A Framework Accommodating Categorized Multiprocessor Real-time Scheduling in the RTSJ

Jinsan Kwon
Dept. of Computer and Information Science
Korea University
South Korea
mrkwon@korea.ac.kr

Hyeonjoong Cho
Dept. of Computer and Information Science
Korea University
South Korea
raycho@korea.ac.kr

Binoy Ravindran
Dept. of Electrical and Computer Engineering
Virginia Tech
USA
binoy@vt.edu

ABSTRACT
In this paper, we present a framework for various multiprocessor scheduling algorithms by minimal modification of current Real-Time Specification for Java (RTSJ) [6]. Although the current version of RTSJ provides a secure platform and rich functionalities for real-time Java applications, it lacks multiprocessor support mechanisms, e.g., absence of functions to support processor affinity, to efficiently utilize multiple processing resources. For this reason, we establish a multiprocessor-aware scheduling framework by using system calls of operating systems to make use of processor affinity, FIFO scheduler, scheduling parameter settings, and precision sleep timer functions. In addition to the framework, we also take categorization taxonomy introduced by Carpenter et al. in [1], which generalizes multiprocessor scheduling algorithms on two criteria of migration degrees and priority change complexity. Then our experimental evaluation on the framework with each scheduler class in the categorization taxonomy shows the framework's runtime overhead, which proves the feasibility of our implementation.

Categories and Subject Descriptors
D.4.1 [**Operating Systems**]: Process Management – *Scheduling*; D.4.7 [**Operating Systems**]: Process Management – *Real-time systems and embedded systems*

General Terms
Algorithms, Design

Keywords
Real-time systems, The Real-Time Specification for Java, multiprocessors, scheduling framework.

1. INTRODUCTION
The superiority of multiprocessor architecture in terms of low power consumption and high scalability has made architects to adopt it for real-time systems widely. Unfortunately, comparing to single processor systems, scheduling real-time applications on multiprocessor platforms accompanies complex issues on utilization of multiple computational resources. In absence of a dominant solution for this problem, a large number of real-time scheduling algorithms designed for multiprocessor architectures have been introduced and it has also increased the need of efficiently accommodating such multiple algorithms on a system. This trend leads some operating systems to having functionalities and extensibility, i.e., modular scheduler structure which treats schedulers as extension modules, to host new scheduling policies available.

In single processor architectures, the Real-Time Specification for Java [6] is one good example having such functionalities in which several scheduling algorithms are implemented together with real-time applications. It provides well-defined environment for running real-time applications including high-precision time representation facilities with timers and supports for real-time scheduling parameters for multiple scheduling algorithms. Plus, as Java is based on very powerful concept of isolation from underlying architectures, real-time applications built on the RTSJ and its corresponding Java virtual machine give much easier and simpler way of implementation and deployment than traditional OS-based methods which inevitably depends on operating system and hardware architecture.

With the aforementioned features, the current version of RTSJ framework clearly shows its well-established foundations for further extension, however, lacks of features as a general scheduling framework and elements for multiprocessor environment are also observed. For this reason, it is needed to discuss about 1) the general way for a scheduling framework to relay its scheduling decisions to underlying platform and 2) the functionalities that have to be offered to scheduling algorithms to work with multiprocessing model along with the current RTSJ.

To extend the RTSJ as a real-time application platform with supports of multiprocessor scheduling algorithms, Wellings in [2] considered five models, i.e., dispatching, allocation, cost enforcement, affinity of interrupts, and failure model, along with a suggestion of APIs to support the models. This suggestion was soon included in JSR 282 [10] and implemented in alpha version of RTSJ 1.1 [11].

While the multiprocessor scheduling algorithms for real-time applications vary one another, Carpenter et al. interestingly categorized those algorithms in [1] using following criteria. The category itself is a two-dimensional space defined by the complexity of priority changes and the migration levels. There are

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
JTRES 2012, October 24-26, 2012, Copenhagen, Denmark.
Copyright 2012 ACM 978-1-4503-1688-0 …$15.00.

three different levels for each criterion, which make total 9 different categories present in the taxonomy. The detailed criteria are:

- The complexity of the priority scheme

With this criterion, algorithms are divided into three groups by how often they change a task's priority. In static priority, a task has unique given priority value, and all jobs created by this task have the same values. Job-level dynamic priority class has a static priority within a job, but the job's priority may differ from another job within the same task. Unrestricted dynamic priority class does not specify any restrictions on priority changing behavior. Well-known scheduling algorithms for these classes are Rate Monotonic (RM) [7], Earliest Deadline First (EDF) [7], and Least Laxity First (LLF) [8], respectively.

