Real-Time Scheduling for Java

by

Catalin A. Francu

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 7, 2002

Copyright Catalin A. Francu, MMII. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and distribute publicly paper and electronic copies of this thesis and to grant others the right to do so.

Author .....
Department of Electrical Engineering and Computer Science
May 24, 2002

Certified by .....
Martin C. Rinard
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Real-Time Scheduling for Java

by

Catalin A. Francu

Submitted to the
Department of Electrical Engineering and Computer Science

May 24, 2002

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

We implement a preemptive, non-idling EDF (earliest deadline first) real-time scheduling algorithm for the FLEX compiler. Real-time threads extend Java threads with timing constraints. The aim of the real-time thread model is to give programmers better control over the order of execution of threads than the priority-based model. The primary advantage is the ability to declare periodic threads with a highly predictable processor demand. The implementation meets the real-time specifications for Java as defined by The Real-Time for Java Expert Group [2]. We use FLEX, the Java bytecode to C compiler [10] as a framework for our implementation.

Thesis Supervisor: Martin C. Rinard
Title: Associate Professor

Acknowledgments

I would like to thank my thesis and academic supervisor Martin C. Rinard for being so patient and supportive. He provided me with valuable suggestions and references to research materials which were vital to the completion of this thesis.

My entire gratefulness goes to William Beebee, who first made me realize that the Master of Engineering thesis is a serious and demanding project, and who taught me to keep my pace in producing this thesis.

I must thank Brian Demsky, C. Scott Ananian, Karen K. Zee, Alexandru Salcianu and all the people involved in the FLEX compiler infrastructure for taking their time to answer all the questions I had during my work on the thesis.

My warmest appreciation goes to Bryan Fink, who gave me the first introduction to the real-time component of FLEX.

Contents

1 Introduction **6**

1.1 Background 6

1.2 Related Work 7

1.3 Goals 8

2 The EDF Algorithm **10**

2.1 Terminology, Assumptions and Notation 10

2.2 Optimality of the EDF Algorithm 13

2.3 Feasibility Analysis Under EDF Scheduling 16

2.4 EDF Pseudocode 17

3 Implementation Key Points **22**

3.1 Program Hooks 22

3.2 Priority and Importance Parameters 23

3.3 Release Parameters 24

3.4 No-Heap Real-Time Threads 25

3.5 Asynchrony 26

3.6 The Programming Interface 27

3.6.1 Rewriting the Scheduler 28

3.6.2 Defining the Real-Time Threads and Parameters 28

3.6.3 Writing the Main Method 30

3.7 Sample Output 31

3.8 Benchmarks 35

3.8.1 The Maximum Processor Load 35
3.8.2 The Maximum Number of Supported Threads 36
3.8.3 The Optimal Time Slice 37
3.9 Conclusions and Future Work 38

Chapter 1

Introduction

1.1 Background

Java is a relatively new language, which learned many lessons from prior object-oriented languages like C++. Its designers focus on the requirements of networked computing. As a result, Java's main qualities, which have made it a preferred language in recent years, are platform independence, trusty garbage collection, a wealth of standard libraries, cleaner code and nice abstraction barriers.

One of the aspects where the Java designers had to cut back to obtain these performances is **real-time reliability**. With most existing Java implementations, consistent real-time behavior is impossible to attain, for several reasons:

1. The programmer has only loose control over the order of execution of different threads, by using priorities, locks and synchronization. Moreover, existing scheduling policies cannot make guarantees with respect to any timing constraints.

2. The threads in `java.lang.Thread` are vaguely characterized by their **priority**, and the Java specification [11] only guarantees that "threads with higher priority are executed in preference to threads with lower priority", but there are no specific mentions about how the scheduler handles overloads.

3. Garbage collection, while very convenient for the programmer, consumes execution time. Not only does it slow down the system, but it may become active at unpredictable times, suspending all the other threads for an indefinite period of time.

4. For real-time applications, the ability to safely and quickly interrupt an I/O operation is vital. We also need to know how much of the I/O has been completed, so that it can be resumed the next time the thread is run. Java lacks non-blocking and interruptible I/O ability.

Our research focuses on implementing a scheduling policy that will overcome these drawbacks. This involves extending threads with timing constraints and designing efficient algorithms for scheduling and feasibility analysis.

1.2 Related Work

The Real-Time Specification for Java [2] was crystallized in 1998-1999 in an attempt to standardize the efforts to incorporate Java platforms into real-time systems. The specification discusses several aspects of the Java platform which need to be revised in order to include real-time support, including scheduling, threads, memory management, synchronization, timers, asynchrony. Most of our research follows the specifications for the real-time scheduling system. We aim at implementing these specifications using efficient feasibility and scheduling algorithms.

The FLEX compiler [10] provides an ideal starting point. It is a Java bytecode to C compiler, with native support for several platforms, including Linux/i386. William Beebee [1] already implemented parts of the real-time specifications, in particular the real-time threads and the memory management system, so that we can focus directly on the scheduling issues.

1.3 Goals

In short, we want to provide an implementation of the real-time Java specifications for the scheduler. The implementation will give programmers the ability to communicate to the scheduler a more specific characterization of real-time threads. This includes precise timing constraints of the form:

> "*This thread has to be executed every 10 milliseconds. During each period, it must run for 3 milliseconds, and complete its work no later than 8 milliseconds after the start of the period. If the thread fails to meet these constraints, the scheduler will trigger an asynchronous event and invoke the event handler associated with this thread*".

Thus, use of the real-time threads models a **contract** between the scheduler and the threads, whereby the scheduler agrees to execute the threads, and the threads promise to yield, according to the timing constraints specified in the contract.

More formally, here is the set of requirements we make of the implementation of the scheduling system:

1. It must be able to quickly determine the feasibility of a proposed schedule for a given set of threads.

2. It must produce an ordering for the execution of a set of threads which minimizes the number of missed deadlines. In theory, the algorithm we use (EDF) is optimal in that, if it fails to produce a schedule which meets all the deadlines, then no such schedule exists.

3. It must allow some special real-time threads (see §3.4) to execute independently of the garbage collection thread, so that timing-critical code can run according to its schedule, without interruption from the GC thread.

4. It must be easily extensible to allow implementations of other scheduling algorithms.

Chapter 2 discusses the theoretical aspects of real-time scheduling and feasibility analysis. Chapter 3 presents implementation issues in the FLEX framework as well as the few places where due to efficiency considerations we slightly disobeyed the specification. Chapter 4 presents results and benchmarks and draws conclusions about the efficiency of the implementation.

Chapter 2

The EDF Algorithm

In this chapter, we analyze the Earliest Deadline First (EDF) algorithm. Section 1 introduces the terminology we will need for the rest of the thesis. Section 2 discusses the optimality of the EDF algorithm. It indicates the conditions under which the optimality holds, and also analyzes the tractability of real-time scheduling as we try to relax these conditions. Section 3 describes the feasibility analysis. Finally, section 4 gives the pseudocode for the actual scheduler we implemented.

2.1 Terminology, Assumptions and Notation

In this thesis, we follow the terminology presented in *Deadline Scheduling for Real-Time Systems* [9].

Definition 1. A *real-time thread* is an executable entity of work which, at a minimum, is characterized by a worst-case execution time (also known as *cost*) and a time constraint.

Definition 2. A *release time*, r, is a point in time at which an instance of a real-time thread is activated to execute.

Definition 3. A *deadline*, D, is a point in time by which an instance of a thread must complete. The deadline is usually measured relatively to the release time of that instance.

Stankovic [9] differentiates between three types of deadlines:

- **Hard deadlines**. It is vital that these deadlines always are met.
- **Soft deadlines**. It is desirable that these deadlines are met, but no catastrophe occurs if they are not.
- **Firm deadlines**. A thread which cannot complete by the given firm deadline should not execute at all.

In the implementation, we do not explicitly use this differentiation. Instead, we use priority and importance parameters (see §3.2) to separate soft from hard deadlines. Thus, in the event of an overload, threads with higher priority and importance execute in preference to threads with lower priority and importance. We do not, for the time being, support firm deadlines in the code, because the real-time specifications [2] do not mention them. The scheduling algorithm, however, is easy to extend to include firm deadlines.

There are three types of real-time threads: periodic, aperiodic, and sporadic.

Definition 4. A *periodic thread* is a real-time thread which is activated (released) regularly at fixed rates. The activation rate is known as the *period* of the thread and is designated by T. The time constraint for a periodic thread is a deadline D that can be less than or equal to the period.

In the most common case, the deadline for a periodic thread is equal to its period, meaning that the thread has to complete the amount of work indicated by its cost any time before the end of the period. Situations are conceivable where the deadline is less than the period, meaning that the thread has to complete its work sometime within D units of time from the beginning of the period. Stankovic [9] also allows the deadline to be greater than the period, but the need for this generalization seems rare for all practical purposes, and we do not handle this case in the thesis.

Definition 5. *Synchronous periodic threads* are a set of periodic threads where all first instances are released at the same time, usually considered time zero.

Definition 6. *Asynchronous periodic threads* are a set of periodic threads where threads can have their first instance released at different times.

Because of the theoretical results stated in §2.2, we can make our scheduling algorithm work regardless of the synchronicity of the threads. Therefore, we will assume every thread set to be asynchronous and we will not treat synchronous thread sets in any special manner.

Definition 7. An *aperiodic thread* is a real-time thread which is activated irregularly at some unknown and possibly unbounded rate. The time constraint is usually a deadline D.

In our implementation, we treat all Java (non-real-time) threads as aperiodic threads. We can do so because Java threads are characterized exclusively by priority and have no timing constraints. Also, we treat real-time threads without release parameters as aperiodic threads (§2.4). The best choice for the value of the deadline for these threads was determined experimentally (see §3.8).

