

Bossa

Julia L. Lawall

DIKU, University of Copenhagen
julia@diku.dk

Gilles Muller

Ecole des Mines de Nantes
Gilles.Muller@emn.fr

1. Introduction

Kernel process scheduling is an old problem, but one for which there still appears to be no one universal solution. Instead, work has focused on designing schedulers that provide good performance for collections of applications having particular properties, such as multimedia applications [2, 3, 6, 14], real-time applications [5], or applications running in an energy-constrained environment [11, 15]. While many such scheduling algorithms have been proposed, few can easily be used by end-users, because it remains very difficult to integrate a new process scheduler into an existing operating system.

Bossa is a framework for kernel process-scheduler development whose goal is to ease both the implementation of a scheduler and its integration into an existing operating system [12]. For this, Bossa provides a domain-specific scheduler programming language and a dedicated run-time system. The Bossa domain-specific language makes it easy to express various scheduling concepts. Properties of this language are furthermore exploited by the Bossa compiler, which is able to guarantee that the behavior of a Bossa scheduler is coherent with the expectations of a target operating system [9]. The high level of the Bossa domain-specific language and the associated safety checks have made it possible to use Bossa in teaching labs. The Bossa run-time system has been ported to multiple versions of Linux, to Chorus [8], and to a Java virtual machine optimized for embedded systems [13]. Using both microbenchmarks and realistic applications, we have found that Bossa incurs little or no observable overhead [12].

Bossa was developed at INRIA and the Ecole des Mines de Nantes, in France, and the University of Copenhagen, in Denmark, between 2000 and 2006, with the support of France Telecom research, Microsoft, and the Danish National Research Council. It has been used in research [4, 7] and is currently the subject of a technology transfer to the embedded system and automotive industries [13], with the support of the French National Research Agency.

2. Description

Bossa relies on an event-based run-time architecture, as illustrated in Figure 1. A small run-time system must be initially integrated into the target operating system, to prepare the operating system for use with Bossa. This run-time system causes the operating system to generate event notifications on the occurrence of various scheduling-relevant events, such as process blocking, process unblocking or the need to elect a new process. The need to integrate the run-time system represents an initial overhead for the use of Bossa, but the changes required have been codified in rewrite rules that are applicable to multiple versions of Linux, and that can serve as a guideline for integrating the run-time system in other operating systems [1].

The Bossa run-time system transmits the event notifications to the scheduler, which then takes appropriate actions. These actions may include changing the state of a process or electing a new process based on scheduler-specific criteria. A Bossa scheduler can be implemented as a single *process scheduler*, which manages a collection of processes according to a single scheduling policy, or as a hierarchy of schedulers, containing both process schedulers and *virtual schedulers*, which manage a collection of schedulers. Figure 2 illustrates a possible scheduling hierarchy, in which a proportional virtual scheduler reserves 90% of the CPU time for the processes managed by an EDF scheduler, but gives 10% of the CPU time to a standard Linux scheduler, if needed, to run non-real time processes.

In the rest of this section, we give a brief tour of the Bossa domain-specific language, using extracts of the simple EDF scheduler shown in Figure 3 as an example, and then describe briefly some of the user-level support that is provided for interacting with Bossa.

2.1 Bossa in a nutshell

The Bossa implementation of a process scheduler contains five kinds of declarations, as illustrated in Figure 3: process attributes (lines 2-6), process states (lines 8-14), ordering criteria (line 16), event handlers (lines 18-45),

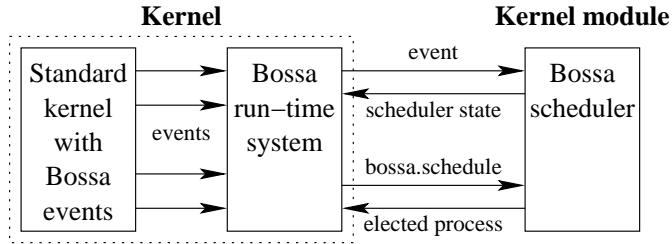


Figure 1. Bossa run-time architecture

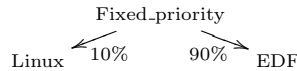


Figure 2. Scheduling hierarchy

and interface functions (lines 47-53). We describe these declarations in turn below. The Bossa implementation of a virtual scheduler is similar, except that the attributes and states refer to the child schedulers, rather than individual processes. Virtual schedulers are described in more detail elsewhere [10].