- The degree of migration allowance

Similar to priority classification, migration level-based classification differentiates an algorithm by how often a job is allowed to migrate between processors. No migration scheme disallows a task from migration at all, and the task must be associated with a specific processor. Restricted migration category policies force only a job released from a task set to execute on one processor. Full migration class does not put any restrictions on migration policies, therefore even jobs currently running can migrate to other processors.

Therefore, the Categorized Multiprocessor Real-time scheduling-supporting Framework (CMRF) that we present in this paper states about both aforementioned issues by using scheduling mechanisms of `PriorityScheduler` in the RTSJ as similar to the priority band model in the Flexible Middleware Scheduling Framework (FMSF) introduced in [3] and the categorization taxonomy introduced by Carpenter et al. in [1]. With the CMRF, we attempt to construct a framework working on Java virtual machines (JVM) accommodating Carpenter's nine categories of real-time scheduling algorithms on multiprocessor architectures. This is done by considering the extensions of the RTSJ suggested by Wellings in [2]. Among the five models addressed in the suggestion, we adopt dispatching and allocation model to the framework, which make it possible to handle tasks with several processors.

The rest of this paper is organized as follows. Chapter 2 reviews related works divided into operating systems and middleware approaches. In Section 3, we introduce the core functions of the CMRF, accommodating multiple scheduling policies on the RTSJ based system with respect to the categorization of the policies. The scheduling overhead and overall performance of result system, with respect to the scheduling algorithms in each category, is then verified in Section 4. In Section 5, we discuss about the framework with the result found in the Section 4. Section 6 then summarizes our conclusion. The API functions introduced with the CMRF are available in appendix Section 7.

2. RELATED WORKS

Since no certain scheduling algorithm is known for a dominant solution for multiprocessor system, there have been various runtime platforms supporting functions to host and employ multiple algorithms, instead of one, on a system. These can be grouped into two, i.e., operating system frameworks and middleware framework, by level of the runtime environment implemented.

2.1 Operating System Frameworks

Frameworks in this group incorporate itself into existing operating systems, mostly Linux, and provide its functions as a scheduling framework in forms of shared libraries and system calls which are exclusive for the OS which the framework is designed for. The frameworks in this type are implemented by manipulating existing OS kernel, or using exported kernel functions which allow other kernel-space programs to guide the kernel when scheduling decisions should be made.

Among various OS-level scheduling frameworks, modification of existing kernel, like what LITMUS^{RT} [4] does, is the most common way to implement the framework. Developed by Calandrino et al., LITMUS^{RT} consists of a set of real-time patches and pluggable scheduler framework which makes traditional non-real-time Linux kernels to be suitable for scheduling real-time applications, by allowing in-kernel preemption and other high-precision time related scheduling events. Under the LITMUS^{RT}, user-made schedulers are exist in forms of built-in kernel modules inserted into the kernel at build time, and should be handled by the APIs shipped with LITMUS^{RT} patch. Once the schedulers are embedded into kernel during its building stage, LITMUS^{RT} makes a device node in `/proc` filesystem which is the main port to communicate with the schedulers built in. Note that all major parts in LITMUS^{RT} are embedded into and work in kernel space, the framework can provide much more choices of functionalities, i.e., kernel variables, system calls, other in-kernel functions and methods while making scheduling decisions, in case of that those functions are not supported by neither the API nor the libraries shipped with the LITMUS^{RT} itself, to scheduler plug-ins while other application-level frameworks cannot provide.

Unlike LITMUS^{RT}, which is well fused into the kernel, RESCH [5] exports kernel functions to deliver events to and relay scheduling decisions from user-defined algorithms. This makes RESCH unique among others, although it is still kernel-space framework. RESCH consists of two parts – the RESCH core and API libraries for scheduler plugins and both parts are in forms of external Linux kernel modules. User-implemented scheduler plugins use RESCH APIs to get scheduling events and make decisions. The plugins are compiled to kernel modules, which can be inserted into the kernel later. In prior to use the newly compiled scheduler, the RESCH core, which is another kernel module, should be inserted into Linux active kernel first, then the scheduler modules may be inserted, called, and used through RESCH libraries. Thanks to the Linux kernel's external module handling capability, all these actions the RESCH performs can be done without the huge effort on modification of the active kernel.

