan's[18] which is implemented on a multiprocessor system and based on C thread[1] interface. In both packages threads are scheduled in a deadline-based manner. By using the mechanism provided in real-time thread packages mentioned above, real-time Java environment can be constructed. Although, there remains an issue how to let non real-time Java threads which is scheduled in a priority based manner co-exist with the real-time ones, and how to define the interfaces for real-time extensions.

As an effort to extend general-purpose language for real-time, there are works done at the ART project at Carnegie Mellon University called RTC++[5]. They have merged real-time threads with C++ constructs. For example, they have proposed a construct `within (t) do s except q`, where a programmer specifies a job `s` to be executed within time `t`. If the timing requirements are not met, exception `q` will be thrown. It is implemented using kernel threads. There are other works such as Real-Time Concurrent C[2] and FLEX[6]. Both extend the C language to satisfy predictability requirements. As an extension to Java for real-time, there is work by Nilsen[14]. In his work, he has proposed a new API for real-time applications. He has introduced statements such as `timed` and `atomic`. The `timed` statement can put an upper bound on the execution time for a block of code, and an `atomic` statement can disable preemption for a block of code. To execute their code, a special compiler or a preprocessor is necessary.

Our extension of Java is based on thread primitives. Using our primitives, the extensions proposed to general language for real-time can be implemented. Note that using this approach, one must use a customized compiler or a precompiler like Nilsen's work. For example, the `timed` statement used in Nilsen's extension can be implemented using our real-time Java threads. `within (t) do s except q` construct used in RTC++ can be implemented using real-time Java threads and deadline handler mechanism. Our extensions are flexible enough to construct higher level of abstraction. At the same time, it is a natural extension since Java programmers are already familiar with thread programming. It can also be used without a customized compiler or precompiler.

7. Conclusion

We discussed the benefits of using Java in new application domains. In those domains, real-time support is essential which is lacking in current Java language and execution environment.

Our goal is to provide a real-time Java execution environment. We have designed a system to support real-time requirements. Our work is divided into three phases (Roast, Grind, Brew). In each phase, components such as real-time Java threads, real-time synchronization support, real-time window, network, and resource management functionality will be implemented. Currently we have finished implementing the first phase and real-time Java threads and real-time synchronization support are implemented.

We explained our real-time Java virtual machine implementation and measured the performance of real-time Java threads and the effect of real-time synchronization. Compared with the original virtual machine's basic performance, synchronization and thread creation costs were higher. This is because we use kernel-level threads and real-time locks, where the original virtual machine uses user-level threads and locks. This lead us to future work to provide real-time thread and synchronization at user-level. From the performance comparison of kernel-level threads and user-level threads, we are expecting performance improvements of 2 to 5 times.

From the evaluation results, we have confirmed that using our real-time Java threads, timing requirements under loaded condition are satisfied and unbounded priority inversions can be avoided using basic priority inheritance protocol. We implemented a self-stabilizing application using real-time Java threads as an example to show the usefulness of real-time Java threads and deadline handlers.

We have just finished the Roast phase of our implementation towards complete real-time Java environment. Work is undergoing for the rest of the two phases, Grind and Brew. Network support will be provided through user-level socket library. GUI will be provided by using a real-time window server ARTIFACT. Currently, we are evaluating and extending the real-time functions of these components. The garbage collector is being redesigned for real-time. From our experiences, we are confident that our technology can be applied to support growing real-time application domains of Java.

Acknowledgement

We would like to thank all the members of MKng project for their valuable comments.

References

[1] E. C. Cooper and R. P. Draves. C threads. Technical Report CMU-CS-88-154, Computer Science Department, Carnegie Mellon University, Mar. 1987.

[2] N. Gehani and K. Ramamritham. Real-Time Concurrent C: A language for Programming Dynamic Real-Time Systems. *Real-Time Systems Journal*, 3(4), 1991.

[3] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.

[4] M. Humphrey, G. Wallace, and J. Stankovic. Kernel-Level Threads for Dynamic, Hard Real-Time Environments. In *Proceedings of the 16th IEEE Real-time Systems Symposium*, Dec. 1995.

[5] Y. Ishikawa, H. Tokuda, and C. Mercer. An Object-Oriented Real-Time Programming Language. *IEEE Computer*, 25(10), 1992.

[6] K. B. Kenny and K.-J. Lin. Building Flexible Real-Time Systems Using the Flex Language. *IEEE Computer*, 24(5), 1991.

[7] T. Kitayama, A. Miyoshi, T.Saito, and H. Tokuda. Real-Time Communication in Distributed Environment -Real-Time Packet Filter Approach-. In *Proceedings of the 4th International Workshop of Real-Time Computing Systems and Applications*, Oct. 1997.

[8] T. Kitayama, T. Nakajima, and H. Tokuda. RT-IPC: An IPC extension for Real-Time Mach. In *Proceedings of the USENIX Symposium on Microkernel and Other Kernel Architecture*, Sept. 1993.

[9] C. Maeda and B. Bershad. Protocol Service Decomposition for High-Performance Networking. In *Proceedings of the 14th Symposium on Operating Systems Principles*, 1993.

[10] T. Mathisen. Pentium Secrets, 1997. Byte magazine http://www.byte.com/art/9407/sec12/art3.htm.

[11] C. Mercer, Y. Ishikawa, and H. Tokuda. Distributed Hartstone: A Distributed Real-time Benchmark Suite. In *Proceedings of the 10th Internatinal Conference on Distributed Computing Systems*, May 1990.

[12] C. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[13] T. Nakajima, T. Kitayama, and H. Tokuda. Experiments with Real-Time Servers in Real-Time Mach. In *Proceedings of USENIX 3rd Mach Symposium*, 1993.

[14] K. Nilsen. Java for Real-Time. *Real-Time Systems Journal*, 11(2), 1996.

[15] S. Oikawa and H. Tokuda. User-Level Real-Time Threads. In *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software*, May 1994.

[16] S. Oikawa and H. Tokuda. Efficient Timing Management for User-Level Real-Time Threads. In *Proceedings of the 1995 IEEE Real-Time Technology and Applications Symposium*, May 1995.

[17] J. Sasinowski and J. Strosnider. ARTIFACT: A Platform for Evaluating Real-Time Window System Designs. In *Proceedings of the 16th IEEE Real-time Systems Symposium*, Dec. 1995.

[18] K. Schwan, H. Zhou, and A. Gheith. Multiprocessor Real-Time Threads. *Operating Systems Review*, 26(1):54–65, Jan. 1992.

[19] L. Sha, R. Rajkumar, and J. P.Lehoczky. *Priority inheritance protocols: An approach to real-time synchronization*. 1987.

[20] H. Tokuda and T. Kitayama. Dynamic QOS Control based on Real-Time Threads. In *Proceedings of the 4th Int. Workshop on Network and Operating Systems Support for Digital Audio and Video*, 1993.

[21] H. Tokuda and T. Nakajima. Evaluation of Real-Time Synchronization in Real-Time Mach. In *Proceedings of USENIX 2nd Mach Symposium*, Oct. 1991.

[22] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Towards a Predictable Real-Time System. In *Proceedings of USENIX Mach Workshop*, Oct. 1990.

[23] T. J. Wilkinson and Associates. Kaffe A free virtual machine to run Java code, 1997. http://www.kaffe.org.