On the automatic evolution of an OS kernel using temporal logic and AOP

Rickard A. Åberg*,[1], Julia L. Lawall**, Mario Südholt*
Gilles Muller*,[2], Anne-Françoise Le Meur†

*OBASCO group
École des Mines de Nantes/INRIA
44307 Nantes Cedex 3, France
{raberg,sudholt,gmuller}@emn.fr

**DIKU
University of Copenhagen
2100 Copenhagen Ø, Denmark
julia@diku.dk

†Compose group
INRIA/LaBRI, ENSEIRB
33402 Talence Cedex, France
lemeur@labri.fr

Abstract

Automating software evolution requires both identifying precisely the affected program points and selecting the appropriate modification at each point. This task is particularly complicated when considering a large program, even when the modifications appear to be systematic.

We illustrate this situation in the context of evolving the Linux kernel to support Bossa, an event-based framework for process-scheduler development. To support Bossa, events must be added at points scattered throughout the kernel. In each case, the choice of event depends on properties of one or a sequence of instructions. To describe precisely the choice of event, we propose to guide the event insertion by using a set of rules, amounting to an aspect, that describes the control-flow contexts in which each event should be generated.

In this paper, we present our approach and describe the set of rules that allows proper event insertion. These rules use temporal logic to describe sequences of instructions that require events to be inserted. We also give an overview of an implementation that we have developed to automatically perform this evolution.

1. Introduction

An important challenge in automating software evolution at the program level is to capture the conditions characterizing the points at which an existing system has to be modified and the changes that should be performed at each such point. This problem is particularly acute when considering large systems that implement many intertwined functionalities, each following a complex protocol. An example of such a system is a modern operating system (OS), which provides a wide range of services, each of which must interact in a consistent way with multiple low-level kernel resources. We examine the problem of OS evolution in the context of adding support for the Bossa framework for scheduler development into the Linux OS kernel. The goal of Bossa is to simplify the design of a kernel-level process scheduler so that an application programmer can develop specific scheduling policies without expert-level OS knowledge [11, 12]. A Bossa scheduling policy is implemented as a module that receives information about process state changes from the kernel via event notifications and uses this information to make scheduling decisions. Preparing a kernel for use with Bossa requires inserting these event notifications at scheduling points throughout the kernel.

Evolution of the Linux kernel to support Bossa is complex, for several reasons. First, we would like Bossa to be usable across the many sub-series Linux releases, which contain bug fixes but not new algorithms. A solution based on patches is not sufficient, because the line numbers of scheduling points and the code immediately surrounding such points can differ across releases. Second, some of the changes required to support Bossa depend on control-flow properties. Detecting such properties by hand is error-prone, even when considering a single version of Linux. Finally, making any changes by hand across multiple files of a large piece of software (Linux currently amounts to over 100MB of source code), is tedious and error-prone. These difficulties are compounded by the fact that standard debugging tools cannot be applied to a running kernel, making it essential that the evolution be done correctly the first time.

These considerations call for an approach to program

1 Partially funded by the EU project EASYCOMP (www.easycomp.org), no. IST-1999-014191; also supported by PELAB, Linköping University, Sweden

2 This work was sponsored in part by Microsoft under contract 2003-146.

evolution that identifies program points by concept rather than line number, and that allows the description of context-sensitive properties. Aspect-oriented programming (AOP) addresses the first issue. This programming technique is targeted towards cleanly implementing a functionality that crosscuts an application. The implementation of such a functionality is isolated in a so-called *aspect*, which contains a collection of code fragments and a formal description of the points at which these fragments should be inserted into the target application. While AOP does not rely on line numbers, typical approaches to AOP only express context information in terms of the stack of pending procedure calls, as determined at run time. In Bossa, the choice of event depends on the sequence of preceding assignments, and this information can be determined statically, based on an intra-procedural analysis, rather than run-time tests. These properties suggest the need for a transformation system that takes into account specific features of kernel code.

In this paper, we present an aspect system that addresses the crosscutting of event notifications scattered over kernel code and allows code fragments to be inserted at the granularity required for the integration of Bossa in an OS kernel. This aspect system uses temporal logic to precisely describe code insertion points and thus resolve the context-sensitivity issue mentioned above. We define an aspect that describes the modifications of the Linux 2.4 kernel needed for the use of Bossa and briefly present the implementation of our aspect system.

In previous work, we instrumented the Linux 2.4.18 kernel by hand to support Bossa. This instrumentation was incomplete, as it did not support all of the kernel services and drivers. Our transformation tool automatically treats all of the files used in a given Linux configuration. Furthermore, by comparing the results produced by our transformation tool with the manual instrumentation, we found that in constructing the manual instrumentation, we did not always choose the correct event notification. Thus, the tool is both easier to use and more robust than a manual instrumentation approach.

The remainder of this paper is structured as follows: Section 2 gives some background on Linux scheduling and Bossa events. Section 3 shows how we define a kernel-evolution aspect for Bossa using rewrite rules based on temporal logic. Section 4 gives an overview of the implementation. Section 5 presents related work and Section 6 concludes.

