The Bossa Framework for Scheduler Development

Julia L. Lawall
DIKU, University of Copenhagen
2100 Copenhagen Ø, Denmark
julia@diku.dk

Gilles Muller
Obasco Group, EMN-INRIA, LINA
44307 Nantes Cedex 3, France
gilles.muller@emn.fr

1. INTRODUCTION

Process scheduling is an old problem, but there is no single scheduler that is perfect for all applications. Indeed, in the last few years, the emergence of new applications, such as multimedia and real-time applications, and new execution environments, such as embedded systems, has given rise to a host of new scheduling algorithms [2, 3, 5, 7, 14, 18, 19, 21, 22, 23, 24, 25]. Nevertheless, because these algorithms are typically highly specialized, few have been included in commercial operating systems (OSes).

Ideally, when the scheduling behavior required by an application is not available, the application programmer can implement a new scheduler in the target OS. Nevertheless, scheduler programming at the kernel level is a difficult task. First, there is no standard interface for implementing schedulers. Thus, the programmer must identify the parts of the kernel that should be modified and the code that should be written in each case. Because scheduling is affected by all kernel services, this analysis requires a global understanding of the kernel behavior. The analysis is further complicated by the pseudo-parallelism present in the kernel due to interrupts. Second, few debugging tools are available at the kernel level. Indeed, any errors in kernel code are likely to crash the machine, making bugs difficult to track down. Together these issues imply that the kind of expertise required to successfully integrate a new scheduler into an existing OS is outside the scope of application programmers.

Bossa. Bossa is a framework to allow application programmers to implement kernel schedulers easily and safely. This framework defines a scheduling interface that is instantiated in a standard OS by an OS expert. Schedulers are written using a *domain-specific language* (DSL) that provides high-level scheduling-specific abstractions to simplify the programming of scheduling policies. To enable compile-time verification that a scheduler interacts correctly with the target kernel, the OS expert configures the DSL compiler with a model of the kernel's scheduling behavior, including information about process state transitions and interrupts. Schedulers can either be compiled with the kernel or dynamically loaded into a scheduling hierarchy. Because Bossa extends a standard OS, applications can continue to use a standard execution environment (drivers, libraries, etc.).

We have implemented Bossa in the Linux 2.4.18 and Linux 2.4.24 kernels. Bossa has been used to implement a variety of scheduling policies, including policies directed towards multimedia applications such as progress-based scheduling [22], policies directed towards real-time systems such as rate monotonic and earliest-deadline first (EDF) [4], and general-purpose policies such as the policy of Linux. Most policies amount to under 200 lines of Bossa code and were implemented in a few hours beyond the time required to understand the scheduling algorithm. Some of these policies were implemented by students with no previous kernel programming experience. Overall, we have found that the use of Bossa allows the scheduler programmer to focus on the features of the policy to be implemented rather than on the details of integrating a new scheduler into an existing OS.

In the rest of this paper, we describe some features of the Bossa framework. Section 2 introduces the Bossa DSL. Section 3 describes an aspect-oriented approach to integrating Bossa into an OS such as Linux. Section 4 evaluates the performance of our approach and illustrates some applications. Section 5 concludes and describes future work. Bossa has been presented in detail elsewhere [1, 10, 11, 15, 16]. This paper thus provides a brief overview of some of the highlights of the Bossa framework.

2. THE BOSSA DSL

The goal of the Bossa DSL is to express scheduling policies in a clear, concise and verifiable way. A Bossa scheduling policy includes a set of declarations and a set of handlers for kernel scheduling events. We introduce the language using excerpts of an implementation of an EDF scheduling policy [13], shown in Figure 1, which illustrates most of the language features. The complete implementation is 162 lines of Bossa code. The complete policy and a grammar of the Bossa DSL are available at the Bossa web site.[1]

Declarations. The declarations of a scheduling policy define the process attributes, the scheduling states, and the ordering of processes.

The `process` declaration lists the policy-specific attributes associated with each process. Those of the EDF policy are the period and the Worst-Case Execution Time (WCET) supplied by the process, the process's current deadline, and a timer that is used to maintain the period.