Definition 8. A *sporadic thread* is a real-time thread which is activated irregularly with some known bounded rate. The rate is known as the *minimum interarrival period*, which is the minimum rate at which instances of this thread may be activated. The time constraint is usually a deadline D.

The rationale for making some threads sporadic is to limit the workload generated by these threads. Notice that, once activated, sporadic threads must complete their work units for that activation before their deadline, just as periodic threads. The minimum interarrival time is also noted by T, since there is no danger of confusion with the period T (which is only defined for periodic threads).

Definition 9. A *hybrid thread set* is a thread set containing both periodic and sporadic threads.

The implementation that we provide assumes all thread sets to be hybrid.

Definition 10. A set of threads is *schedulable* or *feasible* if all timing constraints are met, that is, all threads complete by their respective deadlines.

Definition 11. An *optimal real-time scheduling algorithm* is one which may fail to meet a deadline only if no other scheduling algorithm can meet the deadline.

Definition 12. A *non-idling scheduler* is a scheduler that never leaves the processor idle when there are pending jobs.

Definition 13. A *preemptive scheduler* is a scheduler that interrupts a thread of execution when its time slice runs out.

Definition 14. A *non-preemptive* (or *"cooperative"*) *scheduler* is a scheduler that always waits for the thread to yield control.

2.2 Optimality of the EDF Algorithm

The *earliest deadline first* algorithm is perhaps the most intuitive approach to scheduling: whenever the scheduler has to make a choice among the available threads, it activates the one with the earliest deadline. If several threads are tied for the earliest deadline, the scheduler may activate any of them (possible using other criteria, such as thread priorities, to further distinguish among threads).

Jackson [6] was the first to show the optimality of the EDF algorithm with respect to the maximum lateness. However, Jackson's rule only holds under a strict set of conditions:

1. There are n threads with a single instance each;
2. The threads are independent (there is no synchronization);
3. The release time is 0 for all the threads;
4. The scheduler is non-preemptive.

Real-life requirements force us to extend the theorem beyond all these assumptions: threads normally have multiple instances (that is, their existence spans several time slices), they may be dependent and may have different release times. With a

non-preemptive scheduler, the task of scheduling becomes NP-hard. However, with a preemptive scheduler the problem remains tractable and the EDF algorithm optimal.

The following theorem, due to George [5] shows that EDF is optimal for asynchronous hybrid thread sets, given that all threads have their deadline equal to their period (for periodic threads) or to their minimum interarrival time (for sporadic threads):

Theorem 1. (George et. al.) Non-preemptive non-idling EDF is optimal.

If asynchronous periodic threads are permitted with deadlines not necessarily equal to the periods, then scheduling was shown to be NP-hard [7]. However, such cases are the norm in practice and we must deal with them. Therefore, our implementation provides a preemptive non-idling EDF scheduler in an attempt to minimize the deadline overruns. Users implementing their own scheduling algorithm within our framework can choose to make their scheduler preemptive or non-preemptive (see §3.8).

Figure 2.1 shows an example of EDF scheduling for a set of two periodic threads and one sporadic thread. The real-time parameters for these threads are summarized in Table 2.1. The scheduler is preemptive and gives each thread a time slice of at most two time units.

Thread	Type	Release time	Period / MIT[1]	Deadline	Cost	Priority
1	Periodic	0	7	7	3	20
2	Periodic	2	10	10	3	15
3	Sporadic	5	9	9	3	15

Table 2.1: A hybrid asynchronous thread set

In this case, each thread has the deadline equal to the period or to the minimum interarrival time. The black blocks in Figure 2.1 indicate idle time units. The upward arrows indicate release times, while the downward arrows indicate deadlines. For periodic threads, the deadline of one instance coincides with the release time of the

[1]Minimum interarrival time.

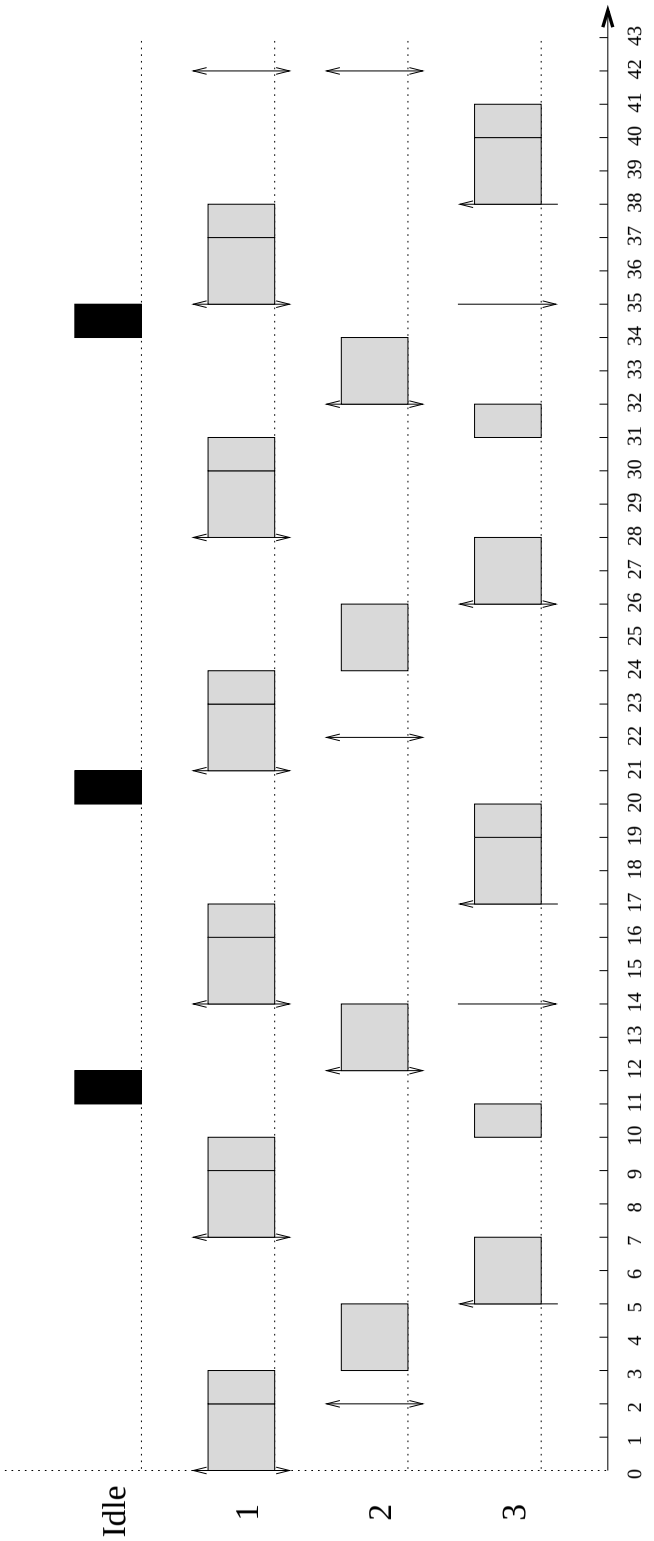

Figure 2.1: EDF schedule for a set of two periodic threads and one sporadic thread

next instance. Notice that, at times 7 and 28, the scheduler had to make a choice among threads 1 and 3, which were tied for the earliest deadline. Any choice would have generated a feasible schedule; in this case, thread priorities were used and thread 1 was selected in both instances.

2.3 Feasibility Analysis Under EDF Scheduling

In the previous section, we have shown that the EDF algorithm is optimal in that it always finds a feasible schedule if one exists. Now we approach the task of determining whether a feasible schedule exists for a given set of threads. A set of theorems, which we include here without proof, allow us to work our way from a particular case to the more general case.

Consider a set of n threads with cost C_i and period T_i (for periodic threads) or minimum interarrival time T_i (for sporadic threads). Each thread has deadline $D_i = T_i$.

Theorem 2. (Liu and Layland) [8] Any set of n synchronous periodic tasks with processor utilization $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$ is feasibly scheduled by EDF if and only if

$$U \leq 1.$$

Theorem 3. (Coffman) [3] Any set of n asynchronous periodic tasks with processor utilization $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$ is feasibly scheduled by EDF if and only if

$$U \leq 1.$$

We now turn to the analysis of hybrid thread sets:

Definition 15. A sporadic thread set is feasible if for any choice of release times compatible with the specified minimum interarrival times, the resulting job set is feasible.

Stankovic [9] proves that the worst-case scenario for a set of sporadic threads occurs when all the threads are activated synchronously and at the highest possible

rate. Therefore, it suffices to analyze the corresponding synchronous periodic thread set:

Definition 16. The synchronous periodic thread set *corresponding* to a sporadic thread set is obtained by replacing each sporadic thread with a periodic thread with release time 0 and period equal to the minimum interarrival time of the sporadic thread.

Theorem 4. A sporadic task set is feasible if and only if the corresponding synchronous periodic thread set is feasible.

Theorem 4 shows that feasibility analysis is as simple as summing up the processor utilization of all threads. Finally, we have to deal with aperiodic threads. In an attempt to comply with *The Real-Time Specification for Java* [2], aperiodic threads include non-real-time threads and real-time threads without temporal constraints. We decided to only schedule these threads for running when none of the periodic and sporadic threads are eligible. In Figure 2.1, we could run aperiodic threads at times 11, 20 and 34 when the processor is idle.