2. Preparing Linux for Bossa

Figure 1 illustrates the relationship between the Linux kernel and a scheduler implemented using Bossa. The kernel generates scheduling events, that are transmitted to a fixed Bossa run-time system, which then negotiates the

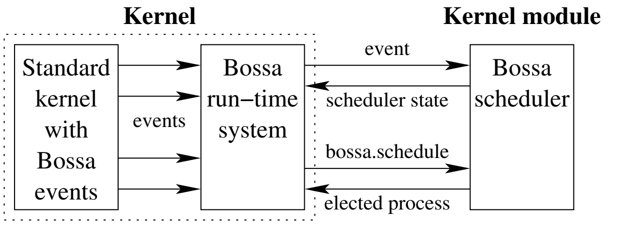

Figure 1. Bossa architecture

scheduling behavior with the scheduler. The run-time system first transmits an event to the scheduler, which processes the event according to its policy and returns a result describing its state to the run-time system. If the state of the scheduler indicates that there is no running process but that some processes are ready to run, then the run-time system requests that the scheduler elect a new process. The use of a fixed event-based interface between the kernel and the Bossa scheduler eliminates almost all dependences between them and thus enables the implementation of a wide range of scheduling policies, with no change to the underlying kernel.

In this paper, our focus is on the insertion of the event notifications into the Linux kernel. To provide some motivation for the choice and position of these events, we first describe the conditions governing the scheduling behavior of Linux. We then use a concrete example to illustrate the use of Bossa events. Finally, we consider some issues involved in inserting event notifications automatically.

2.1. Scheduling in Linux

The basic Linux scheduling operations are process creation, termination, unblocking, blocking, yielding, and the election of a new process. Process creation involves initializing a new process structure and then adding the process to the ready queue. Process termination involves informing the parent of the process that the process is ending and then reclaiming the process structure. Unblocking typically occurs in an interrupt that signals the availability of some resource. Unblocking is performed by the function `try_to_wake_up`, which sets the state of the process to indicate that it is ready to run (`TASK_RUNNING`) and places the process on the ready queue. Finally blocking, yielding and the election of a new process are interdependent, because the blocking or yielding of the running process implies that a new process must be elected. Indeed, in Linux there is no block or yield function: there is only `schedule()` which takes care of blocking or yielding as well.

Linux elects a new process using the function `schedule()`. The effect of a call to `schedule()` on the running process depends on the state of this process. Three states are of particular interest for scheduling: TASK_RUNNING, TASK_UNINTERRUPTIBLE, and TASK_INTERRUPTIBLE. TASK_RUNNING indicates that the process remains ready to run. If the SCHED_YIELD bit is additionally set, however, the process is temporarily ineligible for election if any other ready process is available. The remaining states indicate variants of blocking. Specifically, TASK_UNINTERRUPTIBLE indicates that the process should definitely block, and TASK_INTERRUPTIBLE indicates that the process should only block if there is no pending signal.

The function schedule_timeout() is a wrapper for schedule() that causes the running process to block for at most a limited duration. The instrumentation described for calls to schedule() also applies to calls to schedule_timeout().

2.2. Bossa events

Bossa event notifications amount to function calls that inform the scheduling policy of some change in the overall scheduling state. To identify the source of an event, each event notification is parameterized by a constant indicating the file and function containing the event notification. According to the type and source of each event the scheduler performs an appropriate scheduling action.

To illustrate the use of Bossa events in Linux kernel code, we consider an extract of the Texas Instruments IEEE1394 (Firewire) PCILynx driver for Linux 2.4.18, shown in Figure 2. This code implements scheduling with Bossa when the CONFIG_BOSSA symbol is defined, and amounts to the original Linux code otherwise. Lines 1-11 cause the running process to block until the resource associated with the wait queue md->lynx->mem_dma_intr_wait becomes available. The loop between lines 13 and 25 causes the running process to repeatedly pause until it receives a signal or the condition of the while loop is no longer satisfied.

The original Linux code contains two calls to schedule() (line 10 and line 23). The Bossa runtime system provides specialized variants of this function that implement the Bossa treatment of a running process in specific states. At the call to schedule() in line 10, the state has most recently been initialized (in line 1) to TASK_INTERRUPTIBLE. In this case, we replace the call to schedule() by a call to schedule_interruptible(), which is the variant of schedule() that implements the Bossa treatment of processes in the TASK_INTERRUPTIBLE state. At the call to schedule() in line 23, the state of the running process is TASK_RUNNING; this is the state of a process on return from a preceding call to schedule()

```
1  set_current_state(TASK_INTERRUPTIBLE);
2  add_wait_queue(&md->lynx->mem_dma_intr_wait,
3                 &wait);
4  run_sub_pcl(md->lynx, md->lynx->dmem_pcl, 2,
5              CHANNEL_LOCALBUS);
6
7  #ifdef CONFIG_BOSSA
8          schedule_interruptible(MEM_DMAREAD);
9  #else
10         schedule();
11 #endif
12
13 while (reg_read(md->lynx,
14        DMA_CHAN_CTRL(CHANNEL_LOCALBUS))
15         & DMA_CHAN_CTRL_BUSY) {
16          if (signal_pending(current)) {
17                  retval = -EINTR;
18                  break;
19          }
20 #ifdef CONFIG_BOSSA
21          schedule_running(MEM_DMAREAD);
22 #else
23          schedule();
24 #endif
25 }
26
27 reg_write(md->lynx,
28           DMA_CHAN_CTRL(CHANNEL_LOCALBUS), 0);
29 remove_wait_queue(&md->lynx->mem_dma_intr_wait,
30                   &wait);
```

