User-Level Fine-Grained Adaptive Real-Time Scheduling via Temporal Reflection

Sergio Ruocco
National ICT Australia*
and
University of New South Wales, School of Computer Science and Engineering
Sydney 2052, Australia
sergio.ruocco@nicta.com.au

Abstract

Real-time systems must adapt their behaviour when the timing assumptions they are based on change at run time. A viable approach leading to effective adaptations consists of exploiting application-specific knowledge, but limitations of ordinary schedulers constrain its applicability. In this paper this problem is tackled using a reflective *scheduler, which enables a computing system to perform* temporal reflection*, that is to fully observe and control its own temporal behaviour. The scheduler is implemented for the L4 microkernel, and validated by solving a real-time image analysis problem. Compared with other approaches the reflective scheduler is orders of magnitude more precise, achieving microsecond-level accuracy, while its implementation is entirely at user-level, and it does not require any changes to be made to the microkernel itself.*

1. Introduction

The scheduling of a real-time system relies on assumptions about the temporal behaviour of its components. These assumptions span the application, the programming language, the operating system, the hardware architecture, and the external environment. In simpler cases timing variations can be bounded, but each of these components, and their interactions, are far too complex for such assumptions to hold in all situations, making schedules brittle. Therefore, practical real-time systems should incorporate a robust scheduling framework capable of mantaining the correct temporal behaviour even in situations where the underlying assumptions fail.

An approach dealing with scheduling faults is to use adaptive scheduling. A typical solution in soft real-time systems is to adjust the CPU share of a thread based on the frequency of deadline violations. However, such generic adaptation does not work well for all applications. For example, in a real-time video-streaming server, the deadline misses caused by a congested network channel will not be reduced by a larger CPU share, and possibly they would increase. Often more effective adaptations can take place by means of exploiting application-specific knowledge. For example the video-streaming server can react to reduced network bandwidth by increasing the level of frame compression.

Optimal application-driven adaptive scheduling requires two-way communication between the application and the operating-system scheduler. The application imparts its scheduling decisions to the scheduler; the scheduler actuates them to the best of its ability, collects data on the actual scheduling, in particular violations that might have happened, and makes this data available to the application as a base for future scheduling. This is the basic functioning of an adaptive control loop where the application plays both the roles of the controlling and the controlled entity at the same time, and the operating system scheduler is both an actuator of the controlling variable (the tunable system parameter) and a sensor measuring the controlled variable.

However, ordinary kernel-level schedulers cannot perform either of these control functions in a satisfactory way. Complex temporal requirements that combine task importance, dependence, urgency and deadlines require cumbersome and fragile encodings to map to priorities and coarse-grained threads. More importantly, there is no guarantee that scheduling decisions made by the application can be enforced by the in-kernel scheduler. For example high priority threads may block on locks held by lower priority threads, leading to priority inversion, a problem difficult to solve in all but the simplest cases [22,32]. Moreover, schedulers often do not keep accurate, easy to access accounts of task execution time and past behaviour which applications can observe systematically, and end-to-end measurement performed within the task itself cannot take into account preemptions.

*National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs.

Therefore, applications need new abstractions and new mechanisms that allow a better level of observation and control of scheduling. These requirements can be met by a *reflective* scheduler, a time-driven scheduler based on reflective abstractions and mechanisms that allow a system to perform *temporal reflection*, that is to explicitly model and control its temporal behaviour as an object of the system itself [23].

In our previous work [16] we developed a proof of concept implementation of the reflective scheduler in Java. This paper aims to demonstrate that the reflective approach to scheduling can be applied to provide a high-accuracy adaptive real-time scheduling framework for realistic resource-bounded systems. A common example is a mobile system, where meeting real-time constraints with severely limited CPU, memory and energy resources mandates continous fine-grained tradeoffs and adaptation.

We describe our implementation of the reflective scheduler on top of the L4-embedded microkernel [18]. In general microkernels such as GreenHills Integrity and QNX [5] have long been popular for real-time and embedded operating systems. L4-embedded is a version of L4 designed for use in resource-constrained real-time systems such as mobile phones, security cameras and so on. While being a production-quality kernel used in a number of industrial systems, it is also a research vehicle for a growing community of researchers in systems.

Here we give a detailed account of how we mapped the reflective-scheduler design onto the L4-embedded API. This included both issues of functional nature (e.g., implementing interaction and synchronisation between different parts of the scheduler), and temporal nature — achieving microsecond-level accuracy at user level on top of a microkernel like L4, which trades off predictability for performance [24], without changing its internal implementation.

Our implementation of the reflective scheduler relies only on the standard L4-embedded API and a high-resolution hardware timer. Unlike other approaches to reflective scheduling [21], it does not require the development of a new kernel, and unlike other real-time frameworks for the L4 microkernel [6], it does not require any changes to be made to the microkernel itself.

The reflective scheduler does not interfere with standard L4 scheduling: best-effort threads run in the free time between time-critical operations. It takes advantage of the fact that IPC is the most optimised primitive of the L4 microkernel; the critical path between the release of a time-critical operation and its execution is mostly composed by the delivery of an IRQ, via IPC.

The core of the resulting implementation is less than 100 lines of C code (presented in Section 4), while the complete framework is about 1000 loc. The scheduler is fully portable across L4 microkernels that implement the L4::N1 API specification. Moreover, since the entire implementation relies on simple and general scheduling and notification mechanisms, we believe that the results obtained are not tied to the L4 microkernel, and should be considered valid for other microkernels and RTOSes.