2.2 Middleware Frameworks

Besides OS-level scheduling frameworks, there also has been works on implementation of middleware frameworks host multiple scheduling algorithms. With a tremendous effort of JSR-1 team, the RTSJ [6] defines most the major aspects necessary for scheduling real-time applications. For basic scheduling service, RTSJ implementations equip the `PriorityScheduler` by default. Fundamentally, this scheduler maps a task's priority from middleware, which is a Java virtual machine, to operating system level, and the one-to-one priority level mapping between the framework and OS, which the framework is running on, is preserved. Other schedulers, in addition to the `PriorityScheduler`, may be defined and loaded into the

RTSJ framework by extending the `Scheduler` abstract class. Although the current official release of the RTSJ does not support multiprocessor platforms explicitly, the next release of RTSJ 1.1 [11], developed from JSR-282 [10], is currently in alpha stage and that includes processor affinity and pinning features to support such platforms by default.

Meanwhile, Zerzelidis et al. in [3] presents Flexible Middleware Scheduling Framework (FMSF), a scheduling framework based on the RTSJ environment with accommodation of multiple application schedulers concurrently on single processor environments. This is done by dividing priorities into several scheduling bands, and four priority levels per a band are distributed which are high, medium, medium-lock, and low priorities. Entire framework works with the `PriorityScheduler`, and essentially mapped to priority levels supplied by a fixed-priority preemptive scheduler in the underlying real-time operating system.

While most of middleware frameworks currently work with single processor platforms, JEOPARD consortium introduces Java Environment for Parallel Realtime Development [13], a solution based on JSR-282 [11] for development and operation of platform-independent applications mainly for multiprocessor environment. The interesting point with this project is that the JEOPARD involves not only a scheduling framework as the core feature, but also runtime environment itself and supporting tools as its main purpose. This makes the JEOPARD an unusual middleware solution to run on both traditional hardware platforms running with operating systems and specially designed Java optimized processor with virtual machine interfaces implemented in FPGA code.

3. THE CMRF

From this section we describe design of the CMRF in detail. First, we discuss about the requirements of underlying platforms, the task model used for the framework, and then the framework itself with representative scheduling algorithms for each scheduling category.

3.1 System Assumptions and Requirements

Although multiprocessor architectures can be further divided into several types, we use the term, *multiprocessor*s, to primarily represent the symmetric multiprocessing processors. As RTSJ defines the cost enforcement model optional, the methods to estimate the cost of migration and cooperation between processors are not considered with this architecture.

To support scheduling algorithms for multiprocessor platforms and to make the framework more general at the same time, we take the allocation model introduced in [2]. Wellings defined the model with supporting mechanisms, which we organize system functions for processor affinity feature from Linux operating system to support the model. Scheduling decision made by schedulers using the allocation model then forces a task to use a processor in order to run. This process is performed using dispatching model. Although the model introduced in [2] suggests the allocation process to be free from traditional priority-based mechanism, the CMRF currently use the notion of priority bands introduced by Zerzelidis et al. [3], to relay scheduling decisions from scheduling algorithms in the framework to the global scheduler running in operating system context, which is *SCHED_FIFO* scheduling policy. Since we suppose that there is only one middleware-level scheduler among various scheduling algorithm candidates running at a given time, the number of scheduling bands in our framework is one, instead of many as in [3]. This difference also affects the number of priority levels a scheduling algorithm may have, which depends on the number of levels offered by the underlying operating system in our framework.

SCHED_FIFO scheduling policy and related functions:

```
#include <sched.h>
int sched_setscheduler(pid_t pid,
 int policy,
 const struct sched_param *param);
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
void CPU_SET(int cpu, cpu_set_t *set);
void CPU_ZERO(cpu_set_t *set);
```

Processor affinity functions:

```
int sched_setaffinity(pid_t pid,
 size_t cpusetsize, cpu_set_t *mask);
int sched_getaffinity(pid_t pid,
 size_t cpusetsize, cpu_set_t *mask);
```

Scheduling parameter related functions:

```
int sched_setparam(pid_t pid,
 const struct sched_param *param);
```

Thread identification:

```
#include <sys/syscall.h>
int syscall(SYS_gettid);
```

List 1. Required system calls

Adopting these models from both [2] and [3] requires several system calls served by the operating system, and the list 1 shows the calls required. All the system calls are part of either POSIX.1 standard or its related implementations available in the Linux kernel 2.6 or above to schedule Native POSIX Thread Library (NPTL). Since Java uses Native Thread (essentially NPTL when running the JVM on the kernel version 2.6) to create threads running in the middleware context, scheduling parameters for the real-time threads running in the framework context may also be changed through `sched_setparam()` directly. Note that this function also gives an option of changing a thread's priority to schedulers under the framework even without using the `PriorityScheduler`, and when using with the aforementioned dispatching model, the `sched_setparam()` function makes the framework independent from specific Java runtime environment, especially the use of real-time JVM built only for the framework.