Figure 2. Excerpt of the PCILynx driver after insertion of Bossa event notifications

(*i.e.*, in line 10 or line 23) and there is no intervening process state change. In this case, we replace the call to schedule() by a call to schedule_running(), which is the variant of schedule() that implements the Bossa treatment of processes in the TASK_RUNNING state.

The choice of schedule function does not affect the algorithm expressed by the PCILynx driver code. Instead, the use of a specialized schedule function informs the Bossa scheduling policy of the state of the running process, so that the policy can use this information when it elects a new process.

2.3. Issues in automating Linux instrumentation

The main problem in preparing a kernel for use with Bossa is to determine the state of the running process at the point of each call to schedule(). Our analysis of Linux kernel code shows state changes occur in the same function as the affected schedule() operation, and thus an intra-procedural analysis is sufficient. Furthermore, at each state change operation whose effect reaches a call to schedule(), the affected process is the running process. Thus, we do not need to detect aliases between process references. The program patterns we consider always appear in one of only a few fixed forms due to kernel coding conventions. For example, after macro expansion, the setting of the process state is always expressed by the direct assignment of a constant state value to the `state` field of the process. Thus, a dataflow analysis is not needed. Overall, these properties suggest that automatic instrumentation can be carried out efficiently.

Many of the Bossa event notifications require simply identifying a single statement and inserting code before, after, or in place of this statement. An example is the rule to replace `try_to_wake_up` by the Bossa event notification for unblocking a process. The instrumentation of a call to `schedule()`, however, depends on the state of the running process, which in turn depends on the set of updates to this state that can reach the call to `schedule()`. For example, treatment of the call to `schedule()` in line 10 of Figure 2 requires considering both the code preceding the `while` loop and the entire loop body. We thus argue for rewrite rules that take into account control flow properties.

3. AOP-based instrumentation of the Linux kernel

We present an aspect for Bossa instrumentation as a set of rewrite rules that use temporal logic to describe the conditions under which specific event notifications should be inserted. Temporal logic is commonly used to express properties of sequences of events, particularly in the context of model checking [8]. Our approach is inspired by that of Lacey *et al.*, who use this logic to define rewrite rules that describe common compiler optimizations [9, 10].

We focus on the rules describing the treatment of `schedule()`, as these are the rules where the choice of event depends on properties of sequences of operations. We first identify the set of rules that is needed, then present the form of our rules, including a brief introduction to the temporal logic we are using, then define some of the predicates and other formulas that we use to describe control flow properties, and finally give the rewrite rules that govern the treatment of calls to `schedule()`. The rest of the rewrite rules are given in the appendix. There are 38 rules in all.

3.1. Linux scheduling properties

The treatment of a call to `schedule()` depends on the state of the running process. Some combinations need not be considered. For example, because a process in the state `TASK_UNINTERRUPTIBLE` can at any point receive an unblocking interrupt that sets its state to `TASK_RUNNING`, there is no need to distinguish the case where the state is explicitly only set to `TASK_UNINTERRUPTIBLE`

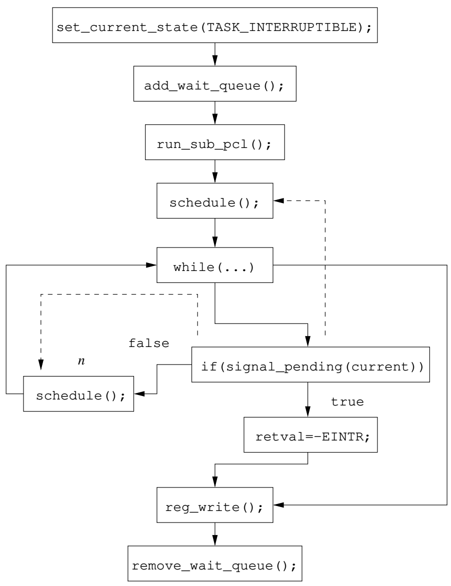

Figure 3. CFG for excerpt of PCILynx driver

from the case where it is additionally set to `TASK_RUNNING` along some control-flow paths. The same observation applies to processes in the state `TASK_INTERRUPTIBLE`. Based on these observations, we need rules that take into account only the following state combinations: `TASK_RUNNING` only, `TASK_RUNNING` and `TASK_UNINTERRUPTIBLE`, `TASK_RUNNING` and `TASK_INTERRUPTIBLE`, and finally `TASK_UNINTERRUPTIBLE` and `TASK_INTERRUPTIBLE`. We furthermore must take into account the possibility of a request to yield the running process. Such a request appears either on all incoming control-flow paths or on none, thus doubling the number of rules. In all, we obtain 8 rules for the treatment of `schedule()`. One more rule is needed for the call to `schedule()` in the kernel idle loop, because Bossa treats the process running the kernel idle loop differently from other processes.

