Bossa: A DSL Framework for Application-specific Scheduling Policies

Luciano Porto Barreto, Gilles Muller

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Bossa: A DSL Framework for Application-specific Scheduling Policies

Luciano Porto Barreto, Gilles Muller

N˚4191

Juin 2001

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4191--FR+ENG

Bossa: A DSL Framework for Application-specific Scheduling Policies

Luciano Porto Barreto, Gilles Muller

Thème 2 — Génie logiciel
et calcul symbolique

Projet COMPOSE

Rapport de recherche n ° 4191 — Juin 2001 — 15 pages

Abstract:

Developing or specializing existing process schedulers for new needs is tedious and error-prone due to the lack of modularity and inherent complexity of scheduling mechanisms. In this paper, we propose a framework based on a Domain-Specific Language for the implementation of scheduling policies. This framework permits the installation of basic scheduling policies, called Virtual Schedulers, and the development of Application-Specific Policies, which tailor a Virtual Scheduler to application-specific requirements. We illustrate our approach with concrete examples that show how specialization and reuse of scheduling policies can be accomplished while retaining OS robustness.

Key-words: Process Scheduling, Operating systems, Domain-Specific Languages

(Résumé : tsvp)

This research is partially supported by France Telecom R&D under the Phenix project.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Bossa: Une infrastructure basé sur les langages dédiés pour la conception des politiques d'ordonnancement

Résumé :

Développer (ou adapter) un ordonnanceur de processus pour répondre à un besoin particulier est une tâche ardue et sujette aux erreurs. Cela est dû à la complexité intrinsèque des mécanismes d'ordonnancement, et au manque de modularité des ordonnanceurs existants. Dans cet article, nous proposons une architecture logicielle pour l'implémentation de politiques d'ordonnancement. Cette architecture logicielle, qui repose sur un langage dédié à ce domaine, divise le problème en deux. D'une part, elle permet de choisir une politique d'ordonnancement de base (nommée *ordonnanceur virtuel*). D'autre part, elle permet de concevoir une politique spécifique à l'application, dont le rôle est de spécialiser l'ordonnanceur virtuel choisi. Nous illustrons cette approche par des exemples concrets, qui démontrent qu'il est possible de réutiliser et de spécialiser des politiques d'ordonnancement, sans pour autant compromettre la robustesse dans le processus de développement des systèmes d'exploitation.

Mots-clé : Ordonnancement de processus, Systèmes d'exploitation, Langages dédiés

1 Motivation

Emerging computing models and applications are continuously challenging the operating system scheduler. Multimedia applications require predictable performance and stringent timing guarantees [12, 16, 19, 36]. Embedded systems need to minimize power consumption [15, 29, 35]. Network routers demand isolated execution of active network programs [24]. Multiprocessor applications call for processor affinity allocation [33] or performance-driven allocation [7]. Meeting all these requirements requires specialized scheduling policies, which traditional scheduling infrastructures are unable to provide. First, most operating systems rely on a fixed and general-purpose process scheduling infrastructure, which is unable to satisfy the diverse CPU requirements of all applications [10, 11]. Second, OS schedulers provide an opaque interface to application developers (*e.g.,* priority values, period). Such an interface is unable to capture the increasing complexity of application requirements [13]. Finally, an application may alternate between several computing behaviors during its lifetime [23] (*e.g.,* interactive, CPU-bound, I/O bound). This unpredictability makes it difficult to build a clairvoyant scheduler that is able to identify behavioral changes and to adapt its operation accordingly. It is thus important that applications provide additional information so as to enhance the decision heuristics of the scheduler.

To overcome these problems, several research projects have introduced innovative scheduling infrastructures. These approaches provided specific scheduling properties as well as adaptive scheduling strategies [9, 20, 26]. For example, fair-share schedulers [11, 14] enable unbiased CPU distribution and cumulative service guarantees [4]. Hierarchical [10] and proportional-share schedulers [23, 34] enforce load insulation, which allows for firewall protection among different schedulers. Beyond properties, other proposals such as rate-controlled [36] and progress-based schedulers [8, 28] provide additional support for adaptation by allowing applications to regulate their CPU use through the observation of system resources and performance-driven metrics related to the application execution.

