A Framework for Simplifying the Development of Kernel Schedulers: Design and Performance Evaluation

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

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

Hervé Duchesne
Obasco Group, EMN-INRIA, LINA
44307 Nantes Cedex 3, France
herve.duchesne@emn.fr

Abstract

Writing a new scheduler and integrating it into an existing OS is a daunting task, requiring the understanding of multiple low-level kernel mechanisms. Indeed, implementing a new scheduler is outside the expertise of application programmers, even they are the ones who understand best the scheduling needs of their applications.

To address these problems, we present the design of Bossa, a language targeted toward the development of scheduling policies. Bossa provides high-level abstractions that are specific to the domain of scheduling. These constructs simplify the task of specifying a new scheduling policy and facilitate the static verification of critical safety properties.

We illustrate our approach by presenting an implementation of the EDF scheduling policy. The overhead of Bossa is acceptable. Overall, we have found that Bossa simplifies scheduler development to the point that kernel expertise is not required to add a new scheduler to an existing kernel.

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 give rise to a host of new scheduling algorithms [3, 8, 10, 11, 12, 17, 18, 24, 27, 28, 31, 35, 37, 38, 39, 40]. 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.

Our approach We propose a framework, Bossa, 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 kernel. In this context, Bossa has been used to implement a variety of scheduling policies, including policies directed towards multimedia applications such as progress-based scheduling [37], policies directed towards real-time systems such as rate monotonic and earliest-deadline first (EDF), 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.

The contributions of this paper are as follows:

- We present the Bossa DSL. This language simplifies the programming of scheduling policies. The Bossa compiler checks scheduling-specific properties, ensuring safety as the scheduling policy evolves.
- We assess the expressiveness of Bossa in terms of the number of lines of code required to implement a variety of multimedia and real-time scheduling policies.
- Using the `lat_ctx` context-switch latency benchmark of the LMbench benchmark suite,[1] we show that the context-switch overhead introduced by Bossa is acceptable for any real-sized process.
- We find no observable overhead from the use of Bossa with a variety of multimedia applications such as video display using `mplayer`.[2]
- We illustrate the value of using an application-specific scheduling policy in the context of video display. By using an EDF scheduling policy, the player maintains the proper frame rate, which is it unable to do when running under the ordinary Linux scheduling policy.

The rest of the paper is organized as follows. Section 2 examines the difficulties that arise when implementing a scheduler in an existing OS. Section 3 introduces the Bossa DSL. Section 4 evaluates the performance of our approach. Section 5 illustrates some applications of Bossa. Section 6 presents related work. Section 7 concludes and describes future work.

2 Analysis of Scheduler Development

To motivate the design of Bossa, we consider the issues that a programmer, in particular a programmer who is not a kernel expert, is faced with when implementing a scheduler in a standard OS kernel, Linux 2.4. We then describe how Bossa addresses these issues.

2.1 Kernel interface in Linux

To structure the implementation of a scheduling policy, the scheduler programmer must first identify the set of operations that the policy should provide. These include electing one of a set of eligible processes and reacting to state changes that affect the eligibility of processes, such as process unblocking and the passage of time. The exact set of operations depends on the needs of the policy and the behavior of the kernel. For example, Linux implements several variants of unblocking. The scheduler programmer must assess whether the differences between these variants are relevant to a given policy.

Once the operations have been identified, the scheduler programmer must determine the point in the kernel code at which each operation should be invoked. Typically, each operation defined by the policy should be invoked within the code implementing the corresponding operation in the kernel. Nevertheless, identifying the exact position and means of invocation can require substantial kernel expertise. In many cases, the Linux kernel implementation of a scheduling operation is spread out across many functions, as shown in Table 1. For example, unblocking in Linux involves a sequence of up to 7 macro and function calls in two different files. In all, the Linux implementation of scheduling-related operations involves 1340 lines of code, in 50 functions. Because the operations provided by the scheduling policy may subsume all or part of the Linux implementation, the scheduler programmer must understand the full impact of each statement in the original code to determine which statements should be modified or removed. Properties of locks and interrupts may need to be taken into account, which may require analysis of the entire kernel, not just the parts related to scheduling.