3.2. Rewrite rules

To be able to capture control-flow properties, our rewrite rules are based on control-flow graphs (CFGs), *i.e.*, graphs in which nodes represent either individual statements or decision points of the program, and edges connect nodes that can be executed in sequence. Figure 3 shows the CFG for the code excerpt of Figure 2.

We use rewrite rules of the form:

$$n : \mathit{LHS} \;\Rightarrow\; \mathit{RHS} \qquad \text{If } \mathit{condition} \ldots$$

where *LHS* is a pattern to match against CFG nodes, *RHS* describes the action that should be performed when this match succeeds, and *condition* ... lists the applicability conditions for this transformation. Some actions are Before(n, f), Rewrite(n, f), or After(n, f), which cause a call to f to be inserted before, in place of, or after, respectively, any node n matching the condition. In each case, the inserted call is given a first argument that is an integer constant indicating the file and function containing the matched node. These constants are used by some parts of Bossa to define specialized handling of events from particular sources. If f is replaced by f(args), the inserted call is also passed the arguments used by the code that matches *LHS*. This option is only meaningful if *LHS* matches a function call.

An example of a rewrite rule is:

$$n : (\mathsf{call}(\mathtt{try\_to\_wake\_up}))$$
$$\Rightarrow \mathsf{Rewrite}(n, \mathtt{bossa\_unblock\_process}(\mathtt{args}))$$

This rule matches any call to the function `try_to_wake_up`. A node matching this pattern is given the name n. The use of Rewrite indicates that the call to `try_to_wake_up` is replaced by a call to `bossa_unblock_process`. The function `wake_up_process` shown below illustrates the effect of applying this rule.

```
inline int
wake_up_process(struct task_struct *p) {
#ifdef CONFIG_BOSSA
  return
    bossa_unblock_process(WAKE_UP_PROCESS,p,0);
#else
  return try_to_wake_up(p,0);
#endif
}
```

The applicability conditions in our rewrite rules describe properties of the nodes along a collection of paths in a CFG. For this purpose, we use judgments of the form: $n \vdash \phi$ where n is a node of the CFG and ϕ is a formula of temporal logic (specifically, a variant of CTL [9]). Formulas in this logic are as follows:

$$\begin{array}{rcl} \phi & ::= & p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \\ & & \mid A(\phi_1 \mathsf{U} \phi_2) \mid E(\phi_1 \mathsf{U} \phi_2) \\ & & \mid A\triangle(\phi_1 \mathsf{U} \phi_2) \mid E\triangle(\phi_1 \mathsf{U} \phi_2) \\ & & \mid AX(\phi) \mid EX(\phi) \mid AX\triangle(\phi) \mid EX\triangle(\phi) \end{array}$$

The formula p is any proposition. The operators $\neg$, $\wedge$, and $\vee$ are defined as in propositional logic. The remaining formulas describe universally and existentially quantified collections of paths. We illustrate the semantics of these formulas by examples.

A judgment of the form $n \vdash A(\phi_1 \mathsf{U} \phi_2)$ is satisfied if for each path beginning at n, every node along the path satisfies ϕ_1 *until* a node n' satisfying ϕ_2 is reached, or the path loops infinitely and every node in the path satisfies ϕ_1.[1]

1 Technically, we use the "weak" form of $A(\phi_1 \mathsf{U} \phi_2)$.

The node n' need not satisfy ϕ_1. For Bossa, we are primarily interested in analyzing nodes whose execution precedes a call to `schedule()`, and thus we consider paths that end, rather than begin, at the given node n. Such a backwards search is expressed by the operator $\triangle$; thus, we typically use formulas of the form $A\triangle(\phi_1 \mathsf{U} \phi_2)$ rather than $A(\phi_1 \mathsf{U} \phi_2)$. Using such a formula, we can describe the property that the state of the running process is set to the state `TASK_RUNNING` on all paths reaching some node n, as follows:

$$n \vdash A\triangle(\neg\mathsf{changeOfState}() \; \mathsf{U} \; \mathsf{changeToRunning}())$$

The formulas $E(\phi_1 \mathsf{U} \phi_2)$ and $E\triangle(\phi_1 \mathsf{U} \phi_2)$ are analogous to $A(\phi_1 \mathsf{U} \phi_2)$ and $A\triangle(\phi_1 \mathsf{U} \phi_2)$ but only require the existence of a path whose nodes satisfy the subformulas. Here, however, looping is not allowed; the path must contain a node satisfying ϕ_2.

Calling the function `schedule()` always causes the state of the running process to become `TASK_RUNNING`, and thus the above formula is not sufficient to detect that the state of the running process is already `TASK_RUNNING` at the point of a call to `schedule()`. To avoid taking the effect of `schedule()` itself into account, we need to start the search for state changes at all nodes preceding the current node n. This property is expressed using the formula $AX\triangle(\phi)$. We thus modify the previous example as follows:

$$n \vdash AX\triangle(A\triangle(\neg\mathsf{changeOfState}() \; \mathsf{U} \; \mathsf{changeToRunning}()))$$

The dashed arrows in Figure 3 represent the paths whose nodes are tested in checking this judgment with respect to the node representing the call to `schedule()` within the `while` loop. The formulas $EX\triangle(\phi)$, $AX(\phi)$, and $EX(\phi)$ are defined analogously.