Nevertheless, it is difficult for application developers to isolate and specialize the scheduling policies provided by these infrastructures. First, existing infrastructures lack a clear separation between basic scheduling policies and application-specific policies. Furthermore, as some scheduling infrastructures implement sophisticated strategies, they generally trade extensibility for simplicity of use with regard to application developers.

While it is clear the need for customized scheduling policies, there is a lack of tools that capture the design singularities of schedulers to ease the development process. We believe that the task of fulfilling specific application requirements would be eased if a developer were able to choose from a collection of basic schedulers that could be specialized according to application-specific needs. The limitations in existing schedulers suggest the need for a software framework in which a developer can easily write, choose and customize a scheduler infrastructure.

This paper

In this paper we present the design of a framework for the development of adaptable process scheduling infrastructures. This framework permits the development and installation of basic scheduling policies, which can be specialized using application-specific policies.

Writing schedulers requires deep OS knowledge and involves development of low-level OS code (*e.g.,* clocks, timers, process queues), which frequently crosscuts multiple kernel mechanisms (*e.g.,* process synchronization, file system and device driver operations, system calls). To effectively implement a scheduler, developers must be aware of those scheduling-related sites in order to instrument the kernel accordingly. Consequently, developing or extending scheduling infrastructures with new functionalities is tedious and error-prone.

We base our approach on a Domain-Specific Language (DSL). A DSL is a high-level language that provides appropriate abstractions, which captures domain expertise and eases program development. Implementing an OS using a DSL improves OS robustness because code becomes more readable, maintainable and more amenable to verification of properties [18].

Our target is to specialize process schedulers for an application with soft-real time requirements that is able to specify adequate regulation strategies for its CPU requirements. We address neither multiprocessor scheduling nor distributed scheduling, although the framework can be extended for these purposes.

The contributions of this paper are as follows:

- We present the design of a modular framework architecture for the development of scheduling policies. This framework permits the implementation of basic schedulers, called Virtual Schedulers, and Application-Specific Policies, which tailor a virtual scheduler to application-specific needs.
- We describe how developers can write application-specific policies to adapt virtual schedulers using a DSL named Bossa. Bossa provides abstractions to operating systems internals, easing the development, maintenance and reuse of scheduling policies.
- We illustrate our approach using several realistic examples based on existing schedulers ([28] and [36]) that show how scheduling policies can be safely deployed in our framework.

This paper is organized as follows. Section 2 describes the execution model of our framework. Sections 3 and 4 present the framework components and describe concrete examples of scheduling policies written in Bossa. In Section 5, we describe how we ensure safe policy execution. Finally, Section 6 concludes the paper with on-going work.

2 Related work

Over the years research on process scheduling has been predominantly concentrated on engineering new mechanisms to fix the deficiencies of existing scheduler infrastructures on the behalf of certain applications. Although these works resulted in ingenious scheduling algorithms, only a few proposals aimed at reducing the implementation effort of schedulers. In this section we present some approaches that proposed new mechanisms for easing the construction of process schedulers.

SPIN [3] is an OS that be customized through the installation of dynamic kernel extensions called *Spindles*. Spindles are written in an enhanced version of Modula-3. By applying restrictions to Modula-3 and by performing static and dynamic verifications over Modula-3 code, SPIN achieves extensibility while guaranteeing that kernel extensions are safe. Although SPIN provides mechanisms for writing user-level schedulers (*i.e.,* thread packages) that are executed in kernel space, it is not possible to replace the global kernel scheduler. The kernel scheduler is by default a round-robin priority-based scheduler. SPIN provides a programming model for the development of general OS services, whereas our approach is more fine-grained as Bossa is especially tailored for writing process schedulers.