3.2 Task Model

As described in [1] and [6], we use a periodic real-time task set τ having n number of tasks, $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ with the CMRF. To follow task model in [6], we have three parameters for each task τ_i instead of two as in [1]. Therefore, each $\tau_i = (Ci, D_i, P_i)$, where C_i of worst-case execution time, P_i of period, and D_i of relative deadline, are assumed with the framework to support both categorization taxonomy and the current RTSJ release. From this part, we use a concept of '`RealtimeThread`' which is equivalent to the task in the RTSJ.

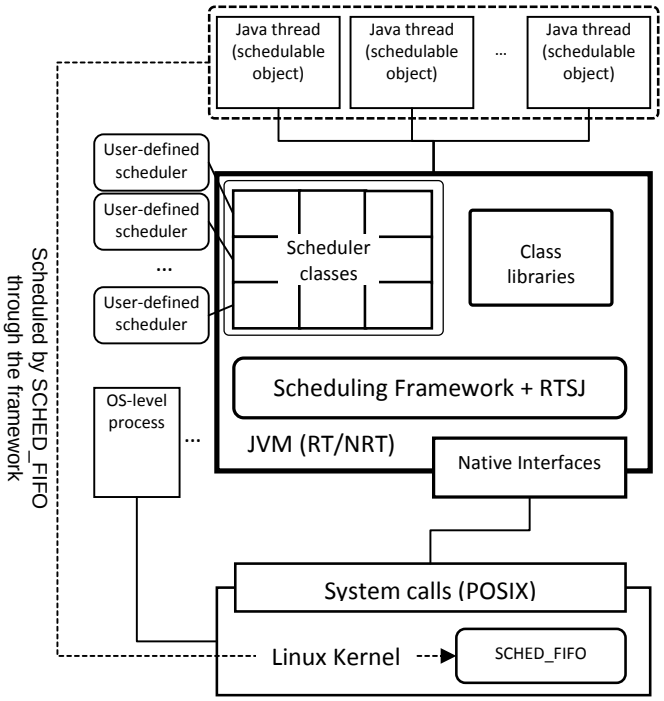

Figure 1. The structure blocks of the CMRF

3.3 Framework

As shown in Figure 1, the CMRF consists of two major parts with supporting libraries, which are native interfaces and RTSJ extension part. Based on the RTSJ 1.0.2, processor affinity features have been added to the `RealtimeThread` class, and the affinity related operations are also available to subclasses of `Scheduler` class. Native interface part works as a helper library for the RTSJ extensions by providing necessary operating system level functions described in the Section 3.1. Note that the system calls identify individual threads by using thread ID (TID), all real-time thread instances created using `RealtimeThread` class on the framework now contains a TID field given by the operating system. The TID is assigned during the first run of a thread, and this is used for the rest of the instance's life time for changing thread priority and processor affinity. Processor affinity with a `RealtimeThread` is represented in a `BitSet`.

3.3.1 Thread Scheduling Flow

In the CMRF, thread scheduling takes place when 1) a new `RealtimeThread` has been created, or 2) a `RealtimeThread` has released a next job, or 3) a job of a `RealtimeThread` has been finished. Whenever those events occur, `reschedule()` method defined in a scheduler, which is registered to the current `RealtimeThread` instance, is called and a thread eligible to run at that time should be chosen within the `reschedule()` method. Then, the chosen thread among the threads in a feasibility set is dispatched to designated processor using `dispatch()` call provided by the CMRF. This routine keeps running on until the entire system terminates.

To make use of the framework, user defined schedulers should be derived from `Scheduler` class in `esrc.cmrf` namespace. Based on `javax.realtime.Scheduler`, the framework's `Scheduler` class provides a static dispatcher and a feasibility

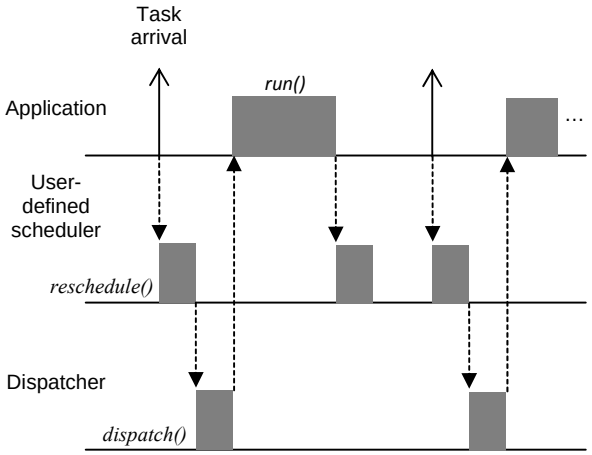

Figure 2. Control flow of the CMRF

set for rescheduling operation. During the rescheduling stage, which is the essential part for a scheduler, one should assign a `RealtimeThread` a proper priority using `PriorityParameters`, because we use *SCHED_FIFO* policy in the OS kernel to schedule the thread essentially. Due to this fact, the priority of the eligible thread should be the highest among the threads running on the same processor, since it is FIFO we are using, but low enough for future preemption at the same time. With the proper priority and a choice of processor among thread's affinity `BitSet`, then, the Dispatcher migrates and signals the thread to continue its execution on the designated processor. Figure 2 shows the control flow of this rescheduling stage.