To be able to implement the various scheduling operations in a manner consistent with the kernel, the scheduler programmer must be aware of the relevant kernel state when each policy operation is invoked and of the kernel's expectations as to the effect of the operation. As an example, we consider unblocking. Intuitively, when the policy's unblocking code is invoked, the unblocking process should currently be blocked. In Linux, however, blocking of an executing process is not an atomic operation, and thus the treatment of unblocking must take into account the possibility that the affected process is still executing. Kernel expectations regarding the effect of each policy operation include properties of process states, flag values, and return values. For example, the unblocking code must leave the target process either executing, if it was executing at the beginning

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

	Lines of code	Functions & Macros
Process creation	216	4
Process termination		
code executed		
by the terminating process	149	3
code executed by the parent	143	2
Blocking	168	3
Unblocking	244	20
Clocktick		
interrupt handler	148	7
interrupt bottom half	24	2
Process election	248	9

Figure 1. Dispersal of scheduling-related code in the Linux 2.4.18 kernel

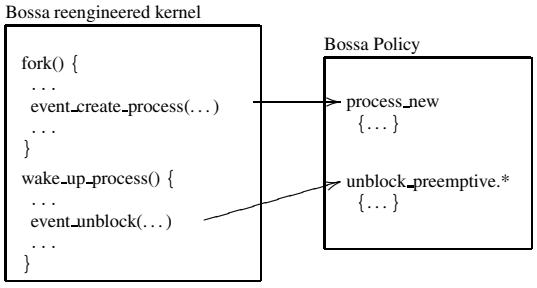


Figure 2. The Bossa event-based interface

of the unblock operation, or eligible for election. The return value of this code must indicate whether a process was actually unblocked. Fully understanding these constraints requires examining code scattered throughout the kernel.

2.2 Kernel interface in Bossa

Bossa encapsulates the points of interaction between a scheduling policy and an OS in an event-based interface, as illustrated in Figure 2. Because each OS has different scheduling-related behavior, this interface is specific to each target OS and is designed by an OS expert (*i.e.*, an expert in the given OS). This expert identifies the set of relevant scheduling events and re-engineers the kernel by replacing scheduling-related code by event notifications. Typical events include process creation and termination, blocking and unblocking, and the need to elect a new process. The OS expert also formally describes the expected behavior of the scheduler handler for each event. This description is provided to the scheduler programmer, who uses it to guide the development of a scheduler, and to the Bossa compiler, which checks that a scheduler satisfies OS-specific requirements and generates code that is compatible with the OS kernel.

The Bossa interface for the Linux kernel contains event notifications for the above events and for process yielding and the passage of time. In all, there are 11 basic events for which a Bossa scheduling policy must define handlers for use with the Linux kernel. In the Bossa framework, event notifications that are generated by multiple kernel services include information about the identity of the service that triggered the event. Such events are organized into a hierarchy, allowing an event handler to treat all instances of an event or only instances generated by a particular source. The Bossa interface for the Linux kernel distinguishes between the sources of blocking, unblocking and yielding events, distinguishing between services such as character devices, SCSI devices, and the network.

Because the size of the source code of a modern kernel, such as Linux, is typically over 100MB, we have developed an automated tool based on the CIL framework [25] to help the OS expert with the reengineering process [2]. The OS expert configures this tool with rewrite rules that describe patterns of kernel code and the associated re-engineering. Rules may use temporal logic to describe control-flow paths, allowing them to describe regions of code without relying on line numbers or inessential details of the region contents, as found in patch-based re-engineering approaches. We have found that the same set of rules can be used for the re-engineering of Linux 2.4.18 and Linux 2.4.24. For Linux, 23 rules are used, allowing the tool to automatically perform about 90% of the required re-engineering. Hand-modifications are needed for some macros and for non-C code, which are not adequately treated by CIL.

