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 though 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 given rise to a host of new scheduling algorithms [1, 6, 8, 13, 20, 24, 26, 29, 31, 32, 33, 34]. 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 [31], 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 poli-

cies 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. Previous papers have presented the Bossa verification process [16] and features for implementing a hierarchy of schedulers [17]. This paper summarizes the compilation process and assesses the performance of the Bossa framework.

The contributions of this paper are as follows:

- 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 real applications, including the Web server Apache, which is highly demanding in terms of context switches.
- 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 when competing with other demanding processes, which it is unable to do when running under the ordinary Linux scheduling policy.

The rest of this 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 presents the Bossa compiler and verifier. Section 5 evaluates the performance of our approach. Section 6 illustrates some applications of Bossa. Section 7 presents related work. Section 8 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 Understanding scheduling 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.

	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

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 analyzing 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

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

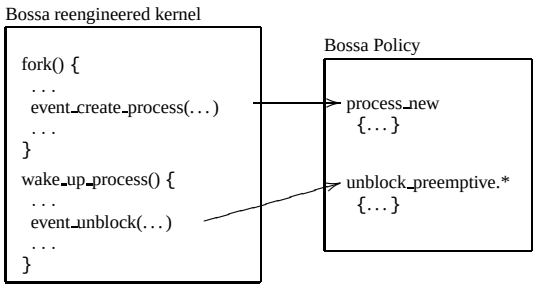


Figure 2. The Bossa event-based interface

currently be blocked. In Linux, however, the 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 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, and may take months of work.

2.2 The Bossa kernel interface

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 the 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. In particular, the complete implementation of process election is removed and replaced by the event notification `bossa.schedule`. We have developed an automated tool to help the OS expert in the re-engineering process [22]. The OS expert also creates a formal model describing the expected behavior of the scheduler handler for each event. This model 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 events such as process creation and termination, process blocking, unblocking and yielding, and the need to elect a new process. In all, there are 10 basic events for which a Bossa scheduling policy must define handlers for use with this kernel. For events such as blocking, unblocking and yielding that are generated by multiple kernel services, Bossa provides specializations of the basic events that include information about the identity of the service that triggered the event. These specialized events are organized into a hierarchy, allowing an event handler to treat all instances of an event or only instances generated by a given source, such as character devices or the network.

3 The Bossa DSL

Programming a scheduling policy requires implementing and keeping consistent complex operations on processes, such as managing process states and process priorities. Because any runtime error can crash the machine, the programmer must carefully check the policy for errors, including both generic errors such as dangling pointers and errors related to incorrect interaction with the target OS. General-purpose languages, such as C, provide no support for the programming of scheduling operations or for the required error checking. The Bossa framework thus includes a DSL that allows the programmer to express scheduling policies in a clear, concise and verifiable way. This language provides high-level abstractions for defining the various data structures and operations needed by a scheduling policy. Bossa provides a verifier that exploits these abstractions to check a scheduling policy. The Bossa compiler then automatically generates an appropriate implementation.

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 [18], shown in Figure 3, 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.[2]

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. 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

[2] `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   : sorted 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 3. Excerpts of an EDF scheduler

of processes in the state. 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 kernel actions. 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 thus represents the currently running process. It is implemented as a process variable, because Bossa currently targets uniprocessors.[3] 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. One is the `unblock.preemptive.*` handler, the first handler in Figure 3, which uses many of the domain-specific constructs. The handler first checks whether the target process is blocked. If so, it preempts 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.

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 definition of the `bossa.schedule` handler, shown in Figure 3, the main effect is to elect a process from the state designated

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

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, enables the Bossa DSL compiler to automatically verify that an event handler satisfies the model of OS-specific requirements provided by the OS expert.

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 forbids storing a reference to a process 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 forbids taking the address of any object, to ensure that data structures are accessed in a controlled way. Finally, 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 constraints ensure termination. Overall, these constraints are well adapted to the domain; as illustrated by Figure 3, the language permits to specify scheduling policies clearly and concisely.

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 [17].

4 Compilation

The Bossa DSL compiler serves as a repository of kernel expertise, making it possible to convert a policy, expressed in a high-level way that requires little kernel knowledge on the part of the programmer, into a module that interacts correctly with the target OS. To address the variation in OS kernel requirements, the compiler is configured by an expert in each target OS with a scheduling-specific model of kernel behavior. When applied to a policy, the compiler first verifies that the policy is compatible with the model of the target kernel and then generates the corresponding C code. The compiled policy can be statically linked with the target kernel or dynamically loaded into the kernel as a module.