3.3. Describing Linux scheduling properties using temporal logic

The conditions of our rules use a number of atomic predicates to describe properties of the source program. The predicate schedule() is true of a call to either `schedule()` or `schedule_timeout()`. The predicate setState(α) matches a setting of a process state to the state α, where α is the name of a Linux process state. The predicate yield() matches a request to yield the current process. The predicate entry() is true of the entry point of the current function. Finally, the predicate inFunction(f) is true of any code that appears in the function f. For conciseness, we also use the predicates shown in Figure 4.

Any of these predicates can also appear as the *LHS* pattern of a rewrite rule.

To simplify the presentation of the rules, we define some abbreviations for complex formulas, as shown in Figure 5. The abbreviation stateRunning() checks that along all control-flow paths the state has most recently been set to

changeToRunning() $\equiv$
setState(TASK_RUNNING) $\vee$ schedule() $\vee$ entry()
changeToBlocking() $\equiv$
setState(TASK_UNINTERRUPTIBLE) $\vee$
setState(TASK_INTERRUPTIBLE)
changeOfState() $\equiv$
changeToRunning() $\vee$ changeToBlocking()

Figure 4. State predicates

stateRunning() $\equiv$
$AX\triangle(A\triangle(\neg$changeOfState() U changeToRunning()))
stateUnintInt() $\equiv$
$AX\triangle(A\triangle(\neg$changeOfState() U changeToBlocking())) $\wedge$
$EX\triangle(E\triangle(\neg$changeOfState() U
setState(TASK_UNINTERRUPTIBLE))) $\wedge$
$EX\triangle(E\triangle(\neg$changeOfState() U
setState(TASK_INTERRUPTIBLE)))
stateUninterruptible() $\equiv$
$AX\triangle(A\triangle(\neg$changeOfState() U (changeToRunning() $\vee$
setState(TASK_UNINTERRUPTIBLE)))) $\wedge$
$EX\triangle(E\triangle(\neg$changeOfState() U
setState(TASK_UNINTERRUPTIBLE)))
stateInterruptible() $\equiv$
$AX\triangle(A\triangle(\neg$changeOfState() U (changeToRunning() $\vee$
setState(TASK_INTERRUPTIBLE)))) $\wedge$
$EX\triangle(E\triangle(\neg$changeOfState() U
setState(TASK_INTERRUPTIBLE)))
requestYield() $\equiv AX\triangle(A\triangle(\neg$schedule() U yield()))
requestNoYield() $\equiv$
$AX\triangle(A\triangle(\neg$yield() U (schedule() $\vee$ entry())))

Figure 5. Abbreviations of complex formula

TASK_RUNNING, as presented in Section 3.2. The abbreviation stateUnintInt() checks that along all paths the state is most recently set to either TASK_UNINTERRUPTIBLE or TASK_INTERRUPTIBLE. The existential clauses used in this case check that each state occurs along at least one incoming control-flow path. The abbreviations stateUninterruptible() and stateInterruptible() are similar, but require that only TASK_UNINTERRUPTIBLE or only TASK_INTERRUPTIBLE appears. Finally, requestYield() checks that yielding is requested along all paths and requestNoYield() checks that yielding is not requested along any paths.

3.4. Rules for kernel instrumentation

We now present the rules that control the instrumentation of calls to `schedule()` and `schedule_timeout()`. Rules are formulated for `schedule()` and apply analogously to `schedule_timeout()`. For example, in the first rule below, `schedule_timeout_running` would

n : (schedule()) $\Rightarrow$
Rewrite(n, schedule_uninterruptible(args))
If $n \vdash$ stateUninterruptible() $\wedge$ requestNoYield()

n : (schedule()) $\Rightarrow$
Rewrite(n, schedule_yield_uninterruptible(args))
If $n \vdash$ stateUninterruptible() $\wedge$ requestYield()

n : (schedule()) $\Rightarrow$
Rewrite(n, schedule_interruptible(args))
If $n \vdash$ stateInterruptible() $\wedge$ requestNoYield()

n : (schedule()) $\Rightarrow$
Rewrite(n, schedule_yield_interruptible(args))
If $n \vdash$ stateInterruptible() $\wedge$ requestYield()

n : (schedule()) $\Rightarrow$
Rewrite(n, schedule_unint_int(args))
If $n \vdash$ stateUnintInt() $\wedge$ requestNoYield()

n : (schedule()) $\Rightarrow$
Rewrite(n, schedule_yield_unint_int(args))
If $n \vdash$ stateUnintInt() $\wedge$ requestYield()

Figure 6. Rewrite rules for blocking states

be used in place of `schedule_running`.