It is easy to see that this approach to aperiodic threads does not affect the feasibility of periodic and sporadic threads, because it is absolutely non-intrusive. On the other hand, the expected fraction of time when we can schedule aperiodic threads is

$$F = 1 - \sum_{i=1}^{n} \frac{C_i}{T_i}.$$

Aperiodic threads may starve if F is close to zero, and they are definitely the first to starve in an overload situation. However, we assume that to be safe. If programmers want to avoid the starvation of a particular thread, they can make that thread real-time and give it some appropriate timing constraints.

2.4 EDF Pseudocode

The pseudocode uses lowercase variables (d_i, t_i etc.) for absolute times, and uppercase variables (D_i, T_i etc.) for relative times. We associate with each thread i the following

values:

- d_i is the next deadline, expressed as an absolute time (as opposed to D_i, which is a relative time)
- t_i is the end of the next period, expressed as an absolute time. For sporadic threads, t designates the earliest absolute time when the thread can be rescheduled for execution.
- P_i is the amount of processor time that the thread has to receive before the deadline d_i. This value gets initialized to C_i at the beginning of each activation.

Algorithm SCHEDULER-INIT()

Comment: Initializes the set of threads to be scheduled;

```
1: for i ← 1, 2, ..., n do
2:    t_i ← the starting time of thread i
3:    d_i ← ∞
4:    P_i ← 0                    // No work to be completed before the starting time
5: end for
```

Algorithm SCHEDULER(now)

Comment: Chooses the next thread to run, given the absolute time now.

```
1: Update the units of work left for the previously running thread
2: for i ← 1, 2, ..., n do
3:    if now > d_i and P_i > 0 then
4:       // This thread has exceeded its deadline
5:       Schedule the deadline miss handler for thread i
6:    end if
7:    if now > t_i then
8:       // This thread has finished one period
9:       if thread i is periodic then
```

```
10:          d_i ← t_i + D_i          // Set the next deadline
11:          t_i ← t_i + T_i          // Set the next period
12:          P_i ← C_i                // Reset the processor usage counter
13:        else
14:          d_i ← ∞
15:          t_i ← t_i+T_i      // Cannot reactivate this thread before T_i units have elapsed
16:        end if
17:      end if
18:    end for
19:    // Now select the thread to run. First look at periodic threads and sporadic threads
20:    // which are in the middle of an instance
21:    deadline ← ∞;  thread ← 0;
22:    for i ← 1, 2, ..., n do
23:      if t_i ≤ now + T_i and P_i > 0 and d_i < deadline then
24:        deadline ← d_i
25:        thread ← i;      // thread i still has some work to complete during this period
26:      end if
27:    end for
28:    if thread ≠ 0 then
29:      return thread
30:    end if
31:    // Next, try to activate the highest-priority, idle sporadic thread
32:    priority ← 0;  thread ← 0;
33:    for i ← 1, 2, ..., n do
34:      if t_i is sporadic and d_i = ∞ and priority_i > priority then
35:        priority ← priority_i
36:        thread ← i;
37:      end if
38:    end for
39:    if thread ≠ 0 then
```

40: $t_{thread} \leftarrow now$ // *Mark the time when we activated this thread*

41: $d_{thread} = now + D_{thread}$

42: $P_i \leftarrow C_i$

43: **return** *thread*

44: **end if**

45: *// Finally, try to activate the highest-priority, idle aperiodic thread*

46: $priority \leftarrow 0$; $thread \leftarrow 0$;

47: **for** $i \leftarrow 1, 2, \ldots, n$ **do**

48: **if** t_i is aperiodic **and** $priority_i > priority$ **then**

49: $priority \leftarrow priority_i$

50: $thread \leftarrow i$;

51: **end if**

52: **end for**

53: *// If still no thread, return 0, meaning the processor stays idle*

54: **return** *thread*

The main principles beyond this scheduler design are:

- Lines 2 - 18 use the current time *now* to determine which threads to advance to their next periods. The deadline miss handlers are invoked for threads which advance to their next periods without having completed the work units for the current period.

- Lines 19 - 30 determine whether there are any periodic threads that still have work to do before their deadline. If so, the thread with the earliest deadline is returned.

- Lines 31 - 44 select and return the highest-priority sporadic thread that can be activated without violating the minimum interarrival interval constraints.

- Lines 45 - 54 select the highest priority aperiodic thread and return it, if no periodic or sporadic threads can be scheduled.

- The scheduler must keep track of what thread was running, and for how long it had been running, when the scheduler was called, so that it can correctly update the work units remaining for that thread.

- Once released, sporadic threads must complete before the deadline. For example, in Figure 2.1, thread 3 is released at time 26 and it must complete by time 35. In an overload situation, inactive sporadic threads will not be activated until the overload goes away. However, active sporadic threads are just as important as periodic threads and may preempt them in an overload situation, depending on their priorities. "Sporadic" should therefore not be confused with "unimportant".

- Aperiodic threads are only activated when all the periodic threads have completed their work units for their respective periods, and when there are no sporadic threads to run. Priorities are only used to distinguish between aperiodic threads. A high-priority aperiodic thread cannot preempt a low-priority periodic thread. The situation of a high-priority, aperiodic thread is inherently ambiguous and we assume that it is more important to meet the real-time constraints, whenever they exist, than to avoid priority inversion.

- The time slice allotted to the selected thread is the minimum between the default time slice and the work left for the thread before its deadline. For aperiodic threads, the default time slice is always allotted, because aperiodic threads do not have a per-period cost.

The feasibility algorithm simply checks whether the sum of the processor utilizations $\frac{C_i}{T_i}$ for periodic and sporadic threads is at most 1. Aperiodic threads are ignored in this analysis.

Chapter 3

Implementation Key Points

The previous chapter discussed scheduling and feasibility analysis from a purely theoretical point of view. Now we move on to presenting the practical aspects of real-time scheduling and how we implement these in the FLEX compiler. Section 1 shows how we employ program hooks to keep track of the elapsed time and to call the scheduler at appropriate times. Sections 2 and 3 present the class hierarchy associated with the priority parameters and with the release parameters of real-time threads. Section 4 presents no-heap real-time threads, a special class of real-time which is independent of the garbage collector. Section 5 talks about asynchrony and how we employ it to detect and handle threads that fail to meet their temporal constraints. Section 6 gives a detailed example of how the programming interface should be used to achieve functional real-time code. Section 7 compares the results from an actual program to the theoretical expectation. Section 8 describes the benchmarks we ran to determine what the appropriate time slice was for our scheduler, and how well it performs under various thread sets and loading factors. Finally, section 8 assesses the results and gives some ideas for future improvements.

3.1 Program Hooks

Each real-time thread runs for a time quantum which is determined by the scheduler. When the quantum expires, the system does a user-thread-level context switch; the

next thread to run is selected according to the policy of the scheduler.

At compile time, we insert **program hooks** at several points in the compiled program. The purpose of the program hooks is to determine whether the time quantum of the currently running thread has expired. At run time, a function `CheckQuanta` is called from each point where program hooks were inserted. If `CheckQuanta` determines that the quantum of the current thread has expired, then the scheduler is invoked; it suspends the currently running thread and it selects the next thread to be executed.

We are inserting program hooks at the beginning of each method and at the beginning of each loop, based on the reasonable assumption that these program points occur frequently enough to allow for a reliable timing behavior. We hoped it would be enough to insert hooks at the beginning of each method only, which would have avoided the slowdown from the numerous quanta checks. However, some computationally-intensive programs, such as the simple example given in Figure 3.1 may run for a long time without calling any functions. The benchmarks for these programs show an unacceptable number of deadline misses unless program hooks are inserted at the beginning of each loop as well (see §3.8).

3.2 Priority and Importance Parameters

Like `java.lang.Thread`'s, real-time threads have an associated **priority** – an integer from the range 10 to 37. Notice that the real-time priorities lie above the priorities of non-real-time threads, to make it clear that threads having any kind of real-time constraints are considered more important than non-real-time threads.

An additional scheduling metric is the **importance** of a thread, which the scheduler uses as a tie-breaker to differentiate among threads of the same priority. This may be useful, for example, if the scheduler manages a synchronous collection of periodic threads with the same period and the same priority level. In a situation of system overload, the importance values can help the scheduler decide which threads have precedence. The class `ImportanceParameters` extends the class `PriorityParameters`,

```
static SchedulingParameters sp = new ImportanceParameters(0, 9);

static RelativeTime rt1 = new RelativeTime(10, 0);
static RelativeTime rt2 = new RelativeTime(3, 0);

static ReleaseParameters rp =
         new PeriodicParameters(null, rt1, rt2, null, null, null);

static RealtimeThread thread = new RealtimeThread(sp, rp) {
  public void run() {
    System.out.println("Starting Thread!");
    long sum = 0;
    int i, j, k;
    for (i = 1; i <= 1000; i++)    // No function calls within these loops
      for (j = 1; j <= 1000; j++)
        for (k = 1; k <= 1000; k++)
          sum++;
    System.out.println(sum);
    System.out.println(PriorityScheduler.invocations + " invocations, "
                       + PriorityScheduler.missed + " missed deadlines");
  }
};
```

Figure 3.1: Sample source code that runs for a long time without calling any functions

which in turn extends from the abstract class `SchedulingParameters`.

The minimal scheduler required by the *Real-Time Specification for Java* [2] is priority-based. However, the EDF scheduler takes its decisions using the temporal constraints of threads and only uses priorities for breaking ties. We opted to overrule the specification here because explicit time constraints are a much better guide for the scheduler than priorities. Specifically, a real-time thread with tight timimg constraints will always preempt a real-time thread with more relaxed timing constraints, regardless of their relative priorities.

3.3 Release Parameters

Timing constraints are stored in objects inheriting from the abstract class `ReleaseParameters`. Every real-time thread will be associated with an instance of a subclass of `ReleaseParameters`. All types of timing constraints have these common properties:

- a **deadline**, meaning the maximum time for which the associated thread is permitted to run upon each invocation.
- a **deadline miss handler**, namely a method that is invoked every time the `run()` method of a thread is still executing after the deadline has passed (that is, if the thread fails to yield within the allocated time frame).