4.1 Modeling kernel behavior

To describe kernel behavior, the model provided by the OS expert comprises a set of *event sequences* that describe the possible sequences of events generated by the kernel and a set of *event types* that describe the possible inputs and corresponding outputs for each event handler.

Event sequences Event sequences describe kernel control flow. The event sequence information provided by the OS expert contains a list of *system events*, which occur during system calls, and *interrupt events*, which occur during treatment of an interrupt. The OS expert also provides automata describing the order of these events within a single system call or treatment of a single interrupt. A sequence of events can be declared to be interruptible or uninterruptible, according to the context in which it is generated in the kernel. Based on these automata describing local behavior, the DSL compiler generates an automaton that reflects the global kernel behavior including all possible interleavings of system and interrupt events.

Event types Event types describe the required effect of each event handler on the states of individual processes. Types are defined in terms of the state classes, `RUNNING`, `READY`, `BLOCKED`, and `TERMINATED`, making them independent of the states defined by any particular policy. An event type specifies: (i) the presence or absence of processes in the states associated with a given state class, (ii) the state classes associated with the states of specific processes, such as the target process, (iii) the permitted and required movement of processes between the states of various state classes during the handler, (iv) properties of flag and return values.

The Linux event type for the `unblock.preemptive.*` event is shown below.

```
unblock.preemptive.*:
  ⟨tgt ∈ BLOCKED⟩ → ⟨tgt ∈ READY⟩, return true
  ⟨p ∈ RUNNING, tgt ∈ BLOCKED⟩ → ⟨[p, tgt] ∈ READY⟩,
      current→need_resched = 1, return true
  ⟨tgt ∈ RUNNING⟩ → ⟨ ⟩, return false
  ⟨tgt ∈ READY⟩ → ⟨ ⟩, return false
```

The first two type rules describe the allowed behavior when the target process is blocked. The handler must make the target process eligible by changing its state to a state in the `READY` state class, and may optionally indicate that the running process should be preempted by changing its state from the `RUNNING` state class to a state in the `READY` state class.

In these cases, the return value to be added by the Bossa compiler is `true`, indicating that a process is actually unblocked, as expected by the Linux kernel context in which the unblocking event is generated. The second type rule, in which the running process is preempted, also indicates that the `need_resched` flag of the Linux representation of the running process should be set to 1, as required by the kernel to request preemption in this context. The remaining type rules address the case where the event occurs before the target process has blocked. In this case, the state of the target process must remain unchanged and the return value is `false`, because there is no state change.

The event types can be used not only to check the handlers of a given policy, but also to prove properties about all policies accepted for a given OS. For example, for Linux, the event types can be used to show that any valid Bossa scheduler satisfies liveness, based on the assumption of the correctness of the underlying kernel and the Bossa event notifications. The proof is as follows. We first show that eligible processes are always in a `READY` state. Processes become eligible by either entering the scheduler or by unblocking. The types for these events require that the target process be placed in a `READY` state. The only events besides process election itself that remove a process from a `READY` state are those such as blocking that make a process ineligible. Given that eligible processes are always in a `READY` state, we next consider the type of `bossa.schedule`. This type requires that some process in a `READY` state be elected, if there is any such process. Thus, we are ensured that if there is any eligible process, then some eligible process will be elected, showing liveness.

4.2 Verification

The verification phase of the DSL compiler checks that a policy defines a complete set of event handlers, that each of the handlers respects the corresponding event type, and that the code implementing each handler is well-defined (*i.e.*, will not crash the OS). The heart of the verification is the checking of event types. The verifier first instantiates each type rule with respect to the states defined by the policy. This instantiation uses the event sequences to omit cases that cannot occur due to the inter-handler control-flow of the policy. For each instantiated rule, the verifier then simulates the execution of the corresponding handler with respect to the rule's input. The result is checked to be compatible with the type rules. Information collected during the simulated evaluation is also used to detect errors such as null-pointer dereferences.

Of the verifications performed, we have found that the checking of the event types detects the most errors, as programmers who are not expert in the target kernel do not, and indeed are not expected to, understand all kernel scheduling conventions. The compiler detects these errors before the policy is deployed in the kernel, at which point errors become time-consuming to track down.