If the state of the process is TASK_RUNNING at the point of a call to `schedule()`, the call to `schedule()` should be replaced by a call to `schedule_running`. The rule is as follows:

n : (schedule()) $\Rightarrow$ Rewrite(n, schedule_running(args))
If $n \vdash$ stateRunning() $\wedge$ requestNoYield() $\wedge$
$\neg$inFunction(cpu_idle)

This rule checks that the state of the process is TASK_RUNNING and that yielding is not requested. The rule additionally specifies that the call to `schedule()` does not appear in the function `cpu_idle`, which is the function implementing the kernel idle loop. We thus additionally have the rule:

n : (schedule()) $\Rightarrow$ Rewrite(n, schedule_from_idle(args))
If $n \vdash$ stateRunning() $\wedge$ requestNoYield() $\wedge$
inFunction(cpu_idle)

The call to `schedule()` in the kernel idle loop matches only this rule, so we do not check for `cpu_idle` in the remaining rules. Finally, the following rule treats the case where the state is TASK_RUNNING and the process should yield.

n : (schedule()) $\Rightarrow$
Rewrite(n, schedule_yield_running(args))
If $n \vdash$ stateRunning() $\wedge$ requestYield()

The remaining rules are similar, and are shown in Figure 6.

We have presented the complete set of rules for transforming calls to `schedule()` and `schedule_timeout()`. As a sanity check, however, we have an extra rule that checks that every call to `schedule()` or `schedule_timeout()` matches at least one of the above rules:

$$n : (\mathsf{schedule}()) \overset{X}{\Rightarrow} \mathsf{Error}(\text{“nomatchingrule''})$$

The symbol $\overset{X}{\Rightarrow}$ indicates that this rule should only be used if no other rule applies. The action $\mathsf{Error}(s)$ signals an error during the transformation process. In the Linux 2.4.18 configurations we have tested, this rule is never triggered. The transformation engine additionally tests whether multiple rules match. Again, in the Linux 2.4.18 configurations we have tested, this never occurs.

As noted in Section 2.3, we assume that scheduling effects are intra-procedural; that is, a setting of the process state to a blocking state or a request for yielding a process always reaches only the calls to `schedule()` and `schedule_timeout()` in the current function. Complete validation of this hypothesis would require an inter-procedural analysis, and our study of Linux kernel code suggests that scheduling behavior is sufficiently stylized that such an analysis is not needed. Nevertheless, we do include the following rules, which check that there is no control-flow path on which a function ends in a blocking state or requesting to yield a process:

$n : (\mathsf{changeToBlocking}()) \Rightarrow \mathsf{Error}(\text{“unexpectedblock''})$
If $E(\neg\mathsf{changeToRunning}()\ \mathsf{U}\ \mathsf{return}())$

$n : (\mathsf{yield}()) \Rightarrow \mathsf{Error}(\text{“unexpectedyield''})$
If $E(\neg\mathsf{schedule}()\ \mathsf{U}\ \mathsf{return}())$

Unlike the earlier rules, these rules look forward in the control flow, and thus use E rather than $E\triangle$. The rules also use the predicate $\mathsf{return}()$, which matches any sort of return statement, including one that is implicit.

In the Linux configurations that we have tested, these error rules are very rarely triggered. The setting of the state to a blocking state with no subsequent resetting of the state to `TASK_RUNNING` occurs in only two functions (in two separate files, among the device drivers). In one case, the function is part of a kernel thread that terminates after the state change operation. In the other case, the state change reaches the call to `schedule()` that is performed on returning from a system call. This call to `schedule()` is implemented as a generic version that can accept a running process in any state. Nevertheless, this is the only case we have found where a process in a blocking state reaches this call to `schedule`; as the occurrence is in driver code, and driver implementers are often more expert in the associated device than the targeted kernel, we may consider whether this code is actually an error. In the Linux configurations that we have tested, the only case where yielding is requested without subsequently calling `schedule()` or `schedule_timeout()` is in the treatment of the yield system call; this system call is reimplemented as a Bossa event, so the Linux code is not relevant in this case.

4. Instrumentation in practice

We have implemented the Bossa kernel instrumentation using the CIL transformation system for C programs developed by Necula *et al.* [1, 13]. CIL has been shown to treat Linux kernel code correctly and it provides a tool for constructing an intra-procedural CFG.

4.1. Implementation

The implementation of the Bossa kernel instrumentation is divided into two phases: an analysis phase and a patching phase. The analysis phase uses CIL to first create and then traverse the CFG of each function in order to identify the nodes at which rule apply. This traversal includes a standard fixed-point computation to test the satisfaction of temporal logic formulas on subgraphs [8]. The rules themselves are specified in a module that is separate from the rest of the analysis, to make the analyzer easy to extend, *e.g.*, to account for changes in a future major release of Linux. On each match, the analysis phase emits a record of the current program point and the matched rule. This record is then used by the patching phase, which is implemented in Perl, to insert the appropriate event notification.

CIL is a program transformation system, and thus is able to construct C code from its internal representation. We could have implemented our rules by making changes to the intermediate representation during the analysis phase, rather than recording the changes and using a separate patching phase. The C code generated by CIL, however, follows the intermediate representation, in which some of the structure of the original program is not preserved. Patching the original source code implies that the generated code is readable and avoids any inefficiencies that might be introduced by the CIL intermediate representation.