Candea and Jones designed Vassal [5], which is a loadable scheduler infrastructure for Windows NT. A Vassal scheduler is as NT driver written in C. The two basic primitives for writing a scheduler are `RegisterScheduler` and `SetSchedulerEvent`. `RegisterScheduler` is used to register the scheduler in the kernel and to inform the address of a function that must be called whenever a new process needs to be chosen. The `SetSchedulerEvent` provides a simple timeout mechanism for activating the scheduler at a specific time. Although these kernel primitives simplify the development of simple schedulers, they are too coarse-grained and limited to allow the construction of more elaborate scheduling policies. Another limitation is that Vassal only supports two simultaneous schedulers (in which one of the two schedulers must be the native NT scheduler). Also, schedulers are assumed to be *trusted* extensions, so there is no verification of properties or safety guarantees.

RTLinux [1] is a hard real-time Linux variant that primarily focuses on providing system timing predictability to applications. In RTLinux, real-time processes are implemented as loadable kernel modules, which have precedence over Linux processes. From the viewpoint of the RTLinux kernel Linux is considered as a low-priority process. Consequently, ordinary Linux processes (*e.g.,* X-windows, shells) are only executed when no real-time process is requesting the CPU. As real-time processes execute in kernel mode, they have access to kernel functions and data structures such as the process run queue. By carefully manipulating real-time priorities and the system run queue, it is possible to develop customized schedulers to control the execution of a set of processes or all processes in the system. One drawback is that undetected programming errors in a scheduler module are likely to crash the OS. Using the C language to develop RTLinux modules aggravates this problem, especially in terms of memory access errors due to eventual misuse of pointers.

Ford and Susarla [10] introduced a hierarchical scheduling framework in which a scheduler process donates its CPU time to other processes. To perform CPU donation, a scheduler invokes a special primitive called `schedule()`, which takes as parameters the process to be run and a condition that specifies when the scheduler (*i.e.,* donating process) should be woken. The kernel uses IPC messages to communicate with a process scheduler (*e.g.,* process creation). As in RTLinux and Vassal, writing schedulers requires meticulous programming since schedulers are written in C.

More recently, Regehr [25] developed an hierarchical loadable scheduler (HLS) architecture, in which it is possible to dynamically load schedulers in the kernel of Windows 2000. Schedulers are written in C using an specialized API that is intended to ease the development effort by hiding OS details. Likewise Vassal, loadable schedulers are considered as trusted entities and assumed to be correctly implemented. There are no verification mechanisms to help the programmer.

3 Execution model

Our framework architecture relies on two basic components: (i) Virtual Schedulers (VSs) and (ii) Application-Specific Policies (ASPs). We consider a process to be the minimal schedulable entity. Every process in the system is associated with a VS. During initialization, a process registers with a virtual scheduler either by joining an existing VS or by loading a new VS and joining it afterwards.

A virtual scheduler is a loadable kernel module that controls the execution of a set of processes. A VS manages processes by selecting a process for execution and determining its CPU quantum. To submit one of its processes for execution, a VS invokes a `dispatch` primitive, which causes a low-level context switching.

To permit the coexistence of multiple and independent schedulers, VSs can be stacked, forming a tree hierarchy as in [10, 11]. The topmost VS in the hierarchy is called Root VS. A VS is registered either with the Root VS or with another VS in the hierarchy. A VS registers with a VS (including the Root VS) in the same way that an ordinary process does.

An application developer specializes a VS by writing an Application-Specific Policy (ASP). An ASP defines a collection of event handlers that are triggered for a particular process. To allow an ASP to communicate with a VS, a virtual scheduler provides a set of functions that we call the *VS interface*.

Application-specific policies are Bossa programs that are evaluated by a Just-In-Time (JIT) compiler running in the kernel [30]. When a process registers with a VS, the JIT compiler translates the ASP specification into machine code. The execution of this machine code is controlled by the VS. The JIT compiler is also able to perform static and

dynamic verification on ASP code. Examples of verifications include checking bounds and type mismatches between the parameters passed by an ASP to a VS.