3.3.2 Native Interfaces

With the CMRF, the system calls introduced in the Section 3.1 are provided as static methods packed in the class `NativeHelper`. The unique feature of the CMRF, which is the capability of running real-time applications without using specially built real-time JVMs, is effective by using follow native functions: `sched_setscheduler()`, `sched_setaffinity()`, `sched_setparam()` and `nanosleep()`. While using NPTL for thread creation running on the framework, initialization of real-time environment with *SCHED_FIFO* policy and scheduling parameters are set through these native interfaces.

3.4 Supported Scheduling Algorithms

In the categorization model introduced in [1], it is possible to reduce the dimension of the category for temporarily by disallowing task migrations. This simplifies scheduling problems as a set of uniprocessor ones, which has already been well-covered by traditional scheduling algorithms, such as RM, EDF [7], and LLF [8]. Then, each scheduling algorithm can be extended to the dimension put backed earlier by adopting different level of migration policies. In this way, Müller et al. in [12] survey various scheduling algorithms sorted under the Carpenter's categorization taxonomy, and find genealogy which shows RM, EDF, and LLF are the progenitor algorithms for all classes. For this reason, we focus the design of the CMRF to support RM, EDF and LLF using a periodic timer which invokes the scheduler at every time quantum. The framework also equips restricted and global migration version of RM and EDF by default.

4. EVALUATION

In this section, we present experimental benchmark results performed on the CMRF to show how much scheduling overhead one should expect with the framework. Test has been done with typical scheduling algorithms of each category, the RM, EDF, and LLF, as found out in [12]. We conducted this evaluation on two different machines and the detailed specifications of each machine are shown in Table 1. Operating system used for the benchmark is Ubuntu Linux 10.04.4 with linux kernel version 2.6.31, PREEMPT_RT patch applied. OpenJDK 6 is used for Java runtime environment.

Table 1. Hardware platform specifications

Processor Model	Intel Xeon E5506	AMD Opteron 6176 SE
Total Cores	8 (4 per processor)	48 (12 per processor)
Clock Speed	2.13 GHz	2.3 GHz
Cache	256KB / 4MB	512KB / 12MB

Table 2. Basic task parameters for a task

Cost (C_i)	≤ 10 ms 0 ns
Deadline (D_i)	100 ms 0 ns
Period (P_i)	100 ms 0 ns
Total released jobs	800,000

To perform the test, we use the parameters described in Table 2 for sample task set containing ten `RealtimeThread` instances. The results in Figure 3 to 8 show the ratio of missing deadlines of jobs while scheduled at each utilization factor maximum of the number of the system's processor. The processor utilization in the figures denotes the overall workload considering all participant processors for entire task set. For example with partitioned scheduling classes, deadline miss ratio at the maximum processor utilization should ideally be close to 0% under dynamic priority scheduling algorithms such as EDF and LLF.

To compare the performance of the framework's basic dispatching facility with other real-time Java virtual machines, we carried out another test which measures time jitters while scheduling two tasks using the parameters in the Table 2. Interestingly, analysis on the 160,000 total jobs for each JVM shows that the CMRF has the lowest arrival time variation in terms of average jitter for periodic tasks among the three JVMs. Java RTS, which is commercial real-time JVM implementing the RTSJ 1.0.2, shows slightly more variation on the arrival times, and the OVM, developed by Armbruster et al. [14], has the most swinging arrival time of about twice of other JVMs. The overall results are shown in Table 3 and 4.

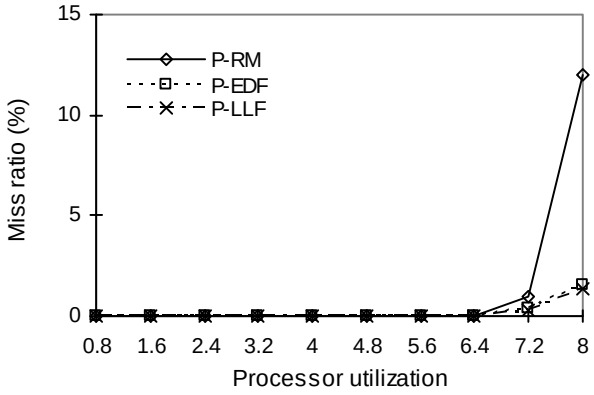

Figure 3. Deadline-miss ratio of partitioned class policies on 8-processor system