3 The Bossa DSL

The goal of the Bossa DSL is to express scheduling policies in a clear, concise and verifiable way. When using a general-purpose language such as C, the scheduler programmer must typically implement and keep consistent complex operations on processes such as determining the state of a process, identifying the set of processes in a given state, and process comparison. The programmer must also check by hand that the program is error-free, including both programming errors such as dangling pointers and errors related to incorrect interaction with the target OS. The Bossa DSL provides high-level abstractions for defining the various data structures and operations needed by a scheduling policy, from which the Bossa compiler generates automatically the appropriate implementations. The Bossa compiler also uses the semantics of the high-level abstractions combined with the model of kernel behavior provided by the OS expert to verify the correctness of a scheduling policy. We now give an overview of the Bossa DSL. The verification process has been described elsewhere [20].

A Bossa scheduling policy has three parts: declarations,

a set of handlers for kernel scheduling events, and an application interface. We introduce the language using excerpts of an implementation of an EDF scheduling policy [22], which illustrates most of the language features. The complete implementation is 162 lines of Bossa code. The complete EDF policy and a grammar of the Bossa DSL are available at the Bossa web site.[3]

Declarations The declarations of a scheduling policy define the process attributes, the scheduling states, and the ordering of processes. Those of the EDF policy are shown below.

```
process = {
  time  period;
  time  wcet;
  time  current_deadline;
  timer period_timer;
}

states = {
  RUNNING  running : process;
  READY    ready   : select sorted queue;
  READY    yield   : process;
  BLOCKED  blocked : queue;
  BLOCKED  computation_ended : queue;
  TERMINATED terminated;
}

ordering_criteria = { lowest current_deadline }
```

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. Process attributes are used to store information associated with a process across multiple invocations of the policy and may be used in the strategy for comparing processes.

The `states` declaration lists the set of process states that are distinguished by the policy. A process managed by the policy is always in exactly one of these states. Each state is associated with a state class (`RUNNING`, `READY`, `BLOCKED`, or `TERMINATED`) describing the schedulability of processes in the state. These state classes allow the Bossa compiler to check that the state changes performed by a scheduling policy reflect the changes in process schedulability resulting from the actions of the kernel. Each state is also associated with an implementation, either a process variable (`process`) or a queue (`queue`), which the compiler can sometimes optimize into an array. All operations on states are independent of the state class and implementation. The compiler automatically generates appropriate low-level code.

In the EDF policy, the state `running` is in the `RUNNING` class, and represents the currently running process. This state is implemented as a process variable, because Bossa currently targets uniprocessors.[4] The states `ready` and `yield` are in the `READY` state class, meaning that processes in these states are able to run. The `ready` state is designated as `select`, meaning that a new process can only be elected from this state. A process that has voluntarily yielded the processor is in the `yield` state. The states `blocked` and `computation_ended` are in the `BLOCKED` state class, meaning that processes in these states are not able to run. A process in the `blocked` state is waiting for a resource, while a process in the `computation_ended` state has completed its computation for the current period. Finally, the state `terminated` is in the `TERMINATED` state class, meaning that processes in this state are terminating. No data structure is associated with this state, because such processes are no longer relevant to the scheduler.

The `ordering_criteria` allows the comparison of two processes according to a sequence of criteria based on the values of their attributes. All process comparison operations are derived from this declaration. Higher or lower values of an attribute are favored using the keywords `highest` and `lowest`, respectively. The EDF policy favors the process with the lowest current deadline. The annotation `sorted` in the declaration of the `ready` state indicates that the associated queue is sorted according to this criterion.