Process attributes The process attributes declaration contains any policy-specific information that is needed about each process. This information may be provided by the user when the process joins the scheduler, such as the process `period` in the case of the EDF policy (line 3), or be calculated by the scheduler, such as the process's upcoming `absolute_deadline` (line 4). Bossa also provides timers that can be associated with each process. In the EDF policy each process is associated with a timer for detecting the end of the process's current period (line 5).

Process states The state of a process describes its schedulability. The EDF scheduling policy distinguishes between five process states: **running** for the process that is currently running, **ready** for the processes that are able to run, **blocked** for processes that have voluntarily blocked, *e.g.*, to wait for I/O, **computation_ended** for processes that have completed their computation within the current time slice, and **terminated** for processes that are terminating. Each state is associated with a *state class*, which provides some information about the role of the state, analogous to the type of a variable in a general purpose language. Bossa defines four state classes: **RUNNING** for the state that contains the running process, **READY** for the states that contain the processes that are able to run, **BLOCKED** for the states that contain the processes that are not able to run, and **TERMINATED** for the state that contains terminating processes. There may be multiple states in a state class, as illustrated in the EDF policy by **blocked** and **computation_ended** that are both in the **BLOCKED** state. State classes are used by the Bossa compiler, to enable appropriate verifications. State classes can similarly provide guidance to a programmer who wants to understand the scheduling policy. Finally, each state declaration is annotated with some information about its implementation. A state is designated as **process** if it can contain at most one process, as is the case of **running**, because Bossa is currently limited to uniprocessors. A state is designated as **queue** if it can contain multiple processes. Exactly one state in the **READY** state class must be designated as **select**. This state is referenced by the process election operator **select** (line 30). No implementation information is associated with the state in the **TERMINATED** state class, because processes that enter this state are not further manipulated by a Bossa scheduler.

Ordering criteria The ordering criteria specifies the attributes that are taken into account to determine the relative priority of processes for a priority-based scheduling policy. This information is used by the process comparison operators `<` and `>` (line 23) and by the process election operator **select** (line 30). In the case of the EDF policy, a process that has the lowest, *i.e.*, earliest, absolute deadline is considered to have highest priority.

Event handlers The event handlers respond to the various notifications of scheduling-related events that are generated by the operating system. The set of relevant events is determined by the target operating system and its various scheduling requirements. For example, our implementation of Bossa in Linux 2.6 distinguishes the following events: process creation (two kinds), process termination, process blocking, process unblocking (where preemption may or may not be allowed), process yielding (three kinds), the passage of time, and the need to elect a process.

```

scheduler EDF = {
  process = {
    time period;
    time absolute_deadline;
    timer period_timer;
  }

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

  ordering_criteria = { lowest absolute_deadline }

  handler (event e) {
    On block.* { e.target => blocked; }

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

    On bossa.schedule { select() => running; }

    On unblock.timer.period_timer {
      start_relative_timer(e.target.period_timer,
        e.target.period);
      if (e.target in computation_ended) {
        e.target.absolute_deadline = now() + e.target.period;
        e.target => ready;
        if ((!empty(running)) && (e.target > running)) {
          running => ready;
        }
      }
    }
  }

  ...
}

interface = {
  void attach (process p, int period) {
    p.period = ticks_to_time(period);
    p.absolute_deadline = now() + p.period;
    start_relative_timer(p.period_timer,p.period);
    p => ready;
  }

  ...
}

```

Figure 3. Extracts of the Bossa implementation of a simple EDF (Earliest Deadline First) scheduling policy

Finally, a policy that uses timers, as does the EDF policy, must include an event handler for the timeout of each kind of timer.