The `states` declaration lists the set of process states that

Verbatim copying and distribution of this entire article are permitted world-wide, without royalty, in any medium, provided this notice, and the copyright notice, are preserved.
Libre Software Meeting July 5-9, 2005, Dijon, France.
Copyright 2005 Julia L. Lawall.

[1] `http://www.emn.fr/x-info/bossa`

```
scheduler EDF = {
  process = {
    time  period;
    time  wcet;
    time  current_deadline;
    timer period_timer;
  }
  states = {
    RUNNING  running : process;
    READY    ready   : select queue;
    READY    yield   : process;
    BLOCKED  blocked : queue;
    BLOCKED  computation_ended : queue;
    TERMINATED terminated;
  }
  ordering_criteria = { lowest current_deadline }

  handler (event e) {
    On unblock.preemptive {
      if (e.target in blocked) {
        if (!empty(running) && e.target > running) {
          running => ready;
        }
        e.target => ready;
      }
    }
    On bossa.schedule {
      if (empty(ready)) { yield => ready; }
      select() => running;
      if (!empty(yield)) { yield => ready; }
    }
    ...
  }
}
```

Figure 1: EDF scheduler

are distinguished by the policy. Each state is associated with a state class (`RUNNING`, `READY`, `BLOCKED`, or `TERMINATED`) describing the schedulability of processes in the state and an implementation as either a process variable (`process`) or a queue (`queue`). The names of the states of the EDF policy are mostly intuitive. For example, the `ready` state is in the `READY` state class, meaning that it contains processes that are able to run. This state is also designated as `select`, meaning that processes are elected from this state. The `computation_ended` state stores processes that have completed their computation within the current period.

The `ordering_criteria` allows the comparison of two processes according to a sequence of criteria based on the values of their attributes. The EDF policy favors the process with the lowest current deadline. The annotation `select` in the declaration of the `ready` state indicates that the associated queue is sorted according to this criterion.

Event handlers. Event handlers describe how a policy reacts to scheduling-related events that occur in the kernel. Examples of such events include process blocking and unblocking and the need to elect a new process. We show only the definitions of the handlers `unblock.preemptive` and `bossa.schedule`, which include most of the scheduling-specific language constructs.

Event handlers are parameterized by an event structure, `e`, that includes the *target process*, `e.target`, affected by the event. The event-handler syntax is based on that of a subset of C, to make the language easy to learn, and provides specific constructs and primitives for manipulating processes and their attributes. These include constructs for testing the state of a process (*exp* `in` *state*), testing whether there is any process in a given state (`empty(`*state*`)`), testing the relative priority of two processes (exp_1 `>` exp_2), and changing the state of a process (*exp* `=>` *state*). The latter operation is the only means of affecting the state of a process, and both removes the process from its current state and adds it to the new one, thus ensuring by construction that every process is always in exactly one state.

An `unblock.preemptive` event occurs when a process unblocks. The EDF handler checks whether the process is actually blocked (`e.target in blocked`), and if so sets the state of the target process to `ready` making it eligible for election. The handler also checks whether there is a running process (`!empty(running)`) and if so whether the target process has a higher priority than this running process (`e.target > running`). If both tests are satisfied, the state of the running process is set to `ready`, thus causing the process to be preempted.

Process election is performed by the `bossa.schedule` event handler. The kernel invokes this handler only when a new process must be elected and there are some eligible processes. The handler must change the state of some `READY` process to a state in the `RUNNING` state class and is the only handler that is allowed to do so. In the EDF `bossa.schedule` handler the main effect is to elect a process from the state designated as `select` (`ready`, in the case of the EDF policy) using the `select()` primitive, which is defined in terms of the `ordering_criteria`. Nevertheless, because the EDF policy has two `READY` states, `ready` and `yield`, it may occur that the only `READY` process is actually in the `yield` state. In this case, the handler first changes the state of the `yield` process to `ready`. The policy furthermore implements the strategy that a yielding process only defers to other eligible processes until the next context switch. Thus, the handler terminates by changing the state of any process remaining in the `yield` state to `ready`.

The structure of the EDF event handlers is quite simple, and is typical of that of most of the handlers found in Bossa policies. This simplicity, combined with the domain-specific operators and the characterization of process states by state classes, makes it easy for a programmer to understand the algorithm implemented by a Bossa scheduling policy and enables the Bossa DSL compiler to automatically verify that an event handler satisfies OS-specific requirements [10].