Periodic threads are associated with an instance of the `PeriodicParameters` class. In addition to the deadline, periodic parameters also include a start time (the first time when the associated thread must be executed) and a period (the interval between successive invocations of the thread). Knowing the deadline and the period of a periodic thread, we can get a very good approximation of the load the thread will place on the system. For example, a thread that requires invocation every 10 milliseconds and promises to yield in 3 milliseconds will roughly use 30% of the resources.
Sporadic threads are associated with an instance of the `SporadicParameters` class. In contrast with the periodic threads, they requested invocation at most as often as a given interarrival time. Together with the deadline, the minimum interarrival time gives us an upper bound on the load incurred on the system by this thread.
Aperiodic threads are associated with an instance of the `AperiodicParameters` class. The release parameters of an aperiodic thread include a cost, namely the amount of time that the associated threads is executed upon each invocation.

The release parameters of a thread can be changed at run time. The period or minimum interarrival time of a thread can be altered, and threads may even switch from periodic to aperiodic. However, there is no guarantee that the changes will take effect before the end of the current period.

3.4 No-Heap Real-Time Threads

As long as a real-time thread accesses the heap, it will be subject to unforeseen and indefinitely long interruptions from the garbage collector. Therefore, the Real-Time Specification for Java [2] defines `NoHeapRealtimeThread`, a subclass of the

`RealtimeThread` with additional memory constraints. Specifically, no-heap real-time threads are never allowed to allocate or reference any object allocated in the heap, or to manipulate the references to heap objects. Therefore, no-heap real-time threads may execute in preference to the garbage collector, or even interrupt it, without causing any inconsistencies in the program.

A sound implementation of the `NoHeapRealtimeThread` class was provided by William Beebee [1] for the FLEX compiler [10]. The implementation conforms to the Real-Time Specification for Java [2], and it provides a natural starting point for the scheduler implementation.

3.5 Asynchrony

In addition to real-time threads, the scheduler must be able to manage **asynchronous event handlers**. According to *The Real-Time Specification for Java* [2], "an `AsyncEventHandler` encapsulates code that gets run at some time after an asynchronous event occurs". An example of an asynchronous event is the execution of a deadline miss handler, which the scheduler calls whenever a real-time thread exceeds its deadline.

In the implementation, we do not execute the code for asynchronous events immediately. Instead, we schedule them for execution using the same scheduler we use for the real-time threads. Event handlers have real-time parameters just like normal threads. We are trying not to impose any limitations on the running time of `AsyncEventHandler`s, so they do not have to complete in a single invocation. We suggest however that they should have higher priorities and tighter temporal constraints than real-time threads. For example, in the case of the deadline miss handler, it is desirable to execute the handler before the next invocation of the thread that caused the event.

Sometimes it is convenient to run the even handler only once even if several events were triggered in connection with this handler. For example, we may only be interested in the number of missed deadlines, without executing code for each of them.

Therefore, the `AsyncEventHandler` declares a private field `fireCount`, accessible through three observers: `getAndClearFireCount()`, `getAndIncrementFireCount()`, and `getAndDecrementFireCount()`. These methods return the pending number of executions of the handler, and they also clear, increment or decrement this number respectively. Event handlers are not implemented by overriding the `run()` method. Instead, the `run()` method is predefined as:

```
public final void run() {
  while (fireCount > 0)
    handleAsyncEvent();
}
```

The programmer now has to implement the function `handleAsyncEvent()`. To execute the handler once for each pending invocation, this method can be written as:

```
public void handleAsyncEvent() {
  while (getAndClearPendingFireCount() > 0) {
       // Handle the event here
    }
}
```

To handle multiple firings at once, the `handleAsyncEvent()` method can be implemented as:

```
public void handleAsyncEvent() {
  int f = getAndClearPendingFireCount();
  System.out.println("Missed " + f + " deadline(s).");
  // Handle all the events at once
}
```

3.6 The Programming Interface

Here are the general steps that the programmer should take in designing a real-time system using FLEX:

3.6.1 Rewriting the Scheduler

If a different scheduling policy better fits the needs, the user can implement it by extending the abstract class `Scheduler` defined in *The Real-Time Specification for Java* [2], §4.2. The key method to override is `chooseThread`, which, given the current time, selects the next thread to be executed. The default implementation we provide is `PriorityScheduler`, whose complete code is given in Appendix A.

3.6.2 Defining the Real-Time Threads and Parameters

Real-time threads should implement the method `run()`, just like Java threads. Asynchronous event handlers (if desired) should implement the method `handleAsyncEvent()`. Below is a simple example of a program that defines two real-time threads which simply print a series of messages.

```
// Define a simple deadline miss handler that handles the deadlines
// missed by the realtime threads in this program. THe handler simply
// prints the number of missed deadlines. Notice that both threads in
// this program use the same handler; of course, the programmer can
// write a different handler for each thread.

// These are the real-time parameters of this handler
static SchedulingParameters sp1 = new ImportanceParameters(0, 9);
static RelativeTime rt1 = new RelativeTime(10, 0);
static RelativeTime rt2 = new RelativeTime(3, 0);
static ReleaseParameters rp1 =
    new PeriodicParameters(null, rt1, rt2, null, null, null);

// This is the actual handler code
static AsyncEventHandler handler = new AsyncEventHandler(sp1, rp1) {
    public void handleAsyncEvent() {
        int f = getAndClearPendingFireCount();
        System.out.println("Missed " + f + " deadline(s).");
    }
};
```

```
// Now we define the real-time parameters for the real-time threads
static SchedulingParameters sp2 = new ImportanceParameters(0, 9);
static SchedulingParameters sp3 = new ImportanceParameters(0, 9);

// Define the timing constraints to be used by the release parameters
static RelativeTime rt3 = new RelativeTime(5, 0);
static RelativeTime rt4 = new RelativeTime(2, 0);

// Define the release parameters for our threads.
// These parameters define a periodic thread that will run for
// 3 milliseconds out of every 10 milliseconds
static ReleaseParameters rp2 =
    new PeriodicParameters(null, rt1, rt2, null, null, handler);

// These parameters define a periodic thread that will run for
// 2 milliseconds out of every 5 milliseconds
static ReleaseParameters rp3 =
    new PeriodicParameters(null, rt3, rt4, null, null, handler);

// Here is the code that each thread has to execute. The constructor
// of each real-time thread must include the real-time parameters
// according to which the thread will run
static RealtimeThread thread2 = new RealtimeThread(sp2, rp2) {
    public void run() {
        System.out.println("Starting Thread 2!");
        for (int i = 1; i<=300; i++)
            System.out.println("*** 2: " + i);
    }
};

static RealtimeThread thread3 = new RealtimeThread(sp3, rp3) {
    public void run() {
        System.out.println("Starting Thread 3!");
        for (int i = 1; i<=300; i++)
            System.out.println("--- 2: " + i);
```

```
            waitForNextPeriod();
    }
};
```

At this point in the program, the scheduler does **not** take any action regarding these threads. The threads are created, but the scheduler will not be notified of their existence until they are started with the `RealtimeThread.start()` method.

Notice that one of the threads is calling the method `RealtimeThread.waitforNextPeriod()`. This is not mandatory in our implementation, because the scheduler is preemptive and will interrupt the running thread when its quantum expires. However, the method `waitForNextPeriod()` may be called explicitly if the thread wants to postpone the remaining work until the beginning of the next period. In the above example, the real-time thread will print approximately one line of output per time period.

3.6.3 Writing the Main Method

Notice that, under normal circumstances, the `main()` method will run in a Java (non-real-time) thread. As mentioned before, our scheduler gives non-RTJ threads the lowest priority. If the `main()` method is too long, it may starve after it invokes the first real-time thread. Therefore, we encourage the programmers to either make the `main()` method a real-time thread in itself, or to make it as short as possible. Using the thread definitions above, the appropriate main function would be:

```
public static void main(String[] args) {
    PriorityScheduler scheduler = PriorityScheduler.getScheduler();
    scheduler.setDefaultQuanta(2, 500000);
    System.out.println("Starting...");
    thread2.start();
    thread3.start();
    System.out.println("Is feasible: " + scheduler.isFeasible());
}
```

The call to `isFeasible()` should return `true`, because the total load is $3/10 + 2/5 = 0.7$. There are two observations to make about the main function:

- The scheduler is not created by a call to the constructor, but by a call to the static method `getScheduler()`. We used this approach to ensure that the scheduler is allocated in an immortal memory area, which will allow it to live until the end of the application; also, immortal memory areas are never subject to garbage collection [1].
- For real-time threads, the methods `RealtimeThread.addToFeasibility()` or `Scheduler.addToFeasibility()` don't have to be called explicitly to inform the scheduler of new real-time threads. These methods are called implicitly whenever the method `RealtimeThread.start()` is invoked. By default, threads are never removed from the scheduler until they terminate. To remove them explicitly, the programmer must call the `removeFromFeasibility` method.
- For non-real-time threads, the method `addToFeasibility` needs to be called explicitly if the scheduler must also handle those threads. By default, non-real-time threads are not handled by the scheduler.

3.7 Sample Output

Consider a thread set containing two periodic threads. The first one runs for 2 milliseconds every 10 milliseconds. The second one runs for 1 millisecond every 4 milliseconds. Thus, the total load incurred upon the scheduler is 0.45. Both threads are released at time 0. Figure 3.2 compares the schedule we expect to see in theory to the schedule generated by our scheduler.