4 Virtual schedulers

A virtual scheduler (VS) is a kernel module that controls the execution of a collection of processes. Depending on its characteristics, a VS may implement diverse regulation functions such as process admission and overallocation control. These functions generally apply to all processes running under the supervision of a VS. Regulation functions are frequently used to ensure certain properties. For instance, a VS may ensure that no process deadlines are missed or that the CPU is fairly distributed among processes. To implement regulation functions, a VS maintains a local state information about each registered process.

A VS also provides an *interface*. This interface consists of a list of functions and events that can be used in the specification of an application-specific policy. An ASP uses interface functions to exchange data with a VS. These functions provide a local view of the process state stored by a VS. From the viewpoint of an ASP, the VS interface is the only visible part of a VS. A VS can export several interfaces. Also, only the ASP of registered processes have access to a VS interface.

In Figure 1 we describe one interface of a rate-based scheduler such as ARC [36]. The rate-based interface consists of four functions. These functions set or retrieve two process attributes maintained by the VS: `period` and `rate`. For this VS, `period` represents a time interval (in ms) and `rate` is a value between 0 and 1 that denotes a CPU time request within each period. These two values are used by the VS to compute the CPU quantum of a process.

```
VS rate-base-scheduler {
     interface {
        int setPeriod (int period);
        int getPeriod();
        int setRate (float rate);
        int getRate();
     }
};
```

Figure 1: A rate-based VS interface

5 Application-specific policies

An Application-Specific Policy (ASP) is a Bossa program that specializes the behavior of a VS with respect to specific application needs. One example of VS specialization using the rate-based interface of Figure 1 is to define an ASP for regulating the process rate by using the `setRate()` function. An ASP computes some values and uses the interface to communicate these values to the VS.

Bossa is based on an event-action model. We adopted an event-action model because ASP code is typically executed in response to process-related events (*e.g.,* process creation, blocking, unblocking, CPU quantum expiration). An ASP is organized into two parts: (i) a list of global variable declarations and (ii) a collection of event handlers. Event handlers are executed either periodically or in response to specific OS events. It is also possible to define ASP functions that can be used in the event handlers.

Bossa provides a language syntax *a la* C and it relies on two abstractions (i) *events* and (ii) *application-specific policy variables*. These abstractions are presented below.

5.1 Events

Bossa provides two classes of events for writing ASPs. These events are *temporal events* and *system events*. Note that the order in which event handlers are defined in the ASP do not impose the order in which they are evaluated.

Temporal events. A temporal event is used to group actions that need to be periodically executed for a process. It is typically used is to compute application-specific metrics that are periodically passed to a VS. As an example, a temporal event can be used by a process to perform a periodic regulation of its CPU requirements. We illustrate the latter approach by presenting one of the adaptive CPU allocation strategies proposed by Yau and Lam [36] for tuning the execution of a video player application (Figure 2). We assume that the video player process is bound to the rate-based VS defined in Figure 1.

The specialized policy, described in Figure 2, declares four global variables (`myRate`, `monInterval`, `cpuTimeInt` and `laxTolerance`), a temporal event handler and a function (`computeLaxity`). The temporal event handler code is executed at every two seconds. This handler computes a `laxity` value, which denotes the amount of unused CPU time of a process during the monitoring interval. The application-specific strategy is to decrease the process rate whenever the `laxity` value is greater than an upper bound, which is defined by `laxTolerance`. Note that to update `myRate`, we use `getRate()` to refetch the most recent rate value, because the VS may have changed this value. Next, the updated `myRate` value is passed to the VS by calling `setRate()`. By analyzing the ASP code, the Bossa compiler

statically verifies the type of `myRate` against the argument type defined by the `setRate()` function.

```
float myRate;
time monInterval 2s;
int cpuTimeInt = 0;
int laxTolerance = 10; // in %
every monInterval  {
    int laxity;
    cpuTimeInt = 0;
    laxity = computeLaxity();
    if (laxity > laxTolerance) {
       myRate = getRate();
       myRate = myRate - 0.1;
       setRate (myRate);
    }
 };
 int computeLaxity() {
     int lax;
     myRate = getRate();
     lax = 100*(1-cpuTimeInt/myRate*monInterval);
     return lax;
 };
```