4.3 Translation to C code

The translation phase converts the high-level abstractions of the Bossa DSL to executable code. For example, state-change operations are implemented according to the specific data structures associated with the states involved and operations involving process priority are replaced by specialized implementations. The translator also generates infrastructure code that allows the policy to interact with user processes via the `procfs` and `ioctl` interfaces.

All of these tasks represent programming that is tedious and error-prone to do by hand. The automatic generation of code from a high-level specification ensures the consistency of the implementation as the policy evolves and hides the need for expertise in areas such as `procfs` that are kernel-specific and unrelated to the domain of scheduling.

5 Evaluation

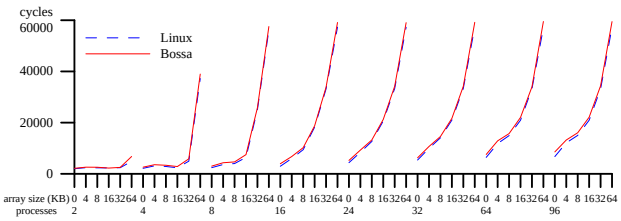
We evaluate Bossa in terms of the expressiveness of the DSL and in terms of performance. Performance experiments were conducted on a 1600MHz Pentium 4 with an 8 KB L1 data cache, a 12 KB L1 instruction cache, a 256 KB L2 cache, 256 MB of RAM, and one 120 GB 7200 RPM Western Digital IDE drive with 8 MB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

Expressiveness of the DSL We are currently using the Bossa DSL to develop an encyclopedic library of multimedia and real-time scheduling policies [15]. 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 that 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 processes. They share 136 lines of code out of 162 and 154 lines, respectively.

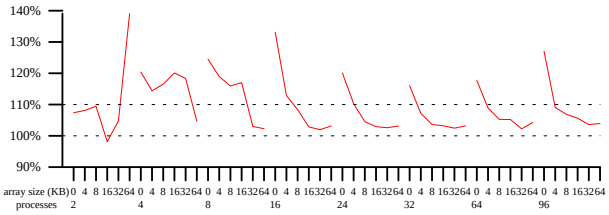
Impact on the context-switch overhead Performing a context switch involves electing a new process, saving the register state of the current process, and installing the register state of the elected process. The context-switch overhead also includes the cost of reloading cache and TLB entries as needed 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

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 [3]	160
BVT [8]	238
Progress-based scheduling [31]	234
Virtual schedulers	
Fixed priority	94
Proportional scheduling	102

Table 1. Code size of some Bossa schedulers



(a) Average context-switch overhead (cycles)



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

Figure 4. Comparison of the Bossa implementation of the Linux policy and the native Linux scheduler (tests done in single user mode)

suite.[4] 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 4 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).

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 4b). 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 4b). 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-purpose system.

Impact on real applications In a real application, the impact of any scheduling overhead is determined by the frequency of context switches. We thus analyze the context-switch behavior of two widely used applications: the video player `mplayer` and the Web server Apache.

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[5] 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.

Apache maintains a pool of threads to treat incoming requests. When a request arrives, it is picked up by an available thread that carries out the processing of the request. This strategy reduces idle time if blocking is required during request processing, but introduces context switches between the threads treating the different requests. When the server is flooded with requests, the number of requests that it can handle is thus limited by the scheduling overhead. For our HTTP server, we used Apache version 1.3. To generate HTTP workloads, we used `httperf`[6] [21]. We performed three tests, each flooding the server with get requests of in-memory pages of various sizes. As shown by Table 2, when running under both Linux and Bossa the Web server handles the same number of requests.

Figure 5 shows that 40-60% of the timeslices of processes forked by Apache have a duration of under 100K cycles. Comparing these results to the context switch time observed for `lat_ctx` (Figure 4a) suggests that these pro-

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

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

[6] `http://freshmeat.net/projects/httperf`, version 0.8.

	page size	request rate (req/s)	bandwidth (MB/s)
Bossa	$<$ 5KB	1163	4.876
Linux	$<$ 5KB	1165	4.842
Bossa	10–15KB	979	9.434
Linux	10–15KB	1007	9.057
Bossa	$>$ 20KB	410	10.128
Linux	$>$ 20KB	408	10.174