Bossa supports both the construction of a single process scheduler, as described above, and the construction of a hierarchy of schedulers. The use of a hierarchy allows multiple process schedulers, each satisfying particular scheduling needs, to coexist in a running OS. In a Bossa hierarchy, the root and interior nodes are *virtual schedulers*, which only manage other schedulers, and the leaf nodes are *process schedulers*, which only manage processes [11].

3. PREPARING LINUX FOR BOSSA

Figure 2 illustrates the architecture of the Bossa framework. A Bossa scheduling policy is compiled by the Bossa DSL compiler into a component that is implemented as a kernel module. This component exports an interface requesting event notifications from the OS kernel whenever process state changes occur. The component then uses the informa-

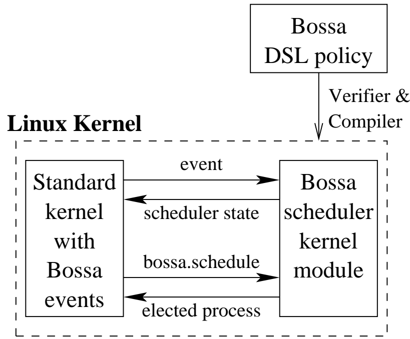


Figure 2: Bossa architecture

tion received via these event notifications to make scheduling decisions, including the election of a new process.

Preparing an OS kernel for use with Bossa thus requires inserting event notifications throughout the kernel, wherever process state changes occur, in accordance with the Bossa interface. A standard solution to extending an OS, such as Linux, with a new feature is to perform the integration by hand and to distribute the result as a patch file. Manual integration is, however, tedious and error-prone, and the result is limited to a single version of the OS. To obtain a solution that is both more manageable and more flexible, we have turned for inspiration to aspect-oriented programming (AOP) [8].

AOP is a programming technique that is targeted towards providing a modular implementation of functionalities that crosscut an application. The implementation of such a functionality is isolated in an *aspect*, which contains a collection of code fragments, known as *advice*, and a formal description, known as a *pointcut*, of the positions at which these fragments should be inserted into the target application. To see how AOP can be used for Bossa, we consider the integration of the `unblock.preemptive` event notification into Linux 2.4. In this case, the Bossa functionality completely subsumes the primitive Linux process wakeup function `try_to_wake_up`. Thus, the Bossa aspect contains a pointcut specifying that any call to `try_to_wake_up` should be replaced and an advice specifying that the replacement should call the Bossa unblock event notification function `rts_unblock` with the same set of arguments.

Existing approaches to AOP typically provide pointcut languages that can modify function calls and some kinds of variable references. Because the need for a scheduling interface was not anticipated by the Linux developers, however, the needs of the Bossa interface do not always match up with the structure of the Linux kernel. As an example, we consider the `bossa.schedule` event. The semantics of this event requires that the policy elect a new process and thus coincides roughly with the behavior of the Linux `schedule` function. The Linux `schedule` function, however, does more than elect a new process; it also initiates the context switch and performs some other bookkeeping actions. Thus, we cannot simply replace a call to `schedule` with the `bossa.schedule` event notification; we must instead specify the fragment of the `schedule` definition that the event notification should replace.

To precisely specify the fragments of Linux code that should be replaced by an event notification, we add features based on temporal logic to the pointcut language. Temporal logic is commonly used to express properties of event sequences, particularly in the context of model checking [6]. Properties include whether an event may eventually occur, or whether it is guaranteed to eventually occur. In the context of specifying properties of programs, we use temporal logic to describe the operations that occur on various paths through a control-flow graph. This use of temporal logic was pioneered by Lacey *et al.*, who use this logic to define rewrite rules that describe common compiler optimizations [9].