Event handlers An event handler begins with the name of one or more handled events. A policy must define a handler for every event, although a wild card `*` can be used to specify a single handler for a collection of related events. The EDF policy defines 11 handlers. Handlers are parameterized by an event structure, `e`, that includes the *target process*, `e.target`, affected by the event, if there is one.

The event-handler syntax is based on that of a subset of C, to make the language easy to learn. The syntax 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.

The EDF policy defines several event handlers that react to process state changes. An example is the `unblock.preemptive.*` handler, below, which uses many of the domain-specific constructs. The handler first checks whether the target process is blocked. If so, it preempts

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

[4] Extension of Bossa to multiprocessors is in progress.

the running process if the target process has a higher priority than the running process, as determined by the ordering_criteria, and changes the state of the target process to ready, making the process eligible for election.

```
On unblock.preemptive.* {
  if (e.target in blocked) {
    if (!empty(running) && e.target > running) {
      running => ready;
    }
    e.target => ready;
  }
}
```

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. The EDF bossa.schedule handler is shown below. The main effect of this handler 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.

```
On bossa.schedule {
  if (empty(ready)) {
    yield => ready;
  }
  select() => running;
  if (!empty(yield)) {
    yield => ready;
  }
}
```

The structure of the event handlers shown here is quite simple, and is typical of that of most of the handlers found in Bossa scheduling policies. This simplicity, combined with the domain-specific operators and the characterization of process states by state classes, enables the Bossa DSL compiler to automatically verify that an event handler satisfies the description of OS-specific requirements provided by the OS expert.

Application interface A Bossa scheduling policy provides an interface to allow user processes to interact with the policy. The interface must at least provide the functions attach and detach, which allow a process to join and leave the scheduler, respectively. The EDF policy also defines a function sched_yield, shown below, that allows an EDF process p to indicate that it has ended its computation within the current period.

```
void sched_yield(process p) {
  p => computation_ended;
}
```

Assessment As compared to a general purpose language, the Bossa DSL is constrained in some ways, to protect against common fatal errors. Some examples are as follows. The language does not permit a reference to a process to be stored in a process attribute or global variable. This protects against dangling references that can crash the system or cause unpredictable behavior after a process has terminated. The language does not permit taking the address of any object, to ensure that data structures are accessed in a controlled way. Finally, the language does not provide for complex data structures other than the built-in structures associated with process states. There are no recursive function calls, and the only available looping construct is iteration over the process variables and queues associated with process states. These restrictions ensure termination.

Bossa supports both the construction of a single process scheduler, as described above, and the construction of a hierarchy of schedulers [29, 32]. 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 [21]. The use of a hierarchy allows multiple process schedulers, each satisfying particular scheduling needs, to coexist in a running OS.

4 Evaluation

We evaluate Bossa from the point of view of the expressiveness of the DSL and from the point of view of performance. Performance experiments were conducted on a 1600MHz Pentium 4 with an 8 KB L1 data cache, a 12 KB L1 instruction cache, a 256 KB L2 cache, 256 MB of RAM, and one 120 GB 7200 RPM Western Digital IDE drive with 8 MB of buffering.

4.1 Expressiveness of the DSL

We are currently using the Bossa DSL to develop a library of multimedia and real-time scheduling policies. Policies in the library are guaranteed to satisfy the safety properties checked by the Bossa verifier and are easy to modify to create new policy variants. Table 1 shows the code size of the Linux 2.4 scheduling policy when implemented in Bossa as well as the code size of various real-time and multimedia scheduling policies. Within a family of scheduling policies, there are substantial opportunities for code reuse. For example, both the EDF and RM policies target periodic

Process schedulers	Lines of code
The Linux 2.4 scheduling policy	201
Earliest-Deadline First (EDF)	162
Rate Monotonic (RM)	154
Deadline Monotonic	159
Least Laxity First	168
RM + polling server	262
Best [5]	160
BVT [12]	238
Progress-based scheduling [37]	234
Virtual schedulers	
Fixed priority	94
Proportional scheduling	102