Figure 2: Application-specific policy for the ARC scheduler

System events. A system event define actions to be executed when a specific OS event related to the process occurs. Examples of system events include I/O completion, semaphore unblocking and timer expiration. System events are particularly useful for inserting application-specific code that computes process metrics in response to OS events. One example is the computation of the laxity value described in Figure 2. Computing laxity for a given process requires maintaining a cumulative value of used CPU time (`cpuTimeInt`) during the monitoring interval. To obtain this value, we need to record the CPU time used by the process each time the process in question is preempted. We implement this adjustment by defining the system event handler of Figure 3. This event handler specification updates the value of `cpuTimeInt` each time a preemption event occurs. We assume here that `Preemption` is an event exported by the VS interface. The variable `usedCpuTime` is a *system variable* (defined in Section 5.2) that provides the cumulative CPU time of the last executed process.

```
On Preemption () {
  cpuTimeInt = cpuTimeInt + usedCpuTime;
};
```

Figure 3: Updating `cpuTimeInt` on process preemption

5.2 Application-specific policy variables

An application-specific policy can declare both global variables and variables local to each event handler. Besides those variables, Bossa provides two other classes of variables that can be used within an event handler. These variables are *system variables* and *process variables*.

System variables. System variables are read-only variables that provide information about certain OS data such as CPU tick counters, current time. One example is `usedCpuTime`, which was used in Figure 3.

Process variables. Process variables are read-only variables related to data residing in the address space of a given process. We can consider process variables as ASP aliases to variables used by the application process during its execution. A process variable typically serves as an indicator that represents a meaningful notion of application *progress* [8]. Process variables that are statically allocated are made visible to an ASP via stub functions. For the moment, we do not support functions to collect values of dynamically allocated variables.

To illustrate the use of process variables, we present an ASP proposed for controlling the CPU requirements of a video player application. This ASP was designed to be used with the OGI progress-based scheduler by Steere *et al* [28]. In our model, the OGI VS interface consists of two functions: `setPeriod(int period)` and `setProportion(int proportion)`. The `proportion` parameter denotes a fraction of the period, which is similar to the `rate` value defined for the VS interface in Figure 1, although it has a different range specification (in parts per thousand).

The ASP strategy is to define the progress of the video player in terms of the input and output levels of its shared memory queues. This progress metric, called queue pressure, is calculated by the `function computeQueuePressure()`, shown in Figure 4. This function is periodically invoked by a temporal event handler (omitted here). This event handler uses queue pressure values to compute a new process proportion. This updated proportion value is then passed to the VS by the `setProportion()` function.

To access shared memory queue objects in the `computeQueuePressure()` function, we first specify a data type called `smqueue`. This data type corresponds to the type of shared memory queues as defined in the application process. With this type information, the Bossa compiler is able to generate stub functions for collecting this data in the application

```
float function computeQueuePressure {
  float qpressure = 0; int delta = 0;
  foreach smqueue q do
     q = getSmqueue();
     delta = f(q.level, q.size);
     if (q.role == 'PRODUCER')
        qpressure = qpressure - delta;
     else
        qpressure = qpressure + delta;
  }
  return qpressure;
};
```

Figure 4: Queue pressure computation

process. As in the application, the `smqueue` type contains the level, size and role (producer or consumer) of a queue. Queue information is collected by the `getSmqueue()` function. In the `computeQueuePressure` function, queue pressure values are updated within a bounded `foreach` loop. This loop traverses all process objects of type `smqueue`.

For efficiency reasons, during the execution of an ASP it is not possible to collect process variables when the involved process is not mapped in memory. Our approach is to save a copy of the process variable in kernel space each time the process is preempted. Therefore, at a given instant, a process variable contains the value of the process variable at the last context switch time for the involved process.