In the case of the `bossa.schedule` event notification, we observe that in the Linux `schedule` function, the fragment of code that deals with process election appears after the taking of the runqueue lock and before the releasing of this lock. The Bossa aspect thus specifies that a `bossa.schedule` event notification should replace the maximal code fragment in the `schedule` function in which every instruction satisfies the following property:

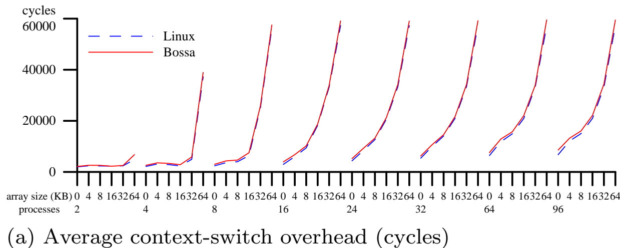
$$\mathsf{AF}\Delta(\texttt{spin_lock_irq(\&runqueue_lock)}) \wedge \mathsf{AF}(\texttt{spin_unlock_irq(\&runqueue_lock)})$$

The operator $\mathsf{AF}\Delta\phi$ matches any code point from which all paths (indicated by A) in a backward direction (indicated by Δ) eventually (indicated by F) reach a point where ϕ is true. In the formula above, ϕ is specified as the code fragment that should be matched. The operator $\mathsf{AF}\phi$ is similar, but considers control-flow in a forward direction. In the formula above, the conjunction of these two operations captures code fragments that are between the taking and releasing of the runqueue lock. These fragments are then as a whole replaced by the `bossa.schedule` event notification. Error checking rules can also be provided in a similar style to check for cases where only some of the control-flow paths satisfy the required property. Using such rules, the Bossa aspect can be applied to multiple versions of the Linux kernel, with the assurance that unexpected code patterns will be detected.

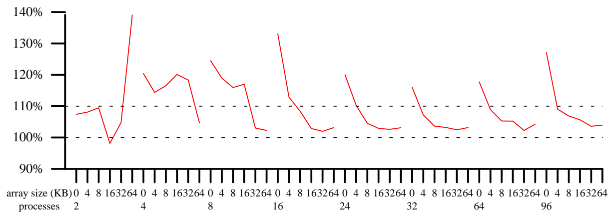
We have implemented an aspect system that provides the above features using the CIL program analysis and transformation framework [17]. Most of the Bossa interface is implemented by an aspect consisting of 23 pairs of pointcuts and corresponding advice. In a few cases, hand modifications are needed *e.g.* in assembly language code and macro definitions, due to limitations of CIL. The aspect code is integrated with the Bossa interface, which thus both describes the interaction between the OS kernel and the Bossa policy, and specifies the means of updating the OS kernel to perform its part of this interaction [15]. We have used the aspect with the standard versions of Linux 2.4.18 and Linux 2.4.24, and with a version of Linux 2.4.24 to which a patch providing high resolution timers had been applied.

4. PERFORMANCE

The use of Bossa moves scheduling operations from the kernel into a separate module, and thus can have an impact on the context switch time. We use the context switch benchmark `lat_ctx` to measure the impact, and find that it is negligible for real-sized applications. We then consider how Bossa can be used to improve the performance of a video player. All measures were taken using the version of Bossa based on Linux 2.4.18.



(a) Average context-switch overhead (cycles)



(b) Increase in the context-switch overhead when using Bossa. The dotted horizontal lines at 100% and 110% highlight the region in which the overhead of Bossa is below 10%

Figure 3: Comparison of the Bossa implementation of the Linux policy and the native Linux scheduler

Impact on the context-switch overhead. Performing a context switch involves electing a new process, saving the register state of the current process, and installing the register state of the elected process. The context-switch overhead also includes the cost of reloading cache and TLB entries as needed during the subsequent execution of the elected process. We measure the cost of these operations using the `lat_ctx` benchmark of the LMBench 2.0.4 benchmark suite.[2] This benchmark passes a token around a ring of processes, triggering a context switch at each step. Each process sums the elements in a local array of a given size to emulate a working set. Varying the size of the array affects the cache and TLB behavior. Figure 3 compares the performance of `lat_ctx` when using the Bossa implementation of the Linux policy to the performance of `lat_ctx` when using the standard Linux scheduler. Measures are grouped first by the array size (0-64KB) and then by the number of processes (2-96). Tests were performed in single-user mode.