Table 1. Code size of some Bossa schedulers

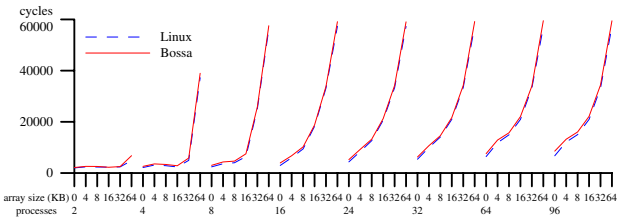
processes. These policies share 136 lines of code out of 162 and 154 lines of code, respectively.

4.2 Impact on the context-switch overhead

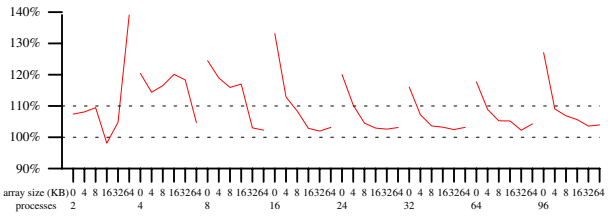
Performing a context switch involves electing a new process, saving the register state of the current process, and installing the register state of the elected process. The context-switch overhead also includes the cost of reloading cache and TLB entries as needed during the subsequent execution of the elected process. We measure the cost of these operations using the `lat_ctx` benchmark from the LMBench 2.0.4 benchmark suite.[5] 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 number of processes (2-96). Tests were performed in single-user mode.

Below the point where the overall memory usage (product of the number of processes and the memory usage per process) of `lat_ctx` is 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).

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



(a) Average context-switch overhead (cycles)



(b) Increase in the context-switch overhead when using Bossa

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

4.3 Impact on multimedia applications

The impact of any scheduling overhead is determined by the frequency of context switches. We thus analyze the scheduling behavior of the widely used MPEG video player application `mplayer`. A video player has periodic behavior, determined by the frame rate. Intuitively, in each period, the player initially blocks to receive the next frame of video data, then performs various computations to decode the frame, and finally blocks to wait for the beginning of the next period. Playing the *Matrix Reloaded* trailer[6] using `mplayer`, we observe that 60% of its time slices have a duration of under 100 cycles. Nevertheless, 15% of its time slices have a duration of over 2 million cycles, implying that overall the behavior of the player is dominated by computation rather than context switching. Thus, any overhead of Bossa should have no noticeable effect on the overall execution time. Indeed, we find that the use of Bossa with the Linux scheduling policy gives the same performance as the original Linux scheduler.

5 Playing Bossa

The goal of Bossa is to allow application programmers to easily develop and deploy schedulers that meet specific needs. We consider two such uses of Bossa: improving the scheduling of a video player and teaching real-time scheduling principles.

[6] The video size is 10.2 MB, its resolution is 1280 x 1024, and it lasts 151 seconds.

	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 2. 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 consisting of a Fixed-priority scheduler at the root, and the Linux 2.4 scheduler and the EDF scheduler of Section 3 at the leaves. 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 2 shows the performance of the player using Bossa on the *Matrix Reloaded* trailer with and without reservations when competing with Linux kernel compilation. The performance is measured as the difference in the percentage of the complete audio and video that has been treated so far. In both cases, we have raised the priority of the X process to a Linux real-time priority, so that when the player blocks to allow the video display, the X process runs immediately, thus reducing the impact of X 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 2). With EDF, the player maintains correct synchronization.