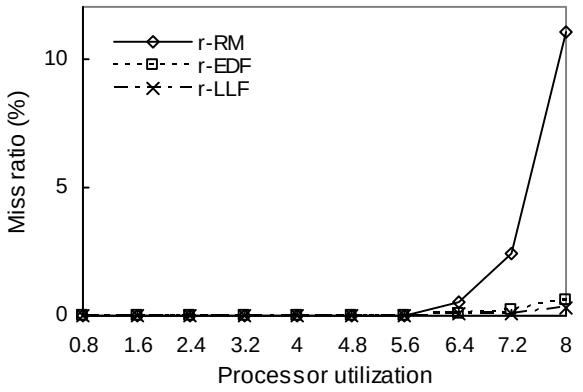

Figure 4. Deadline-miss ratio of restricted migration class polices on 8-processor system

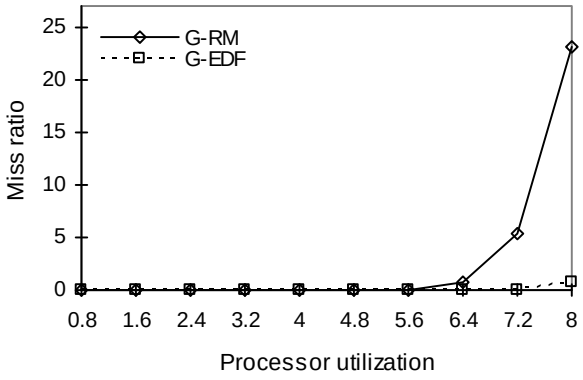

Figure 5. Deadline-miss ratio of global migration class polices on 8-processor system

Table 3. Periodic scheduling jitters on JVMs (Xeon E5506)

(in nanosec.)

	JRTS	OVM	CMRF
Jitter, Min.	2,438	2,529	2,823
Jitter, Max.	144,426	190,048	144,438
Average jitter	53,546	62,201	52,404
Deviation	20,286	23,017	14,053

Table 4. Periodic scheduling jitters on JVMs (Opteron 6176)

(in nanosec.)

	JRTS	OVM	CMRF
Jitter, Min.	1,457	58	1,308
Jitter, Max.	2,295,608	7,672,276	2,220,614
Average jitter	302,836	417,186	293,807
Deviation	180,192	359,516	117,969

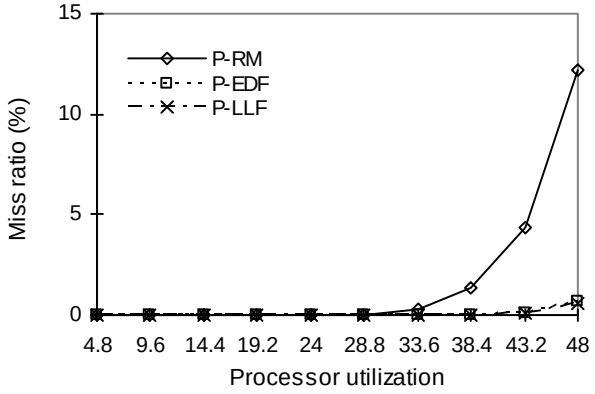

Figure 6. Deadline-miss ratio of partitioned class polices on 48-processor system

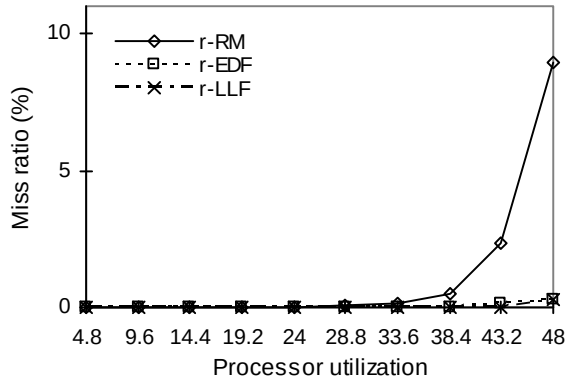

Figure 7. Deadline-miss ratio of restricted migration class polices on 48-processor system

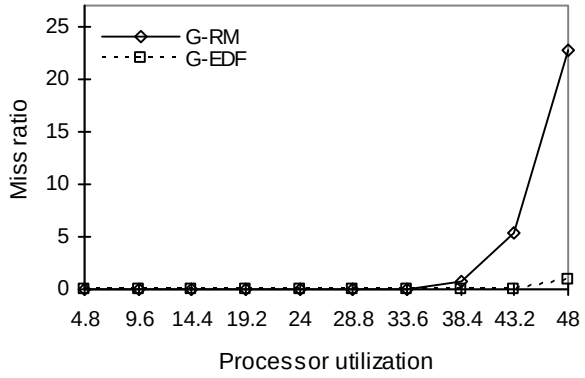

Figure 8. Deadline-miss ratio of restricted migration class polices on 48-processor system