When the overall memory usage (product of the number of processes and the memory usage per process) of `lat_ctx` is below 64KB, the cost of the scheduling policy plays a significant role in the context-switch overhead. Indeed, the use of Bossa increases the overhead by up to 39%, with the worst case being that of 2 processes that manipulate a 64KB array (Figure 3b). When the overall memory usage is above 64-128KB, however, the context-switch overhead increases significantly for both Linux and Bossa. In these cases, the use of Bossa increases the context-switch overhead by only 2-5% as compared to Linux (Figure 3b). While these experiments show some overhead for Bossa, `lat_ctx` represents a worst case, because its computation time is dominated by scheduling and because the memory sizes used are much smaller than those used by real applications running on a general-pupose system.

[2] `http://www.bitmover.com/lmbench/`

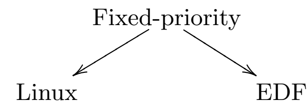


Figure 4: A scheduling hierarchy for use with MPEG video display

	Min	Max
Linux: Player only	0.005	0.048
EDF: Player only	0.019	0.024
Linux: Player, kernel compilation	0.009	22.683
EDF: Player, kernel compilation	0.017	0.018

Table 1: Distance between Video and Audio

MPEG video display. On a lightly loaded system, a video player can achieve the frame rate required by the video by sleeping for an appropriate time after processing each frame. On a heavily loaded system, the player needs to reserve a portion of CPU time within a fixed interval, to ensure both that it receives adequate access to the CPU and that it receives this access at the appropriate rate.

We consider the use of the video player `mplayer` with a scheduling hierarchy containing a Fixed-priority scheduler at the root, and the Linux 2.4 scheduler and the EDF scheduler of Section 2 at the leaves, as shown in Figure 4. The Linux 2.4 scheduler has lower priority than the EDF scheduler. All processes run on the Linux 2.4 scheduler, except the video player, which runs on the EDF scheduler. We slightly modified `mplayer` to dynamically construct the hierarchy, attach itself to the EDF scheduler, and yield at the end of the processing of each frame. Table 1 shows the behavior of the player using Bossa on the *Matrix Reloaded* trailer with and without reservations when competing with Linux kernel compilation. The behavior is represented as the difference in the percentage of the complete audio and video that has been treated so far. In both cases, we have given the X process a Linux real-time priority, so that when the player blocks to allow the video display, the X process runs immediately, thus reducing its impact as a performance bottleneck. Under the Linux 2.4 scheduling policy, the video falls far behind the audio in the presence of kernel compilation (third row of Table 1). With EDF, the player maintains correct synchronization.

5. CONCLUSION

In this paper, we have given an overview of the Bossa framework to facilitate the development of kernel schedulers. Our approach is based on a the use of a DSL that simplifies programming and allows critical properties to be verified at compile time.

We have demonstrated the expressiveness of our approach by implementing several well-known scheduling policies in Bossa. Our initial experience with the Bossa compiler has shown that it is useful in catching both common inattention errors and errors related to incorrect understanding of the target OS. Since integration of a policy into the kernel is handled by the compiler and the framework, it is easy to test new policy variants. Thus, scheduler programming is made accessible to non-kernel experts. Indeed, we have developed lab materials for teaching basic scheduling principles to un-

dergraduates using Bossa and have used these materials in graduate and undergraduate courses over the past few years. In our experience, the ease of use and robustness of the DSL has allowed students to freely experiment with scheduling at the OS kernel level without crashing the machine.

We are currently porting Bossa to the real-time OS Jaluna and are considering how to extend Bossa to multiprocessors. Finally, based on our success in using AOP to integrate the Bossa framework into an existing OS, we are currently studying how to use similar techniques to implement other kinds of OS evolutions [12].

Availability. Bossa and all material described in this paper are available at the Bossa web site:
`http://www.emn.fr/x-info/bossa/`.

6. REFERENCES

[1] R. A. Åberg, J. L. Lawall, M. Südholt, G. Muller, and A.-F. Le Meur. On the automatic evolution of an OS kernel using temporal logic and AOP. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, pages 196–204, Montreal, Canada, Oct. 2003. IEEE.

[2] A. Atlas and A. Bestavros. Design and implementation of statistical rate monotonic scheduling in KURT Linux. In *IEEE Real-Time Systems Symposium*, pages 272–276, Phoenix, AZ, Dec. 1999.