The main differences are:

1. The threads do not start at time 0. Starting the threads is in itself a real-time job which implies the execution of `RealtimeThread.start()`. This method is long and takes a while to complete. In this case, the `main()` method started the

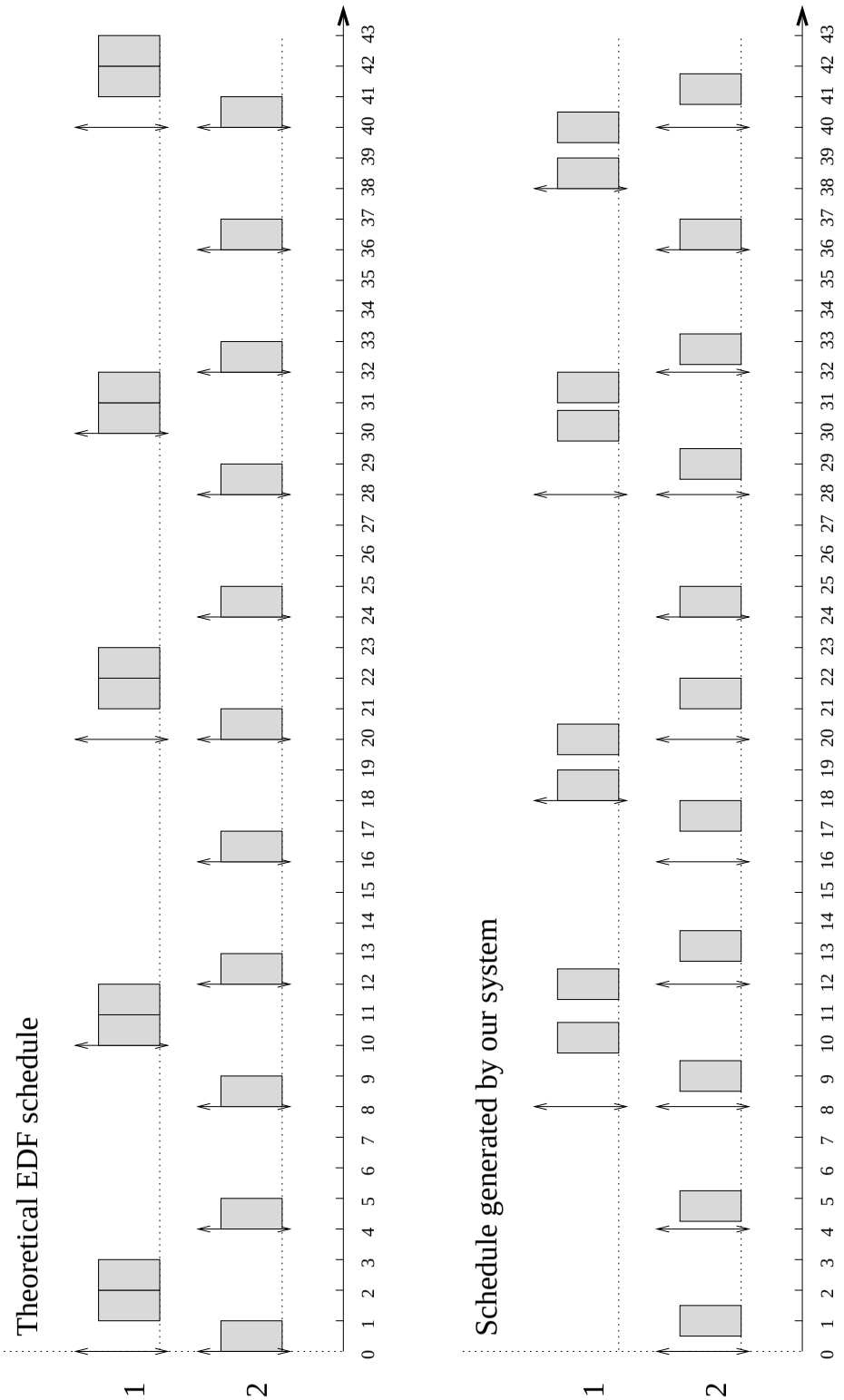

Figure 3.2: A theoretical EDF schedule (*left*) versus the practical results (*right*)

second periodic thread. While it was also starting the first thread, the second thread had enough time to complete its first two periods. That is why the actual release times of the threads are different in practice.

2. Consecutive activations of threads are often separated by an amount of internal work executed by the scheduler. That is why the threads drift away from their ideal schedule. For example, the activation of the second thread that was supposed to begin at the 20-th millisecond was actually pushed back by nearly one millisecond.

3. Although in this short graph no deadlines were missed, it is conceivable, under heavier loads and larger thread sets, that the scheduler may begin missing deadlines. The delays observed in practice may have cascading effects. For example, the late activation of thread 1 at the 19-th millisecond also caused a delay in the activation of thread 2 at the 20-th millisecond. These delays can accumulate over time and cause deadline misses.

These glitches become apparent under heavier loads. We ran the same experiment with thread 1 running for 4 milliseconds every 10 milliseconds, and thread 2 running for 2 milliseconds every 4 milliseconds. The total load was $0.4 + 0.5 = 0.9$. Figure 3.3 shows the results. The negative effects we notice are:

1. The scheduler took a longer time to start the first thread; this is a direct consequence of the heavier load. While the main thread was busy trying to start the threads, the second real-time thread began running, consuming 50% of the processor.

2. The first thread missed a deadline at time 21, and only ran for 3 milliseconds (instead of the required 4) between times 31 and 41.

3. The second thread missed two deadlines at time 24 and 32, and only ran for 1 millisecond (instead of the required 2) between times 28 and 32.

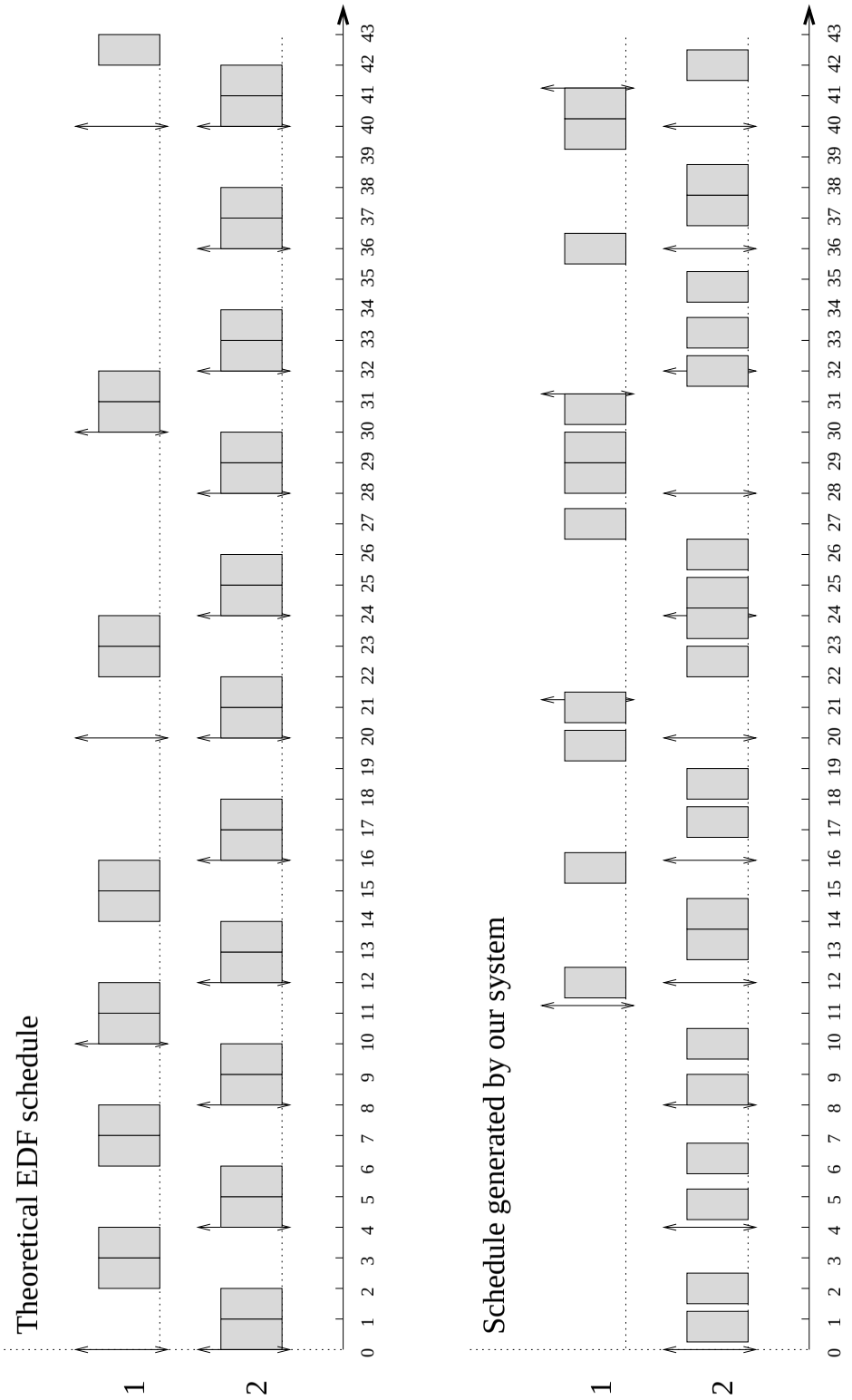

Figure 3.3: Theoretical versus practical results under a CPU load of 0.9

3.8 Benchmarks

We tested the robustness and scalability of our system from three points of view. Section 3.8.1 analyzes the performance of the scheduler to various loading factors, and gives an idea of the overhead it adds to the system. Section 3.8.2 analyzes the performance of the scheduler on thread sets of various sizes, to give an idea of its practical applicability. The last section presents the results of several tests we ran to determine what the optimal time quantum should be.