Figure 9 depicts the results of a test on the CMRF measuring absolute time taken to schedule tasks configured same as the jitter test. As the graph shows, the time consumed for scheduling tasks is drastically increases along with the increase of the migration level. Partitioned schemes, which are P-FP, P-RM, P-EDF, and P-LLF, does not involves task migration operations in the `dispatch()` call, therefore, it takes even less time than

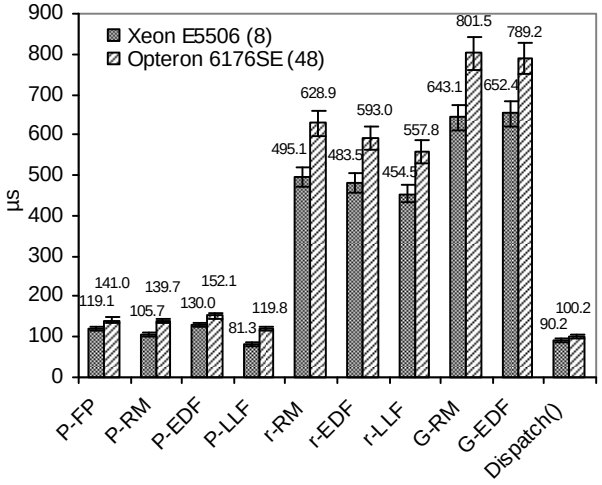

Figure 9. Execution times of schedulers and `dispatch()` call

`dispatch()` call itself which takes about 90 and 100µs to migrate and signal to run a task.

5. DISCUSSION

Although the Categorized Multiprocessor Real-time scheduling-supporting Framework provides the core functions for scheduling algorithms on multiprocessor platforms, it is more desirable to provide more time-accurate interfaces to reduce the scheduling overheads found in the previous section.

One thing that should be addressed with the CMRF is about timers. The timers used in the framework are currently based on the `nanosleep()` function, which is undesirable for events that may occur within less than a millisecond interval because of its accuracy. This issue can be covered by using other OS level timers and POSIX signal functions, however, it may accompany other issues involving operating system level signal handling which may cost a lot for the framework to handle with.

Another aspect that needs to be considered is about job dispatching model. As Wellings mentioned in [2], the current `PriorityScheduler` assumes a single run queue per priority level, which also applies to the operating system's FIFO scheduler. The main feature about the dispatching model is to relieve the dependencies on the concept of priority based ready queue therefore the entire dispatching can be more generalized for execution eligibility rather than traditional priority based processes. Originally, this model influenced the dispatcher design in our framework, however, the priority levels we are using in the scheduling framework have totally no relations from the level defined in the RTSJ, due to the fact that our framework only works on the *SCHED_FIFO* policy in the operating system. Therefore, the priority level in a `RealtimeThread` directly represents the level in the operating system, and this makes the decoupling relations between priority level and the ready queue of the framework more difficult. For this reason, although the dispatcher in the framework partially adopts the dispatching model in [2], priorities of a `RealtimeThread` should be carefully assigned before dispatching a task since this also effects to preemption behavior of FIFO scheduler.

6. CONCLUSION

So far we have examined the possibility of extending the current version of RTSJ to accommodate scheduling algorithms that make use of multiprocessor platforms. Even with traditional Java runtime environment, the framework functions supporting both migration and priority change provide facilities for real-time thread scheduling on the middleware level without using specific real-time JVMs. We have shown the processor affinity and scheduling parameter related system calls support the categorization model, and as a result, provide necessary functions for RM, EDF, LLF scheduling algorithms and their descendants.

7. APPENDIX: FRAMEWORK API

Class `RealtimeThread`:

```
package javax.realtime;
public class RealtimeThread implements
Schedulable {
  private BitSet affinity;
  private int cpu;
  private int tid;
  private AbsoluteTime nextReleaseTime;

  public RealtimeThread
    (SchedulingParameters scheduling,
    …
    BitSet affinity);
  public int getTID();
  public void setTID(int tid);
  public BitSet
    setAffinity(BitSet affinity);
  public BitSet getAffinity()
  public boolean setCurrentCPU(int cpu)
  public int getCurrentCPU()
}
```