4.2. Assessment

The implementation of the analysis amounts to about 1400 lines of OCaml code, and the implementation of the patching script amounts to about 250 lines of Perl code. The number of files processed depends on the Linux configuration that is used. For a configuration providing standard features on our test machine, 93 `.c` files were processed, of which 77 were instrumented with Bossa events; processing these files additionally implied the patching of 4 header files. This processing requires roughly 15 minutes on a 800 MHz Pentium III, including kernel compilation time. Table 7 summarizes the number of uses of the `schedule()` rules. Table 8 shows the distribution of the uses of these

Event	Occurrences
schedule_from_idle	1
schedule_running	32
schedule_pause_running	19
schedule_uninterruptible	21
schedule_interruptible	34
schedule_unint_int	2
schedule_timeout_running	2
schedule_timeout_uninterruptible	11
schedule_timeout_interruptible	31

Figure 7. The number of occurrences of each `schedule` event (non-occurring events are omitted)

Directory	Events
arch/i386	11
drivers	46
fs	31
ipc	3
include	1
kernel	22
mm	12
net	25

Figure 8. The distribution of `schedule` events

rules within the kernel source tree. Most of the remaining rules apply at only one point in the kernel.

5. Related Work

Our work is in the spirit of Event-based AOP [4], a form of AOP that allows the modification of programs at points determined in terms of arbitrary execution events and the relations between such events. The approach presented in this paper can be interpreted as providing an event-definition language using temporal logic (in contrast to EAOP's functional or imperative event-definition languages) and compile-time aspect application (as opposed to EAOP's dynamic aspect application).

AspectC is another aspect system targeted towards C code and has been used to implement various OS concerns [2, 3]. In this work, the `cflow` construct of AspectJ has been found useful to describe the set of functions that should appear on the call stack if an aspect is to apply. Walker and Murphy propose to consider ordered sequences of calls rather than simply sets of pending calls [14]. Nevertheless, these approaches are based on run-time analysis of the current context, which implies less efficient code than the static analysis of all possible paths, as used in our case.

There have been several uses of logic in specifying non-local properties in program transformation rules. Lacey and de Moor use temporal logic to describe conditions on rewrite rules [9]. We follow their approach here. Subsequent work by Lacey *et al.* shows how to prove the correctness of standard compiler optimizations based on this approach [10]. Drape *et al.* present a variant of logic programming that permits to conveniently express rules of the form we have used here [5]. Nevertheless, their target is .NET rather than kernel C code.

Metal is a language for writing static checkers, that are then executed using the xgcc static analysis engine [7]. Metal checkers have been used to find many bugs in Linux and OpenBSD [6]. Although the goal of these tools is to check properties, xgcc provides some functions for modifying the abstract syntax tree that might be usable for program transformation. The main difference between Metal and our approach is in the underlying logic that is used. Metal is essentially a language for describing state machines. While arbitrary C code can be invoked, thus extending the expressiveness of the language, this code must be manually verified to satisfy the independence and determinacy conditions imposed by xgcc. Indeed, our rules rely on existential and universal quantification over paths, and cannot be expressed in Metal without resorting to the use of C code. By expressing our rules completely within a single logic, we are assured that the rules are well-defined. Furthermore, we can profit from a large body of research on understanding and implementing temporal logic, and our rules serve as an unambiguous form of documentation.

6. Conclusion

In this paper, we have presented an AOP-based transformation system for evolution of an existing OS kernel to support the Bossa framework. The aspect defines 38 rules that use temporal logic to define conditions for instrumenting the original kernel. This approach is applicable to any Linux configuration. The rule language is flexible enough to support future evolutions both in Linux and in Bossa itself.

In future work, we plan to port Bossa to other OSes, such as BSD and Windows, and to apply the Bossa approach to other system services. We anticipate that aspects should be useful to integrate the framework with the OS in these settings as well.

The transformation system and other Bossa-related information can be found at
`http://www.emn.fr/x-info/bossa`.

References

[1] *CIL - Infrastructure for C Program Analysis and Transformation.* `http://manju.cs.berkeley.edu/cil/`.

[2] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Symposium on the Foundation of Software Engineering (ESEC/FSE-01)*, pages 88–98, Vienna, Austria, Sept. 2001.

[3] Y. Coady, G. Kiczales, J. S. Ong, A. Warfield, and M. Feeley. Brittle systems will break – not bend: Can aspect-oriented programming help? In *Proceedings of the Tenth ACM SIGOPS European Workshop*, pages 79–86, Saint-Emilion, France, Sept. 2002.

[4] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proceedings of the 3rd Intl. Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *Lecture Notes in Computer Science*, pages 170–186, Kyoto, Japan, Sept. 2001.

[5] S. Drape, O. de Moor, and G. Sittampalam. Transforming the .NET intermediate language using path logic. In *Intl. Conf. on Principles and Practice of Declarative Programming*, pages 133–144, Pittsburgh, PA, Oct. 2002.