3.8.1 The Maximum Processor Load

Table 3.1 summarizes the results of a test set designed to determine the maximum processor load that our scheduler can handle without severe deadline misses. We compiled programs with processor loads varying from 0.2 to 0.9; the results for each program are averaged over 10 executions. Column 3 gives the total (real) running time of the program. Columns 4 and 5 provide scheduler statistics: the number of invocations and the total time consumed by the scheduler.

CPU Load	Deadline Misses	Running time	Invocations	Scheduler Time
0.2	0.4	3.43 s	13600	450 ms
0.3	0.7	3.15 s	10200	320 ms
0.4	1.4	2.91 s	8300	300 ms
0.5	1.9	2.53 s	5600	160 ms
0.6	2.0	2.28 s	5900	180 ms
0.7	3.3	1.85 s	4600	140 ms
0.8	6.8	1.96 s	2300	160 ms
0.9	9.6	2.02 s	1600	180 ms

Table 3.1: Behavior of the scheduler under various load factors

The execution time of the program decreases as the processor load increases, because there are fewer idle times. As expected, for low loads the time spent in the scheduler's `chooseThread()` method is proportional to the number of invocations of the scheduler, and missed deadlines are not an issue. At loads exceeding 0.7, anomalies

begin to occur. The frequency of deadline misses increases significantly; also, the duration of the program begins to increase at high loads, because the scheduler itself consumes more processor time. Overall, the scheduler performance is satisfactory given that the CPU load is below 0.7.

3.8.2 The Maximum Number of Supported Threads

Table 3.2 summarizes the results of a test set designed to determine the behavior of the scheduler when confronted with thread sets of various sizes. We ran experiments ranging from thread sets of size 2 to thread sets of size 8. All the threads where periodic, with identical parameter. The cumulative load of each thread set was 0.6 (so, for example, each thread in the thread set of size 2 incurred a load of 0.3, and each thread in the thread set of size 5 incurred a load of 0.12).

# of threads	Deadline Misses	Running time	Invocations	Scheduler Time
2	2.0	2.28 s	5900	180 ms
3	5.1	3.53 s	6800	340 ms
4	7.5	4.83 s	9100	720 ms
5	10.2	5.17 s	10500	1210 ms
6	16.1	6.71 s	13500	2300 ms
7	43.3	9.80 s	41800	4900 ms
8	85.0	18.30 s	76000	8300 ms

Table 3.2: Behavior of the scheduler under various sizes of the thread set

Because the implementation of the EDF algorithm is linear in the number of threads, the work load of the scheduler increases as the number of threads increases. This has an impact on the general behavior of the program, even though the work load generated by the threads is fixed at 0.6. Not only does the scheduler consume more CPU time, but the threads begin missing more deadlines and therefore they take longer to complete.

3.8.3 The Optimal Time Slice

Our real-time scheduler is preemptive. This means it has to make decisions regarding the duration that each thread must be allowed to run before it is replaced. Tables 3.3 and 3.4 summarize the experiments we ran to determine the appropriate time slice. The columns are:

- The default quantum in milliseconds;
- The number of scheduler invocations (averaged over 20 experiments);
- The number of deadlines missed (averaged over 20 experiments);
- The average number of deadlines missed per 10,000 invocations of the scheduler.

The first set of data are collected from a program which consists of two periodic threads. Thread 1 has a period of 10 milliseconds and a cost of 3 milliseconds. Thread 2 has a period of 5 milliseconds and a cost of 2 milliseconds. The total workload on the scheduler is 70%. The approximate duration of the program is 0.35 seconds.

Quantum [μs]	Invocations	Deadline misses	Average deadline misses
500	66000	45	6.81
1000	43800	21.6	4.93
1500	17500	17.75	10.14
2000	11900	5.8	4.87
2500	11300	5.7	5.04
3000	10800	10.2	9.44

Table 3.3: Default quantum benchmarks for short-lived threads

The second set of data are collected from a program which consists of two periodic threads. Thread 1 has a period of 100 milliseconds and a cost of 30 milliseconds. Thread 2 has a period of 50 milliseconds and a cost of 20 milliseconds. The total workload on the scheduler is again 70%. The approximate duration of the program is 3.3 seconds.

Quantum [μs]	Invocations	Deadline misses	Average deadline misses
1000	74200	12.0	1.61
2000	61600	12.4	2.01
3000	83700	16.3	1.94
4000	94500	14.2	1.50
5000	88700	21.0	2.36
10000	95300	11.9	1.24
15000	94000	8.9	0.94
20000	70700	9.1	1.28
25000	68800	9.4	1.36
30000	85000	10.3	1.21

Table 3.4: Default quantum benchmarks for longer-lived threads

For the first program, the best default time slice to use is somewhere between 2000 and 2500 microseconds. If the quantum is lower than 1000 microseconds, the number of scheduler invocations increases dramatically and the overhead slows down performance.

On the other hand, the number of deadlines missed by the second program is lowest when the time slice is around 15000 microseconds. Not contrarily to the intuition, the most appropriate quantum depends on the nature of the program being run. We have therefore decided to allow the programmer to invoke the method `Scheduler.setDefaultQuanta()` to define a time slice specific to the code being run.

3.9 Conclusions and Future Work

The FLEX compiler is a well-structured environment that permitted us to obtain fully functional code. The real-time scheduler performs well under reasonable loads (up to 0.7 of the CPU) and on average-sized thread sets. Its limitations stem mainly from. The linear complexity of the EDF algorithm, which creates a big scheduler overhead. FLEX makes it easy to explore new directions in scheduler designs. Here are a few ideas that deserve further investigation:

1. Implement **firm deadlines**. If the scheduler detects that an activation of a

thread will fail to meet its deadline, the scheduler should block the activation of that thread. This may prove beneficial in an overload situation if the firm thread has a low priority, because blocking it for a few periods may lighten the pressure on the system and give the scheduler time to cope with the overload. A possible approach to handling firm deadlines is to give the scheduler a small **look-ahead**, or the ability to analyse the evolution of the system in the short future and decide whether it can afford to activate a firm thread.

2. Make the scheduler **priority-based**, rather than EDF, under heavy overload. When the scheduler determines that it cannot avoid deadline overruns by one or more threads, it may consider switching to priority-based scheduling. Under EDF scheduling, priority inversion may occur if a lower-priority thread has an earlier deadline than a higher-priority thread. Under priority-based scheduling and in the event of an overload, the scheduler can at least attempt to satisfy the deadlines for vital threads.

3. Implement scheduling in time **logarithmic** in the number of threads. For now it is not obvious how this can be accomplished, because the scheduler must take into account the current time, and information (such as the deadline and the work units left during a period) needs to be updated frequently for most threads. A good starting point is provided by Dannenberg [4], who proposes a strategy with $O(1)$ scheduling / dispatching and $O(\log n)$ background work.

4. Switch to a more real-time environment than standard Linux. This will give a more accurate evaluation of the efficiency of this scheduler, for two reasons. First, the finest clock measurements that one can make on a standard Linux machine are in the order of microseconds, which may be too coarse to allow accurate results. Second, threads under Linux have a latency which can reach seconds in the worst case. A solution would be to install a real-time kernel patch. The default time slice needs to be reevaluated to accommodate the new environment.

Appendix A: Code

Below is the source code for the most important classes in the real-time package. The complete source codes can be found on the FLEX web page at `http://www.flex-compiler.lcs.mit.edu/`