[3] J. L. Bruno, E. Gabber, B. Özden, and A. Silberschatz. Move-to-rear list scheduling: a new scheduling algorithm for providing QoS guarantees. In *Proceedings of ACM Multimedia*, pages 63–73, Seattle, WA, Nov. 1997.

[4] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Scheduling in Real-Time Systems*. Wiley, West Sussex, England, 2002.

[5] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, pages 261–276, Kiawah Island Resort, SC, Dec. 1999.

[6] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2000.

[7] K. Jeffay, F. D. Smith, A. Moorthy, and J. Anderson. Proportional share scheduling of operating system services for real-time applications. In *IEEE Real-Time Systems Symposium*, pages 480–491, Madrid, Spain, Dec. 1998.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97 – Object-Oriented Programming, 11th European Conference*, number 1241 in Lecture Notes in Computer Science, pages 220–242, Jyväskylä, Finland, June 1997.

[9] D. Lacey and O. de Moor. Imperative program transformation by rewriting. In R. Wilhelm, editor, *Compiler Construction, 10th International Conference, CC 2001*, number 2027 in Lecture Notes in Computer Science, pages 52–68, Genova, Italy, 2001.

[10] J. L. Lawall, A.-F. Le Meur, and G. Muller. On designing a target-independent DSL for safe OS process-scheduling components. In *Generative Programming and Component Engineering: Third International Conference, GPCE 2004*, number 3286 in Lecture Notes in Computer Science, pages 436–455, Vancouver, Canada, Oct. 2004.

[11] J. L. Lawall, G. Muller, and H. Duchesne. Language design for implementing process scheduling hierarchies (invited paper). In *ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation - PEPM'04*, pages 80–91, Verona, Italy, Aug. 2004.

[12] J. L. Lawall, G. Muller, and R. Urunuela. Tarantula: Killing driver bugs before they hatch. In *The 4th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, pages 13–18, Chicago, IL, Mar. 2005.

[13] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, Jan. 1973.

[14] J. R. Lorch and A. J. Smith. Scheduling techniques for reducing processor energy use in MacOS. *Wireless Networks*, 3(5):311–324, Oct. 1997.

[15] G. Muller, J. Lawall, J.-M. Menaud, and M. Südholt. Constructing component-based extension interfaces in legacy systems code. In *ACM SIGOPS European Workshop 2004 (EW2004)*, pages 80–85, Leuven, Belgium, Sept. 2004.

[16] G. Muller, J. L. Lawall, and H. Duchesne. A framework for simplifying the development of kernel schedulers: Design and performance evaluation. In *HASE 2005 - High Assurance Systems Engineering Conference*, Heidelberg, Germany, Oct. 2005. To appear.

[17] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In R. N. Horspool, editor, *Compiler Construction, 11th International Conference, CC 2002*, number 2304 in Lecture Notes in Computer Science, pages 213–228, Grenoble, France, Apr. 2002.

[18] J. Nieh and M. S. Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97)*, pages 184–197, Saint-Malo, France, Oct. 1997.

[19] J. Regehr and J. A. Stankovic. Augmented CPU reservations: towards predictable execution on general-purpose operating systems. In RTAS'2001 [20], pages 141–148.

[20] *Proceedings of the 7th Real-Time Technology and Applications Symposium (RTAS'2001)*, Taipei, Taiwan, May 2001.

[21] Y. Shin and K. Choi. Power conscious fixed priority scheduling for hard real-time systems. In *Proceedings of the 36th ACM/IEEE conference on Design Automation Conference (DAC'99)*, pages 134–139, New Orleans, LA, June 1999.

[22] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion

allocator for real-rate scheduling. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 145–158, New Orleans, LA, Feb. 1999.

[23] D. K. Y. Yau and S. S. Lam. Adaptive rate-controlled scheduling for multimedia applications. *IEEE/ACM Transactions on Networking*, 5(4):475–488, Aug. 1997.

[24] W. Yuan and K. Nahrstedt. Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, pages 149–163, Bolton Landing (Lake George), NY, Oct. 2003.

[25] W. Yuan, K. Nahrstedt, and K. Kim. R-EDF: A reservation-based EDF scheduling algorithm for multiple multimedia task classes. In RTAS'2001 [20], pages 149–156.