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

[7] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Intl. Conf. on Programming Language Design and Implementation (PLDI)*, pages 69–82, Berlin, Germany, 2002.

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

[9] D. Lacey and O. de Moor. Imperative program transformation by rewriting. In R. Wilhelm, editor, *Intl. Conf. on Compiler Construction (CC)*, volume 2027 of *Lecture Notes in Computer Science*, pages 52–68, Genova, Italy, 2001.

[10] D. Lacey, N. D. Jones, E. Van Wyk, and C. C. Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 283–294, Portland, OR, Jan. 2002.

[11] J. Lawall, G. Muller, and L. P. Barreto. Capturing OS expertise in a modular type system: the Bossa experience. In *Proceedings of the ACM SIGOPS European Workshop 2002 (EW2002)*, pages 54–62, Saint-Emilion, France, Sept. 2002.

[12] G. Muller, J. L. Lawall, L. P. Barreto, and J.-F. Susini. A framework for simplifying the development of kernel schedulers: Design and performance evaluation. Technical report 03/2/INFO, Ecole des Mines de Nantes, 2003.

[13] G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for C program analysis and transformation. In *Intl. Conf. on Compiler Construction (CC)*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228, Grenoble, France, Mar. 2002.

[14] R. J. Walker and G. C. Murphy. Joinpoints as ordered events: Towards applying implicit context to aspect-orientation. In *Proceedings for Advanced Separation of Concerns Workshop*, pages 134–139, Toronto, Canada, May 2001.

Appendix

The remaining rules address the following issues: initialization of Bossa, insertion of Bossa event notifications, removal of undesired functionality from the kernel, invocation of some Bossa utility functions, and addition or augmentation of type and function declarations.

The rules for initializing Bossa are as follows:

$$n : (\mathsf{call}(\mathtt{trap\_init})) \Rightarrow \mathsf{Before}(n, \mathtt{bossa\_init})$$
$$\text{If } \mathsf{inFunction}(\text{start\_kernel})$$

$$n : (\mathsf{call}(\mathtt{rest\_init})) \Rightarrow \mathsf{Before}(n, \mathtt{bossa\_ioctl\_init})$$
$$\text{If } \mathsf{inFunction}(\text{start\_kernel})$$

Most of the Bossa event notifications involve `schedule()`, and the corresponding rules have already been presented. The remaining rules are as follows:

$$n : (\mathsf{call}(\mathtt{wake\_up\_process})) \Rightarrow$$
$$\mathsf{Before}(n, \mathtt{bossa\_new\_process}(\mathtt{args}))$$
$$\text{If } \mathsf{inFunction}(\text{do\_fork})$$

$$n : (\mathsf{call}(\mathtt{free\_pages})) \Rightarrow \mathsf{Before}(n, \mathtt{bossa\_process\_end}(\mathtt{args}))$$
$$\text{If } \mathsf{inFunction}(\text{release\_task})$$

$$n : (\mathsf{call}(\mathtt{calc\_load})) \Rightarrow \mathsf{Before}(n, \mathtt{bossa\_clock\_tick})$$
$$\text{If } \mathsf{inFunction}(\text{update\_times})$$

In some cases, the kernel uses the `need_resched` field of the executing process to indicate that `schedule()` should subsequently be called. In Bossa, the decision to call `schedule()` is controlled by the Bossa policy. We thus remove some of the assignments to the `need_resched` field from the kernel.

$$n : (\mathsf{assign}(\mathsf{field}(\mathtt{need\_resched}), \mathtt{1})) \Rightarrow \mathsf{Rewrite}(n, \{\,\})$$
$$\text{If } \mathsf{inFunction}(\text{rest\_init}) \vee \mathsf{inFunction}(\text{update\_process\_times})$$
$$\vee\ \mathsf{inFunction}(\text{do\_fork})$$

It is convenient that the system calls that affect the process priority transmit the requested priority to the Bossa scheduling policy, in case it can interpret the standard Linux priorities. The rules are as follows:

$$n : (\mathsf{assign}(\mathsf{field}(\mathtt{nice}), \mathtt{niceval}) \Rightarrow$$
$$\mathsf{Before}(n, \mathtt{bossa\_set\_priority}(\&(\mathtt{p\text{->}bossa}), \mathtt{niceval}))$$
$$\text{If } \mathsf{inFunction}(\text{sys\_setpriority})$$

$$n : (\mathsf{assign}(\mathsf{field}(\mathtt{nice}), \mathtt{newprio}) \Rightarrow$$
$$\mathsf{Before}(n, \mathtt{bossa\_set\_priority}(\&(\mathtt{current\text{->}bossa}), \mathtt{newprio}))$$
$$\text{If } \mathsf{inFunction}(\text{sys\_nice})$$

Finally, some rules augment existing structure and enumeration type declarations, and add new variable and function declarations. New variables and functions that should either be local to a single kernel file or that require access to information that is local to an existing kernel file are implemented separately in files that are part of the Bossa runtime system source directory. Rules are then provided to insert `#include` directives to cause such declarations to be inserted into the target file. This strategy avoids cluttering the rules with complex function definitions; rules are limited to small changes within existing definitions.