6 Robustness

DSLs have been successful in improving safety in the development of device drivers [17, 32], memory coherence protocols [6], packet filters [2] and active network applications [31]. As we plan to execute VSs and ASPs in the kernel, we need to guarantee that such inserted code does not jeopardize OS integrity.

We ensure policy robustness either by construction or by verification. The first approach consists of restricting the language such that DSL programs are safe. The second approach analyses policy code to verify properties. In our framework, we guarantee that Bossa code does not perform harmful modification to the system data structures and code. We achieve this by suppressing pointers in the language and by enforcing a read-only semantics of process and system variables. Also, the DSL compiler performs verification on ASP code, such as checking bounds and data type mismatches of parameters passed to a VS interface function. Moreover, ASPs are not allowed to have unbounded loops and recursive functions. This constraint ensures that event handlers terminate. Excessive overheads eventually caused by

heavyweight ASPs can be prevented in several ways: by establishing a fixed quota for event handling evaluation, by defining a minimum interval for temporal events, or by penalizing each process for the execution of its own event handlers.

7 Implementation

Two fundamental goals we have defined for Bossa is to achieve both flexibility and good performance. Flexibility is needed to allow VSs and ASPs to be dynamically loaded and executed in the kernel. This requirement makes it unsuitable to use an static compilation model. Otherwise, employing a conventional interpretation scheme is neither appropriate as it would introduce severe performance penalties. It is important to have in mind that scheduling code in modern OSes is likely to be executed hundreds of times per second, so an inefficient scheduler implementation can easily degrade overall systems performance. Our solution to provide flexibility while retaining performance is to implement a JIT compiler that runs in the kernel. The JIT compiler translates ASP code into machine code, which is then executed in the kernel. This approach was particularly inspired by our previous experience with PLAN-P [31], which is a DSL for the development of active network policies. Likewise PLAN-P, we believe that an in-kernel JIT compiler is also appropriate to provide a flexible and efficient implementation for Bossa.

8 Conclusions and on-going work

We have presented the design of a framework for the development of application-specific scheduling infrastructures. We are currently implementing Bossa in Linux. Our next work is to extend the Bossa for writing virtual schedulers and to investigate appropriate abstractions and properties for the composition of scheduling hierarchies. We plan to assess our framework by conducting experiments on scheduling infrastructures using real workloads. We will evaluate Bossa by analyzing the behavior of soft real-time applications, such as video players.

Acknowledgments

We thank Julia Lawal for her helpful comments on earlier versions of this paper.

References

[1] Michael Barabanov. A Linux-based realtime operating system. Master's thesis, New Mexico Institute of Mining and Technology, June 1997.

[2] A. Begel, S. McCanne, and S. Graham. BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture. In *Proceedings of ACM SIGCOMM'99*, pages 123–134, September 1999.

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, December 1995.

[4] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. Move-to-rear list scheduling: a new scheduling algorithm for providing QoS guarantees. In *Proceedings of ACM Multimedia, Seattle, Washington*, November 1997.

[5] George M. Candea and Michael B. Jones. Vassal: Loadable scheduler support for multi-policy scheduling. In *Proceedings of the 2nd USENIX Windows NT Symposium*, pages 157–166, Berkeley, August 3–5 1998.

[6] Satish Chandra, Bradley Richards, and James R. Larus. Teapot: A domain-specific language for writing cache coherence protocols. *IEEE Transactions on Software Engineering*, 25(3):317–333, May/June 1999.

[7] J. Corbalán, X. Martorell, and J. Labarta. Performance-driven processor allocation. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, San Diego, California, October 2000.

[8] J. Douceur and W. Bolosky. Progress-based regulation of low-importance processes. In SOSP99 [27], pages 247–258.

[9] K. Duda and D. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In SOSP99 [27], pages 261–276.

[10] Bryan Ford and Sai Susarla. CPU Inheritance Scheduling. In OSDI'96 [22], pages 91–105.

[11] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In OSDI'96 [22], pages 107–121.