PriorityScheduler.java

```
// PriorityScheduler.java, created by cata
// Copyright (C) 2001 Catalin Francu <cata@mit.edu>
// Licensed under the terms of the GNU GPL; see COPYING for details.
package javax.realtime;
import java.util.HashSet;
import java.util.Iterator;

public class PriorityScheduler extends Scheduler {
  // Real-time thread priorities
  static final int MAX_PRIORITY = 38;
  static final int MIN_PRIORITY = 11;
  static final int NORM_PRIORITY =
    (MAX_PRIORITY - MIN_PRIORITY) / 3 + MIN_PRIORITY;

  static HashSet allThreads = null;
  static HashSet disabledThreads = null;
  // The runtime constraints for the threads we're maintaining
  static ThreadConstraints thread[] =  null;
  public static RelativeTime defaultQuanta = null;
  static PeriodicParameters mainThreadParameters = null;
```

```
static PeriodicParameters noRTParameters = null;

// The first threadID that we can allocate
long nextThreadID = 0;
// The number of threads we are maintaining
int nThreads = 0;

// The number of missed deadlines and thenumber of times that the
// scheduler was invoked
public static long missed = 0;
public static long invocations = 0;

// The thread that chooseThread selected upon the previous invocation,
// and how long we've allowed that thread to run
long runningThread = 0;
RelativeTime runningTime = null;

static PriorityScheduler thisScheduler;

// Do not call this constructor; instead, call
// PriorityScheduler.getScheduler().
public PriorityScheduler() {
  super();
  if (thread == null) {
    thread = new ThreadConstraints[10];
    allThreads = new HashSet();
    disabledThreads = new HashSet();
    defaultQuanta = new RelativeTime(2, 0);
    runningTime = new RelativeTime(0, 0);
    mainThreadParameters =
      new PeriodicParameters(null, new RelativeTime(5, 0),
                             new RelativeTime(1, 0), null, null, null);
    noRTParameters =
      new PeriodicParameters(null, new RelativeTime(50, 0),
                             new RelativeTime(1, 0), null, null, null);
  }
```

```
}

// Creates and returns a scheduler of this type
public static PriorityScheduler getScheduler() {
  ImmortalMemory.instance().enter(new Runnable() {
      public void run() {
        PriorityScheduler.thisScheduler = new PriorityScheduler();
      }
    });

  return thisScheduler;
}

// Start maintaining this thread (called by the runtime system during
// execution of the thread.start() method).
public void addToFeasibility(final Schedulable t)
{
  allThreads.add(t);
  thread[nThreads] = new ThreadConstraints();
  thread[nThreads].threadID = ++nextThreadID;
  thread[nThreads].schedulable = t;
  thread[nThreads].beginPeriod = null;
  nThreads++;

   ImmortalMemory.instance().enter(new Runnable() {
       public void run() {
        // Give the runtime system a chance to update its data structures
        addThreadInC(t, nextThreadID);
       }
     });
}
protected native void addThreadInC(Schedulable t, long threadID);

// Remove this thread from the list of maintained threads (either
// from an explicit call or when the thread ends)
public void removeFromFeasibility(Schedulable t)  {
```

```
  long threadID = removeThreadInC(t);
  allThreads.remove(t);

  int i = 0;
  while ((thread[i] == null || thread[i].schedulable != t)
         && i < nThreads)
    i++;
  if (i < nThreads) {
    thread[i] = null; // To ensure deallocation
    thread[i] = thread[--nThreads];
  }
}

protected native long removeThreadInC(Schedulable t);

// Use this to notify the runtime system of the time slice we want
// to allot to the selected thread
protected native void setQuantaInC(long microsecs);

public long chooseThread(long micros) {
  invocations++;
  AbsoluteTime now = new AbsoluteTime(0, (int)micros*1000);

  if (nThreads == 0)
    return (runningThread = 0);

  ReleaseParameters rp = null;
  int earliest = -1; // The periodic thread returned by EDF
  // If earliest = -1, we'll choose the sporadic thread with the
  // earliest starting time
  int earliestSporadic = -1;
  for (int i = 0; i < nThreads; i++)
    if (thread[i] != null) { // if this thread is still alive
      // First, if this was the running thread, reduce its workLeft
      if (runningThread == thread[i].threadID) {
        thread[i].workLeft = thread[i].workLeft.subtract(runningTime);
```

```
  if (thread[i].workLeft.compareTo(RelativeTime.ZERO) == -1) {
    AsyncEventHandler h = thread[i].schedulable
                          .getCostOverrunHandler();
    if (h != null) h.run();
  }
}
// Second, update the thread parameters
if (thread[i].threadID == 1)
  rp = mainThreadParameters;
else if (thread[i].schedulable instanceof RealtimeThread)
  rp = thread[i].schedulable.getReleaseParameters();

// If the thread has no real time parameters, make it aperiodic
if (rp == null)
  rp = noRTParameters;
//        System.out.println(rp);
if (thread[i].beginPeriod == null) {
  // This is the first time we're handling this thread.
  // If the thread is sporadic, we'll set its endPeriod when we run
  // it for the first time.
  thread[i].beginPeriod = now;
  thread[i].endPeriod = (rp instanceof PeriodicParameters) ?
    thread[i].beginPeriod.add(((PeriodicParameters)rp).getPeriod()) :
    new AbsoluteTime(AbsoluteTime.endOfDays);
  thread[i].workLeft = rp.getCost();
  thread[i].deadline = thread[i].beginPeriod.add(rp.getDeadline());
}
else if (now.compareTo(thread[i].endPeriod) >= 0) {
  // This thread is passing into a new period
  // (1) Check to see if the thread missed its deadline
  if (thread[i].schedulable instanceof RealtimeThread &&
      thread[i].workLeft.compareTo(RelativeTime.ZERO) == 1) {
    missed++;
    AsyncEventHandler h = rp.getDeadlineMissHandler();
    if (h != null) h.run();
  }
```

```
    // (2) Update the thread constraints
    thread[i].beginPeriod.set(thread[i].endPeriod);
    if (rp instanceof PeriodicParameters)
      thread[i].beginPeriod.add(((PeriodicParameters)rp).getPeriod(),
                                thread[i].endPeriod);
    else
      thread[i].endPeriod.set(AbsoluteTime.endOfDays);
    thread[i].workLeft.set(rp.getCost());
    thread[i].beginPeriod.add(rp.getDeadline(), thread[i].deadline);
  }
  // Third, use the thread for the EDF algorithm The thread must
  // (1) not be disabled, (2) have some work left to do during
  // this period, (3) have the earliest deadline among all threads
  if (!disabledThreads.contains(new Long(thread[i].threadID)))
    if (rp instanceof PeriodicParameters ||
        (rp instanceof SporadicParameters &&
         thread[i].workLeft.compareTo(rp.getCost()) == -1)) {
      // This thread is either periodic, or it is sporadic AND we have
      // started running it, so now we have to finish this period in time
      if (thread[i].workLeft.compareTo(RelativeTime.ZERO) == 1 &&
          (earliest == -1 ||
           thread[i].deadline.compareTo(thread[earliest].deadline)==-1))
        earliest = i;
    }
    else if (rp instanceof SporadicParameters &&
             thread[i].workLeft.compareTo(rp.getCost()) == 0) {
      // This thread is sporadic and we haven't started this period yet,
      // So we'll remember it in case we have nothing urgent to do
      if (earliestSporadic == -1 ||
          thread[i].beginPeriod.
          compareTo(thread[earliestSporadic].beginPeriod) == -1)
        earliestSporadic = i;
    }

}
```

```
  // If nothing urgent, run a sporadic thread
  if (earliest == -1 && earliestSporadic != -1) {
    earliest = earliestSporadic;
    // We're activating a new period for this sporadic thread, so we have to
    // set startPeriod and endPeriod
    thread[earliest].beginPeriod.set(now);
    thread[earliest].beginPeriod
      .add(((SporadicParameters)thread[earliest].schedulable.
            getReleaseParameters()).getMinInterarrival(),
           thread[earliest].endPeriod);
  }

  // If the thread has enough work left to do, give it a full
  // quanta. Otherwise, give it only the time it needs.
  if (earliest != -1) {
    runningThread = thread[earliest].threadID;
    if (thread[earliest].workLeft.compareTo(defaultQuanta) == -1)
      runningTime.set(thread[earliest].workLeft.getMilliseconds(),
                      thread[earliest].workLeft.getNanoseconds());
    else
      runningTime.set(defaultQuanta.getMilliseconds(),
                      defaultQuanta.getNanoseconds());
    setQuantaInC(runningTime.getMilliseconds()*1000);
    return runningThread;
  }

  // Nothing to do, remain idle
  return runningThread = 0;
}

public void stopAll()  {
}

public void fireSchedulable(Schedulable h) {
  h.run();
}
```

```
public boolean noThreads() {
  return nThreads == 0;
}

public void disableThread(long threadID) {
  disabledThreads.add(new Long(threadID));
}

public void enableThread(long threadID) {
  disabledThreads.remove(new Long(threadID));
}

public String getPolicyName() {
  return "EDF";
}


/****************************** Priorities *********************************/
public int getMaxPriority() {
  return MAX_PRIORITY;
}

public int getMaxPriority(Thread thread) {
  return (allThreads.contains(thread)) ?
    MAX_PRIORITY : Thread.MAX_PRIORITY;
}

public int getMinPriority() {
  return MIN_PRIORITY;
}

public int getMinPriority(Thread thread) {
  return (allThreads.contains(thread)) ?
    MIN_PRIORITY : Thread.MIN_PRIORITY;
}
```

```
public int getNormPriority() {
  return NORM_PRIORITY;
}

public int getNormPriority(Thread thread) {
  return (allThreads.contains(thread)) ?
    NORM_PRIORITY : Thread.NORM_PRIORITY;
}

/************************* Feasibility algorithm ************************/

protected static native int stopSwitchingInC();
protected static native int restoreSwitchingInC(int state);

// This one's a bit twisted, but it's much more
// implementation-friendly. We decide feasibility by calling
// changeIfFeasible with no parameters.
public static boolean isFeasible() {
  return changeIfFeasible(null, null, null);
}

// changeIfFeasible()
//   This is where the actual feasibility decision is made.
//   Algorithm: Plain EDF -- add up the fractions cost/period for
//   periodic threads, cost/minInterarrival for sporadic threads.
public static boolean changeIfFeasible(Schedulable schedulable,
                                       ReleaseParameters release,
                                       MemoryParameters memory) {
  if (schedulable != null &&
      !allThreads.contains(schedulable)) {
    return false; // We are not responsible for this thread.
  }
  int switchingState = stopSwitchingInC();
  double load = 0.0;
  HashSet groupsSeen = new HashSet();
```

```
for (Iterator i = allThreads.iterator(); i.hasNext(); ) {
  Schedulable s = (Schedulable) i.next();
  // Use the release parameters for each Schedulable, except for the one
  // we are trying to change, for which we'll use the newly proposed
  // parameters.
  ReleaseParameters rp = (s == schedulable) ?
    release : s.getReleaseParameters();

  if (rp != null) // Main thread (and possibly other threads) have null rp
    // Periodic threads already have a cost and a period
    if (rp instanceof PeriodicParameters) {
      load += (double)((PeriodicParameters)rp).getCost().getMilliseconds()/
        ((PeriodicParameters)rp).getPeriod().getMilliseconds();
    }
    else if (rp instanceof SporadicParameters) {
      load += (double)((SporadicParameters)rp).getCost().getMilliseconds()/
        ((SporadicParameters)rp).getMinInterarrival().getMilliseconds();
    }
  // We must look up the period in the ProcessingGroupParameters
  // of the thread.
    else {
      ProcessingGroupParameters pgp = s.getProcessingGroupParameters();
      if (!groupsSeen.contains(pgp)) { // Haven't seen this group before
        groupsSeen.add(pgp);
        load += (double)pgp.getCost().getMilliseconds() /
          pgp.getPeriod().getMilliseconds();
      }
    }
}

System.out.println("Load = " + load);
if (load <= 1.0 && schedulable != null) {
  schedulable.setReleaseParameters(release);
  schedulable.setMemoryParameters(memory);
}
```

```
    restoreSwitchingInC(switchingState);
    return (load <= 1.0);
  }
}
```