We describe briefly the `unblock.preemptive` event handler (lines 21-28), as it uses most of the available constructs. An `unblock.preemptive` event notification occurs when a process becomes newly able to run, and that process is allowed to preempt the running process. The parameter `e` of the `handler` block (line 18) refers to the current event, and its field `target` refers to the affected process. The handler begins in line 22 by using the `in` construct to check

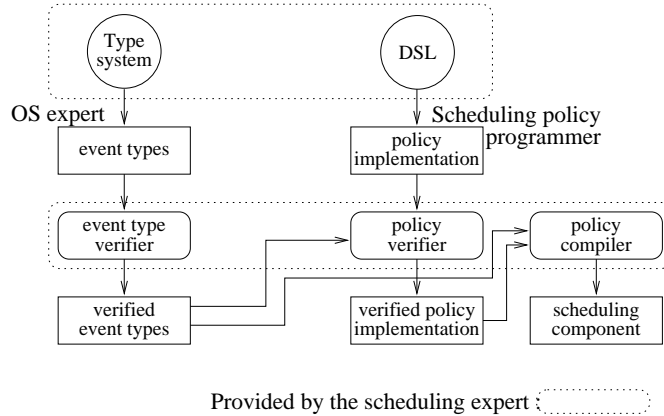


Figure 4. Bossa verification and compilation tools

whether the unblocking process is in the `blocked` state.¹ If the process is actually blocked, line 23 checks whether there is a running process, *i.e.*, whether the `running` state is not empty, and whether the priority of the unblocking process is greater, according to the ordering criteria, than that of the running process. If both conditions hold, the state of the running process is changed to `ready`, to indicate that this process should be preempted (line 24). Finally, the state of the unblocking process is changed to `ready` (line 26).

This handler may seem simple, but, as is typical for code written using a domain-specific language, it hides substantial complexity. The state change operator `=>` hides the low-level and state-dependent data structure manipulations and associated bookkeeping code that are required to move a process from one state to another. The operator `>` performs a comparison that in general may involve many criteria. A state change from or to the `RUNNING` state (lines 24 and 30) induces various low-level operating-specific operations to give or retract a process’s access to the CPU. Finally, the Bossa compiler checks that the state change operations performed by the handler correspond to the expectations of the operating system. These expectations are provided using an auxiliary specification language that is described elsewhere [9].

Interface functions The interface allows the scheduler programmer to define various functions permitting user-level interaction with the scheduler. The EDF scheduler in Figure 3 defines the function `attach` that allows a process to move to it from another scheduler (lines 48-53). This function takes the process’s period as an argument, initializes the process attributes appropriately, and starts the process’s timer. An interface function `detach` (not shown) performs any clean up operations required when a process moves from EDF to another scheduler. Interface functions can also be defined to retrieve any statistics that are maintained by the scheduler or to allow the user-level to provide information to be taken into account by the scheduling policy.

2.2 Bossa tool support

Figure 4 gives an overview of the Bossa compilation process and associated tools. In particular, Bossa provides a language for describing relevant properties of the operating system [9]. Factorizing this information out of the compiler allows the compiler to be essentially operating-system independent while still being able to verify that a scheduling policy satisfies operating-system-specific requirements. For Linux, the compiler generates C code implementing a given scheduling policy as a Linux kernel module, which can be either compiled with the kernel or loaded dynamically. Interface functions are implemented as Linux `ioctl` functions. The Bossa compiler additionally generates stub functions allowing user-level applications to interact with the scheduling policy’s interface functions.

The Bossa framework includes scripts for loading a scheduler into the Linux kernel, for adding or removing schedulers from the current scheduling hierarchy, for attaching a process to a given scheduler, and for doing all of these operations from within a C program. As an example of the latter, Figure 5 shows code that might be used by an application that would like to temporarily run under an EDF scheduler. The function `start()` adds a Proportion scheduler as the root of the current scheduling hierarchy, if it is not already present, giving the current scheduling hierarchy a proportion of 10% (lines 11-12), and then adds an EDF scheduler as a child of Proportion, with a proportion of 90% (lines 13-14). The result is the scheduling hierarchy that was shown in Figure 2. The application