Class `NativeHelper`:

```
package javax.realtime;
public class NativeHelper {
  public static native boolean
    sched_setscheduler_FIFO(int pid);
  public static native boolean
    sched_setaffinity(int pid, int cpu);
  public static native int
    setpriority(int tid, int prio);
  public static native int gettid();
  public static int getMinRTPriority();
  public static int getMaxRTPriority();
  //Timers has always the highest priority
  public static int getMaxTimerPriority();
  public static void nanosleep
    (long millis, long nanos);
}
```

Class `Scheduler`:

```
package esrc.cmrf;
public abstract class Scheduler
 extends javax.realtime.Scheduler {
  protected static volatile
    ArrayList<Schedulable> feasibilitySet;
  public abstract void reschedule
    (AbsoluteTime theTime,
    Schedulable schedulable);
  public void addToFeasibilitySet
    (Schedulable schedulable);
  public void removeFromFeasibilitySet
    (Schedulable schedulable);
}
protected static class Scheduler.Dispatcher{
  public static void dispatch
    (RealtimeThread rtt, int cpu);
}
```

8. REFERENCES

[1] Carpenter, J., Funk, S., Holman, P., Srinivasan, A., Anderson, J. and Baruah, S. 2004. A categorization of real-time multiprocessor scheduling problems and algorithms. In J. Y. Leung, editor, *Handbook on Scheduling Algorithms, Methods, and Models,* page 30.130.19. Chapman Hall/CRC, Boca Raton, Florida.

[2] Wellings, A. J. 2008. Multiprocessors and the Real-Time Specification for Java. In *Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing* (ISORC '08). IEEE Computer Society, Washington, DC, USA, 255-261. DOI=10.1109/ISORC.2008.22 http://dx.doi.org/10.1109/ISORC.2008.22.

[3] Zerzelidis, A. and Wellings, A. J. 2010. A framework for flexible scheduling in the RTSJ. *ACM Trans. Embed. Comput. Syst.* 10, 1, Article 3 (August 2010), 44 pages. DOI=10.1145/1814539.1814542 http://doi.acm.org/10.1145/1814539.1814542.

[4] Calandrino, J., Leontyev, H., Block, A., Devi, U. and Anderson, J. 2006. LITMUS^{RT} : A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International,* vol., no., pp.111-126, Dec. 2006 DOI=10.1109/RTSS.2006.27 http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4032341&isnumber=4032321.

[5] Kato, S., Rajkumar, R. and Ishikawa, Y. 2009. *A Loadable Real-Time Scheduler Suite for Multicore Platforms*, Technical Report CMU-ECE-TR09-12, December, 2009.

[6] Gosling, J. and Bollella, G. 2000. *The Real-Time Specification for Java.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[7] Liu, C. L. and Layland, J. W. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* 20, 1 (January 1973), 46-61. DOI=10.1145/321738.321743 http://doi.acm.org/10.1145/321738.321743.

[8] Mok, A. K. 1983. *FUNDAMENTAL DESIGN PROBLEMS of DISTRIBUTED SYSTEMS for the HARD-REAL-TIME ENVIRONMENT.* Technical Report. Massachusetts Institute of Technology, Cambridge, MA, USA.

[9] Walter, A. 2008. *Multicore Support for Realtime Java.* Technical Report. Aicas GmbH, Karlsruhe, Germany.

[10] Dibble, P. and Wellings, A. J. 2009. JSR-282 status report. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems* (JTRES '09). ACM, New York, NY, USA, 179-182.

DOI=10.1145/1620405.1620431 http://doi.acm.org/10.1145/1620405.1620431.

[11] Dibble, P. *JSR282: RTSJ version 1.1*. http://jcp.org/en/jsr/detail?id=282.

[12] Müller, D. and Werner, M. 2011. Genealogy of Hard Real-Time Preemptive Scheduling Algorithms for Identical Multiprocessors. In *Central European Journal of Computer Science*, vol.1, no.3, pp.253-265, September, 2011 DOI=10.2478/s13537-011-0023-z http://dx.doi.org/10.2478/s13537-011-0023-z.

[13] Siebert, F. 2008. JEOPARD: Java environment for parallel real-time development. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems* (JTRES '08). ACM, New York, NY, USA, 87-93. DOI=10.1145/1434790.1434804 http://doi.acm.org/10.1145/1434790.1434804.

[14] Armbruster, A., Baker, J., Cunei, A., Flack, C., Holmes, D., Pizlo, F., Pla, E., Prochazka, M., and Vitek, J. 2007. A real-time Java virtual machine with applications in avionics. *ACM Trans. Embed. Comput. Syst.* 7, 1, Article 5 (December 2007), 49 pages. DOI=10.1145/1324969.1324974 http://doi.acm.org/10.1145/1324969.1324974.