ThreadConstraints.java

The runtime parameters associated with each thread are:

- `threadID` – An identification number used to communicate between the real-time Java package and the runtime component written in C. Passing the threads themselves would have been time-consuming and error-prone.
- `workLeft` – A relative time indicating the amount of processor time that the thread needs before its deadline. When this value is negative, the cost overrun handler needs to be called. When this value is positive and the deadline has passed, the deadline miss handler needs to be called.
- `beginPeriod` – An absolute time indicating the beginning of the current period. For periodic threads, the beginning of a period is equal to the end of the previous period. For sporadic threads, the beginning of the period is set to the time when the thread is released.
- `endPeriod` – An absolute time indicating the end of the current period (for periodic threads) or the end of the minimum interarrival time (for sporadic thread). After this time, all the runtime parameters need to be updated: periodic threads are advanced to the next period, and sporadic threads become eligible for reactivation.
- `deadline` – An absolute time indicating the point in time by which the thread has to complete all its work for that period.

```
// ThreadConstraints.java, created by cata
// Copyright (C) 2001 Catalin Francu <cata@mit.edu>
```

```
// Licensed under the terms of the GNU GPL; see COPYING for details.
package javax.realtime;

// These are the runtime-constraints associated with a thread.
public class ThreadConstraints {
  Schedulable schedulable; // The thread itself
  long threadID;
  RelativeTime workLeft;
  AbsoluteTime beginPeriod, endPeriod, deadline;
}
```

Other Examples

Here are two other examples of programs we have run through our real-time system.

ToyTree.java

This is a program that builds a complete binary tree of given depth and then it rotates random nodes a given number of times.

```
class ToyTree {
    public static void main(String [] args) {
    int asize=0;
    int repeats=0;
    boolean RTJ = false;
    boolean stats = false;
    javax.realtime.MemoryArea ma = null;
    javax.realtime.MemoryArea mb = null;
    try {
        asize=Integer.parseInt(args[0]);
        repeats=Integer.parseInt(args[1]);
        if (RTJ=!args[2].equalsIgnoreCase("noRTJ")) {
        if (args[2].equalsIgnoreCase("CT")) {
            ma = new javax.realtime.CTMemory(Long.parseLong(args[4]));
            mb = new javax.realtime.CTMemory(Long.parseLong(args[4]));
        } else if (args[2].equalsIgnoreCase("VT")) {
```

```
        ma = new javax.realtime.VTMemory(1000, 1000);
        mb = new javax.realtime.VTMemory(1000, 1000);
    } else {
        throw new Exception();
    }
    stats = args[3].equalsIgnoreCase("stats");
    }
} catch (Exception e) {
    System.out.println("Toy Tree <tree depth> <repeats> " +
            "<noRTJ | CT | VT> [stats | nostats] [ctsize]");
    System.exit(-1);
}

long start;
if (RTJ) {
    ToyTreeRTJ ta=new ToyTreeRTJ(ma,asize,repeats);
    ToyTreeRTJ tb=new ToyTreeRTJ(mb,asize,repeats);
    start=System.currentTimeMillis();
    ta.start();
    tb.start();
    try {
    ta.join();
    tb.join();
    } catch (Exception e) {System.out.println(e);}
} else {
    ToyTreeNoRTJ ta=new ToyTreeNoRTJ(asize,repeats);
    ToyTreeNoRTJ tb=new ToyTreeNoRTJ(asize,repeats);
    start=System.currentTimeMillis();
    ta.start();
    tb.start();
    try {
    ta.join();
    tb.join();
    } catch (Exception e) {System.out.println(e);}
}
long end=System.currentTimeMillis();
```

```
    System.out.println("Elapsed time(mS): "+(end-start));
    if (stats) {
        javax.realtime.Stats.print();
    }
    }
}

class ToyTreeRTJ extends javax.realtime.RealtimeThread {
    public ToyTreeRTJ(javax.realtime.MemoryArea ma, int size, int repeat) {
    super(ma);
    this.size=size;
    this.repeat=repeat;
    }

    int size;
    int repeat;

    static class TreeEle {
    public TreeEle(TreeEle l, TreeEle r) {
        left=l;
        right=r;
    }
    TreeEle left;
    TreeEle right;
    }

    TreeEle buildtree(int size) {
    if (size==0)
        return null;
    return new TreeEle(buildtree(size-1),buildtree(size-1));
    }

    void fliptree(TreeEle root) {
    if (root==null)
        return;
    TreeEle left=root.left;
```

```
    TreeEle right=root.right;
    if (left!=null) {
        TreeEle temp=left.left;
        left.left=left.right;
        left.right=temp;
        fliptree(left.left);
        fliptree(left.right);

    }
    if (right!=null) {
        TreeEle temp=right.left;
        right.left=right.right;
        right.right=temp;
        fliptree(right.left);
        fliptree(right.right);
    }
    root.left=right;
    root.right=left;
    }

    public void run() {
    TreeEle root=buildtree(size);
    for(int i=0;i<repeat;i++)
        fliptree(root);
    }
}

class ToyTreeNoRTJ extends Thread {
    public ToyTreeNoRTJ(int size, int repeat) {
    super();
    this.size=size;
    this.repeat=repeat;
    }

    int size;
    int repeat;
```

```
static class TreeEle {
public TreeEle(TreeEle l, TreeEle r) {
    left=l;
    right=r;
}
TreeEle left;
TreeEle right;
}

TreeEle buildtree(int size) {
if (size==0)
    return null;
return new TreeEle(buildtree(size-1),buildtree(size-1));
}

void fliptree(TreeEle root) {
if (root==null)
    return;
TreeEle left=root.left;
TreeEle right=root.right;
if (left!=null) {
    TreeEle temp=left.left;
    left.left=left.right;
    left.right=temp;
    fliptree(left.left);
    fliptree(left.right);

}
if (right!=null) {
    TreeEle temp=right.left;
    right.left=right.right;
    right.right=temp;
    fliptree(right.left);
    fliptree(right.right);
}
```

```
    root.left=right;
    root.right=left;
    }

    public void run() {
    TreeEle root=buildtree(size);
    for(int i=0;i<repeat;i++)
        fliptree(root);
    }
}
```

Fib.java

This is a program that computes the n-th Fibonacci number recursively (i.e., in time exponential in n).

```
import javax.realtime.RealtimeThread;
import javax.realtime.CTMemory;
class Fib {
  public static void main(String args[]) {
    int i = Integer.parseInt(args[0]);
    System.out.println("fib("+i+")");
    Compute c = new Compute();
    c.i = new Integer(i);
    CTMemory scope = new CTMemory(1000000);
    scope.enter(c);
  }
}
class Compute2 implements Runnable {
  Integer i;
  public void run() {
    Compute c = new Compute(i);
    c.start();
    try {
      c.join();
    } catch (Exception e) { System.out.println(e); }
```

```
    System.out.println("fib("+i.toString()+")="+c.target.toString());
  }
}
class Compute extends RealtimeThread {
  Integer source;
  Integer target;
  Compute(Integer s) {
    source = s;
  }
  Integer target() {
    return target;
  }
  public void run() {
    int v = source.intValue();
    if (v <= 1) {
      target = new Integer(v);
    } else {
      Compute c1 = new Compute(new Integer(v-1));
      Compute c2 = new Compute(new Integer(v-2));
      c1.start();
      c2.start();
      try {
        c1.join();
        c2.join();
      } catch (Exception e) { System.out.println(e); }
      target = new Integer(c1.target().intValue() + c2.target().intValue());
    }
  }
}
```

Bibliography

[1] W. Beebee, *Region-Based Memory Management for Real-Time Java*, M.Eng. Thesis, MIT, 2001.

[2] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, M. Turnbull, *The Real-Time Specification for Java*, Addison-Wesley, 2000.

[3] E. G. Coffman, Jr., *Computer and Job-Shop Scheduling Theory*, Wiley, New York, 1976.

[4] R. B. Dannenberg, *A Real-Time Scheduler / Dispatcher*, Proceedings of the 14th International Computer Music Conference, Köln, 1988.

[5] L. George, P. Muhlethaler, N. Rivierre, *Optimality and Non-Preemptive Real-Time Scheduling Revisited*, Rapport de Recherche RR-2516, INRIA, Le Chesnay Cedex, France, 1995.

[6] J. R. Jackson, *Scheduling a Production Line to Minimize Maximum Tardiness*, Research Report 43, Management Science Research Project, University of California, Los Angeles, 1955.

[7] J. Y.-T. Leung, M. L. Merrill, *A Note on Preemptive Scheduling of Periodic, Real-Time Tasks*, Information Processing Letters, 11(3), 1980.

[8] C. L. Liu, J. W. Layland, *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, Journal of the Association for Computing Machinery, 20(1), 1973.

[9] J. A. Stankovic, M. Spuri, K. Ramamritham, G. Buttazzo, *Deadline Scheduling for Real-Time Systems*, Kluver Academic Publishers, 1998.

[10] The FLEX Compiler Infrastructure, `http://www.flex-compiler.lcs.mit.edu/`

[11] Sun Microsystems Inc., *Java^{TM} 2 SDK Standard Edition Documentation*, `http://java.sun.com/j2se/1.3/docs/`