Teaching real-time scheduling Bossa is an ideal tool for teaching scheduling. In this setting, the main goals are to provide the student with a realistic and comparative understanding of various scheduling algorithms, without discouraging the student with tedious programming details, reboots, and kernel debugging. We have used Bossa in teaching at the undergraduate level. In these courses, we have observed that the assistance provided by the DSL and the associated verifications allows students with little or no kernel expertise to implement several classical scheduling policies for use at the kernel level in a few hours. The student is presented with a scheduler implemented as a hierarchy with a proportional scheduler at the root and the Bossa implementation of the standard Linux scheduler as the only child. To test a new process scheduler, the student adds it dynamically under the proportional scheduler, without rebooting the kernel. The proportional scheduler reserves a small amount of time for the standard Linux scheduler, allowing the student to test policies while retaining control over the system.

To fully understand a scheduling policy, the student must observe its dynamic behavior over time. We have modified the scheduler profiling tool Hourglass [30] to allow the processes it manages to be attached to Bossa schedulers. The use of this tool allows the student to observe the frequency of context switches and the effect of the process election strategy of a given policy.

6 Related work

Our work is related both to research on scheduler development and to work on improving OS development.

Scheduler development Although most research on process scheduling has focused on the discovery of new algorithms, a few approaches have addressed the difficulty of scheduler development.

The work closest to ours is the HLS framework for the development of scheduling hierarchies [1, 29]. HLS provides a library of schedulers that can be combined to achieve various effects and an API for implementing new schedulers. Nevertheless, such schedulers remain ordinary C code, that does not distinguish basic scheduling structures from policy-specific code. As compared to Bossa, scheduler implementations are thus less readable and less amenable to static verification, which has not been considered in HLS. Part of the work on HLS has focused on formally specifying the effect of composing schedulers [32]. We are considering how to incorporate this approach into Bossa.

Other work on scheduler development includes that of Ford and Susarla in which a process can donate its CPU time to other processes [15], Vassal which allows a new scheduler to be dynamically loaded into the Windows NT kernel [9], the S.Ha.R.K. kernel that is specifically designed to facilitate the implementation of new scheduling algorithms [16], and the MaRTE OS that exports a set of scheduling events [33]. None of these approaches provides verification of a scheduler implementation, so schedulers have to be assumed to be correct, although scheduling code remains low-level and error-prone.

Safety of OS code Recently, there has been much interest in compile-time error detection in the context of OS code. CCured [26], Cyclone [19] and Splint [14] check C programs for common programming errors, such as invalid pointer references. These approaches provide little or no support for checking domain-specific properties. Meta-level Compilation [13] checks properties that can be de-

scribed as a sequence of matching operations, such as locking and unlocking, and has been successfully applied to OS code. SLAM uses model checking to check similar properties [4]. These approaches work well when the programmer follows certain coding conventions (*e.g.* using kernel macros to change the interrupt level rather than using assembly code). A DSL, on the other hand, restricts the programmer to a limited set of abstractions, thus enabling more precise verifications.

High-level languages have been used in other systems projects to facilitate verification and optimization, including Modula-3 in SPIN [6], OCaml in Ensemble [23], and Standard ML in FoxNet [7]. As compared to these approaches, the use of domain-specific tools further targets verification and optimization to specific domain needs.

7 Conclusion and future work

In this paper, we have presented a complete framework to facilitate the development of kernel schedulers. Our approach is based on 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. The model of scheduling behavior provided by the OS expert presents relevant information in a concise form, rather than the large number of functions that must be studied to understand Linux scheduling behavior from the source code. Thus, scheduler programming is made accessible to non-kernel experts.

The current design of Bossa has some limitations that we want to address in the near future.

- We are planning to port Bossa to other OSes, such as Windows NT and real-time versions of Linux. We are also considering how to extend Bossa to multiprocessors.

- Process execution may be controlled by the availability of other resources than the CPU, such as the availability of disk, network, and energy resources. We plan to extend Bossa to allow control of these features to be incorporated in a scheduling policy.

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

References

[1] L. Abeni and J. Regehr. How to rapidly prototype a real-time scheduler. In *Work in Progress session of the 23rd IEEE Real-Time Systems Symposium (RTSS 2002)*, Austin, TX, Dec. 2002.

[2] 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.