[12] Chih han Lin, Hao hua Chu, and Klara Nahrstedt. A soft real-time scheduling server on the Windows NT. In *Proceedings of the 2nd USENIX Windows NT Symposium*, pages 149–156, Berkeley, August 3–5 1998.

[13] Joseph L. Hellerstein. Achieving Service Rate Objectives with Decay Usage Scheduling. *IEEE Transactions on Software Engineering*, 19(8):813–825, August 1993.

[14] J. Kay and P. Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, January 1988.

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

[16] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: operating system support for multimedia applications. In *Proceedings of the International Conference on Multimedia Computing and Systems, Boston, MA, USA, May 15–19, 1994*, pages 90–99, 1994.

[17] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for Hardware Programming. In *4th Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 17–30. USENIX Association, October 2000.

[18] G. Muller, C. Consel, R. Marlet, L.P. Barreto, F. Mérillon, and L. Réveillère. Towards robust oses for appliances: A new approach based on domain-specific languages. In *Proceedings of the ACM SIGOPS European Workshop 2000 (EW2000)*, pages 19–24, Kolding, Denmark, September 2000.

[19] J. Nieh, J. Hanko, J. Northcutt, and G. Wall. Svr4 unix scheduler unacceptable for multimedia applications. In *Proceedings of the 4th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 35–47, November 1993.

[20] Jason Nieh and Monica S. Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proceedings of the 1997 ACM Symposium on Operating Systems Principles*, pages 184–197, St-Malo, France, October 1997.

[21] *Proceedings of the First Symposium on Operating Systems Design and Implementation*, 1994.

[22] *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, Washington, October 1996.

[23] David Petrou, John W. Milford, and Garth A. Gibson. Implementing lottery scheduling: Matching the specializations in traditional schedulers. In *Proceedings of the USENIX Annual Technical Conference (USENIX-99)*, pages 1–14, Berkeley, CA, June 6–11 1999.

[24] Xiaohu Qie, Andy Bavier, Larry Peterson, and Scott Karlin. Scheduling Computations on a Software-Based Router. In *Proceedings of SIGMETRICS'2001*, June 2001. To appear.

[25] John Regehr. *Using Hierarchical Scheduling to Support Soft Real-Time Applications on General-Purpose Operating Systems*. PhD thesis, University of Virginia, 2001.

[26] John Regehr and John A. Stankovic. Augmented CPU Reservations: Towards Predictable Execution on General-Purpose Operating Systems. In *Proceedings of the 7th Real-Time Technology and Applications Symposium (RTAS 2001)*, Taipei, Taiwan, May 2001.

[27] *Proceedings of the 1999 ACM Symposium on Operating Systems Principles*, Kiawah Island, SC, December 1999.

[28] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the 3rd Symposium on Operating Systans Design and Implementation (OSDI-99)*, pages 145–158, February 1999.

[29] Vishnu Swaminathan and Krishnendu Chakrabarty. Real-time task scheduling for energy-aware embedded systems. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS) – Work-in-progress session, November, 2000, Orlando, FL,USA*, 2000.

[30] S. Thibault, C. Consel, J.L. Lawall, R. Marlet, and G. Muller. Static and dynamic program compilation by interpreter specialization. *Higher-Order and Symbolic Computation*, 13(3):161–178, 2000.

[31] S. Thibault, C. Consel, and G. Muller. Safe and efficient active network programming. In *17th IEEE Symposium on Reliable Distributed Systems*, pages 135–143, West Lafayette, Indiana, October 1998.

[32] S. Thibault, R. Marlet, and C. Consel. Domain-specific languages: from design to implementation – application to video device drivers generation. *IEEE Transactions on Software Engineering*, 25(3):363–377, May–June 1999.

[33] Raj Vaswani and John Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 26–40, October 1991.

[34] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In OSDI'94 [21], pages 1–11.

[35] Mark Weiser, Alan Demers, Brent Welch, and Scott Shenker. Scheduling for reduced CPU energy. In OSDI'94 [21], pages 13–23.

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

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399