¹It is a peculiarity of Linux that an unblocking process may never have blocked [9].

```

#include "user_stub_Proportion.h"           1
#include "user_stub_EDF.h"                 2
#include "user_stub_Linux.h"              3

int mount(char *parent_name, char *child_name, int child_property_count, int *child_properties); 4
void mount_root(char *new_root_name, int child_property_count, int *child_properties); 5
void unmount(char *tgt_name);              6

void start() {                             7
    int prop,pid;                           8
    prop = 10;                               9
    mount_root("Proportion",1,&prop);        10
    prop = 90;                               11
    mount("Proportion","EDF",1,&prop);      12
    pid = 0; // the current process         13
    period = 10;                             14
    EDF_attach(pid,period);                 15
}                                           16

void end() {                                17
    int pid;                                 18
    pid = 0; // the current process         19
    Linux_attach(pid,BOSSA_SCHED_OTHER,0,20); 20
    unmount("EDF");                         21
    unmount("Proportion");                 22
}                                           23

```

Figure 5. Application code for moving to and from an EDF scheduler

then adds itself as a child of the EDF scheduler. The function `end()` then moves the application back to the Linux scheduler (lines 26-27), and unmounts the EDF and Proportion schedulers (lines 22-23), if they are not managing other processes (lines 24-25).

3. Conclusions and Future Work

In this paper, we have provided an overview of the Bossa framework, particularly focusing on the domain-specific language provided for implementing scheduling policies. This language makes it easy to translate the high-level specification of a scheduling policy, as commonly found in research papers, into an implementation that can be directly compiled and linked with a Bossa-ready version of Linux or another operating system. In addition to the ease of development, the Bossa compiler checks essential correctness properties, particularly focusing on the properties required by the target operating system, which are often not well documented and thus unknown to programmers that are not kernel experts. The benefits of Bossa have been validated by our use of Bossa in teaching, where students with no previous experience with the Linux kernel have been able to implement and test several classical scheduling policies within a day, and without crashing the operating system.

Currently, Bossa is limited to a single processor. The increasing availability of multicore and multiprocessor systems will, however, raise new challenges for kernel process schedulers, and so we plan to extend Bossa to these settings. We also plan to develop libraries of scheduling policies that represent typical strategies for managing processes running on dozens of processors, taking into account both spatial and temporal constraints.

Availability

Various versions of Bossa for Linux are available at the Bossa web site, <http://www.emn.fr/x-info/bossa/>, as well as some documentation and research papers. In particular, the paper “A Framework for Simplifying the Development of Kernel Schedulers: Design and Performance Evaluation” provides a complete overview of the system [12].

Acknowledgements

We are grateful to Christophe Augier, Etienne Dauphin, Arnaud Denoual, Hervé Duchesne, Asger Henriksen, Mads Hvelplund, Varun Kohli, Anne-Françoise Le Meur, Luciano Porto Barreto, and Rickard A. Åberg who have contributed to various parts of the Bossa framework. We are furthermore grateful to France Telecom research, Microsoft, and the Danish National Research Council for their support of this work.

References

- [1] R. A. Åberg, J. L. Lawall, M. Südholt, G. Muller, and A.-F. Le Meur. On the automatic evolution of an OS kernel using temporal logic and AOP. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, pages 196–204, Montreal, Canada, Oct. 2003. IEEE.
- [2] 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.
- [3] 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.
- [4] J. Cordry, N. Bouillot, and S. Bouzefrane. Bossa et le concert virtuel réparti, intégration et paramétrage souple d'une politique d'ordonnancement spécifique pour une application multimédia distribuée. In *13th International Conference on Real-Time Systems*, Paris, France, Apr. 2005.
- [5] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Scheduling in Real-Time Systems*. Wiley, West Sussex, England, 2002.
- [6] P. Goyal, X. Guo, and H. M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 107–121, Seattle, WA, Oct. 1996.
- [7] M. Grenier. *Optimisation de l'ordonnancement sous contrainte de faisabilité*. PhD thesis, INPL, Oct. 2007.
- [8] Jaluna OSware. <http://www.jaluna.com>.
- [9] 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.
- [10] 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.
- [11] J. R. Lorch and A. J. Smith. Scheduling techniques for reducing processor energy use in MacOS. *Wireless Networks*, 3(5):311–324, Oct. 1997.
- [12] G. Muller, J. L. Lawall, and H. Duchesne. A framework for simplifying the development of kernel schedulers: Design and performance evaluation. In *HASE 2005 - High Assurance Systems Engineering Conference*, pages 56–65, Heidelberg, Germany, Oct. 2005.
- [13] SadaJ. <http://www.sadaj.net/>.
- [14] 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.
- [15] 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.