[3] 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.

[4] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In M. Dwyer, editor, *8th International SPIN Workshop*, number 2057 in Lecture Notes in Computer Science, Toronto, Canada, 2001.

[5] S. A. Banachowski and S. A. Brandt. The BEST scheduler for integrated processing of best-effort and soft real-time processes. In *Multimedia Computing and Networking (MMCN)*, volume 4673, San Jose, CA, Jan. 2002.

[6] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. J. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 267–284, Copper Mountain Resort, CO, Dec. 1995.

[7] E. Biagioni, R. Harper, and P. Lee. A network protocol stack in standard ML. *Higher-Order and Symbolic Computation*, 14(4):309–356, Dec. 2001.

[8] 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.

[9] G. M. Candea and M. B. Jones. Vassal: Loadable scheduler support for multi-policy scheduling. In *Proceedings of the 2nd USENIX Windows NT Symposium*, pages 157–166, Seattle, WA, Aug. 1998.

[10] A. Chandra, M. Adler, and P. Shenoy. Deadline fair scheduling: Bridging the theory and practice of proportionate fair scheduling in multiprocessor systems. In RTAS'2001 [34], pages 3–14.

[11] H.-H. Chu and K. Nahrstedt. CPU service classes for multimedia applications. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems (ICMCS'99)*, pages 296–301, Florence, Italy, June 1999.

[12] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In SOSP'99 [36], pages 261–276.

[13] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, San Diego, CA, Oct. 2000.

[14] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, Jan./Feb. 2002.

[15] B. Ford and S. Susarla. CPU inheritance scheduling. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 91–105, Seattle, WA, Oct. 1996.

[16] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo. A new kernel approach for modular real-time systems development. In *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems*, June 2001.

[17] K. Jeffay and G. Lamastra. A comparative study of the realization of rate-based computing services in general purpose operating systems. In *Proceedings of the Seventh IEEE International Conference on Real-Time Computing Systems and Applications*, pages 81–90, Cheju Island, South Korea, Dec. 2000.

[18] 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.

[19] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 275–288, Monterey, CA, June 2002.

[20] J. L. Lawall, A.-F. Le Meur, and G. Muller. On designing a target-independent DSL for safe OS process-scheduling components. In *Third International Conference on Generative Programming and Component Engineering (GPCE'04)*, number 3286 in Lecture Notes in Computer Science, pages 436–455, Vancouver, Canada, Oct. 2004.

[21] 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.

[22] 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.

[23] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable. Building reliable, high-performance communication systems from components. In SOSP'99 [36].

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

[25] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction, 11th International Conference, CC 2002*, pages 213–228, Grenoble, France, Apr. 2002.

[26] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139, Portland, OR, Jan. 2002.

[27] 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.

[28] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, pages 89–102, Banff, Canada, Oct. 2001.

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

[30] J. Regehr. Inferring scheduling behavior with hourglass. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, pages 143–156, Monterey, CA, June 2002.

[31] J. Regehr and J. A. Stankovic. Augmented CPU reservations: towards predictable execution on general-purpose operating systems. In RTAS'2001 [34], pages 141–148.

[32] J. Regehr and J. A. Stankovic. HLS: A framework for composing soft real-time schedulers. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, pages 3–14, London, UK, Dec. 2001.

[33] M. A. Rivas and M. G. Harbour. POSIX-compatible application-defined scheduling in MaRTE OS. In *14th Euromicro Conference on Real-Time Systems (ECRTS'02)*, pages 67–75, Vienna, Austria, June 2002.

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

[35] 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.

[36] *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, Kiawah Island Resort, SC, Dec. 1999.

[37] 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.

[38] 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.

[39] 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.

[40] W. Yuan, K. Nahrstedt, and K. Kim. R-EDF: A reservation-based EDF scheduling algorithm for multiple multimedia task classes. In RTAS'2001 [34], pages 149–156.