Table 2. Apache performance

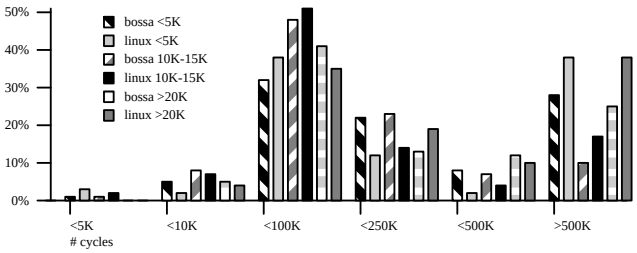


Figure 5. Timeslice durations in Apache

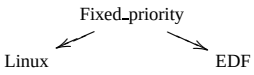


Figure 6. Scheduling hierarchy for `mplayer`

	Min	Max
Linux: Player, kernel compilation	0.009	22.683
EDF: Player, kernel compilation	0.017	0.018

Table 3. Distance between `mplayer` audio and video

	request rate (req/s)	% of Apache Alone
Apache Alone	975	100%
fboding Apache 75%	732	75%
fboding Apache 25%	253	25%
fboding Apache 75%&25%	722/321	75%/32%

Table 4. Apache performance (two servers)

cesses spend a lot of time context switching. Because each of the Apache processes has a resident set of 500-800 KB, the overall memory used by the Apache processes at any given time should be in the range where the overhead of Bossa forms a negligible part of the context switch time, as indicated by the `lat_ctx` tests (Figure 4b).

6 Playing Bossa

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

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, as illustrated in Figure 6. 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 3 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 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. With EDF, the player maintains correct synchronization.

Web server isolation The use of a scheduling hierarchy permits to run multiple web servers in isolation, so that if one is under attack or flooded, the others still react with guaranteed performance. We have implemented this approach using a proportion-based virtual scheduler controlling two Linux schedulers each running a different Apache 1.3 server. One is allocated 25% of the CPU, while the other is allocated 75%. As shown by the second and third lines of Table 4, when flooding one server, the proportion of pages it serves matches its CPU allocation. When flooding both servers simultaneously by two different clients, the 25% server serves a little bit more than its allocation, resulting in 107% of the pages served by a single server.

Teaching real-time scheduling Bossa is an ideal tool for teaching scheduling. In this setting, the main goal is to provide the student with a realistic and comparative understanding of various scheduling algorithms, without discour-

aging him 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.

7 Related work

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

Other work on scheduler development includes that of Ford and Susarla in which a process can donate its CPU time to other processes [11], HLS which allows the creation of scheduling hierarchies [25], Vassal which allows a new scheduler to be dynamically loaded into the Windows NT kernel [7], and the S.Ha.R.K. kernel and MaRTE OS which are OSes specifically designed to facilitate the implementation of new schedulers [12, 27]. In these approaches, schedulers are implemented using ordinary C code, for which no scheduling-specific verification is provided. Thus schedulers have to be assumed to be correct, although scheduling code remains low-level and error-prone.

Recently, there has been much interest in compile-time error detection in the context of OS code. CCured [23], Cyclone [14] and Splint [10] 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 [9] checks properties that can be described as a sequence of matching operations, such as locking and unlocking, and has been applied to OS code. SLAM uses model checking to check similar properties [2]. 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 [4], OCaml in Ensemble [19], and Standard ML in FoxNet [5]. As compared to these approaches, the use of domain-specific tools further targets verification and optimization to specific domain needs.

8 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 currently porting Bossa to Windows XP, Linux 2.6, and the real-time OS Chorus. 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] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In SOSP'99 [30], pages 261–276.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] J. Lawall, H. Duchesne, G. Muller, and A.-F. L. Meur. Bossa Nova: Introducing modularity into the Bossa domain-specific language. In *Generative Programming and Component Engineering: Fourth International Conference, GPCE 2004*, number 3676 in Lecture Notes in Computer Science, Tallinn, Estonia, Sept. 2005. To appear.

[16] 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.

[17] 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.

[18] 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.

[19] 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 [30].

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

[21] D. Mosberger and T. Jin. httperf – a tool for measuring web server performance. *Performance Evaluation Review*, 26(3):31–37, Dec. 1998.

[22] 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.

[23] 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.

[24] 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.

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

[26] J. Regehr and J. A. Stankovic. Augmented CPU reservations: towards predictable execution on general-purpose operating systems. In RTAS'2001 [28], pages 141–148.

[27] 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.

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

[29] 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.

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

[31] 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.

[32] 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.

[33] 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.

[34] W. Yuan, K. Nahrstedt, and K. Kim. R-EDF: A reservation-based EDF scheduling algorithm for multiple multimedia task classes. In RTAS'2001 [28], pages 149–156.