We validate our implementation showing an adaptive scheduling strategy that solves a real-time image analysis problem. The solution also demonstrates that the reflective scheduler can perform time-critical operations with microsecond accuracy (on a 400 MHz XScale), an accuracy orders of magnitude beyond what the standard L4 kernel provides, and one order of magnitude beyond a different approach to reflective scheduling [21].

These results show that reflective approaches, generally considered too slow to the point of being impractical even for normal application programming, if properly applied and implemented are instead a viable approach to design and implement adaptive real-time systems, and worthy of further research.

The rest of the paper is structured as follows. Section 2 introduces temporal reflection and the reflective scheduler. Section 3 introduces the L4-embedded microkernel features we used to implement the scheduler. Section 4 presents the design and implementation of the reflective scheduler. Section 5 validates it by solving a real-time image analysis problem. Section 6 discusses related work in the areas of real-time reflective and adaptive scheduling. Finally Section 7 hints at future work and concludes the paper.

2. Reflective scheduler

The reflective scheduler stems from previous research in the application of *computational reflection* to the design and implementation of software architectures for real-time and, more generally, time-sensitive systems [16]. After a brief introduction to computational reflection, we describe the architecture of the reflective scheduler and its functioning, which finally leads to the definition of temporal reflection.

2.1. Computational reflection

Reflection in computing systems was originally introduced by Smith in his fundamental work [25,26]. Smith [26] defines

> "... 'reflection' in its most general sense ... the ability of an agent to reason not only introspectively, about its self and internal thought processes, but also externally, about its behaviour and situation in the world. Ordinary reasoning is external in a simple sense; the point of reflection is to give an agent a more sophisticated stance from which to consider its own presence in that embedding world."

and a reflective system as "a computer system able to reason about itself". Smith's definitions owe to the fact that he introduced reflection in computing through the field of artificial

intelligence, and the actual roots of reflection are in philosophy and logic: Demers [1] provides an accurate overview of the field. Maes [15] gives a less esoteric definition of computational reflection as "the activity performed by a computational system when doing computation about (and by that possibily affecting) its own state and computations".

2.2. Reflection at language and system level

Most research on reflection focuses on *procedural* reflection, that is "self-referential behaviour in programming languages" (see [26] and [25], p. 41). As a consequence, most reflective approaches to real-time scheduling (see Section 6) blindly use techniques and mechanisms developed for procedural reflection, furthering the "tendency to mix reflection and the use of these techniques. In a sense, the tools [are] taken for the concept" [1], resulting in unsound designs and inefficient implementations.

A fertile area of research for applications of reflection develops when the focus of programming shifts from the level of language statements and features, or programming-in-the-small, up to the systemic level of the run-time structure and behaviour of a computational system as a whole, or programming-in-the-large [2]. By applying computational reflection at system level it is possible to deal with systemic aspects of computation that are relevant for real-time and, more generally, time-sensitive systems. The key aspect of interest in a time-sensitive system is its overall temporal behaviour: it is both complementary and orthogonal to its software architecture [4, 31], and, like its architecture, it is not a static property that can be localised in one of its components. Rather, it is an emergent property of the system that results from their execution and interaction at run-time [17].

2.3. Reflective scheduler

Classic real-time schedulers run tasks according to their priorities or deadlines as determined by the application designer. The resulting behaviour is correct as long as the assumptions made during the scheduling analysis hold. However, as soon as the assumptions change such that the current schedule fails, the application must recover quickly from the failure and bring the system to a safe state to avoid serious consequences.

One way to deal with scheduling failures is adaptive scheduling. A control-based approach to real-time adaptive scheduling is to have a modified scheduler notifying a controller module of failures like missed deadlines or jobs[1] dropped because of overload (the *controlled* variable). Based on this information the controller adapts the probability of that failure to repeat in the future by changing a *controlling* variable; in [14] such a variable is the estimated utilisation of the processor. The controller can trade scheduling failures with another system parameter, like energy consumption [33].

[1] A job is the instance of task released at a certain time.

However, schedulers do not keep track of and report on the concatenation of transient temporal interdependencies among jobs that led to a scheduling failure. The job α of a task A can violate its deadline either because A's worst case execution time (WCET) was miscomputed, or it was blocked waiting for a lower priority job β (or preempted by a higher priority job γ) that overran its WCET but still met its deadline; in turn, β and γ may have been waiting for other jobs, and so on. As a consequence, the controller does not have sufficient information to determine exactly which assumptions were violated and led to the scheduling failure. The adaptations it can possibly perform are only *ex-post* and coarse-grained. For finer-grained adaptations new abstractions and new mechanisms that allow a better level of observation and control of scheduling become necessary.

These requirements can be addressed by a *reflective* scheduler, a time-driven scheduler based on reflective abstractions and mechanisms, centred on the idea of representing and controlling the system temporal behaviour as a timed succession of actions. Listing 1 and Figure 1 depict the abstractions and components the scheduler is based on, which we describe in the following paragraphs (adapted and extended from [16]).

```
class TimeInterval {
  int plannedBegin, plannedEnd;
  int actualBegin, actualEnd;
  TimeInterval (int pBegin, int pEnd);
}
abstract Action {
  TimeInterval timeInterval;
  abstract void perform();
  Action (TimeInterval tInt) {
    timeInterval = tint;
  }
}
class RealTimeLine {
  void addAction(Action);
  int now();
}
```

Listing 1. Architectural abstractions

TimeInterval models the planned time interval and the actual time interval for the execution of an operation bracketing them with two [begin, end] pairs of time instants.

Action models the temporal aspects of a computation binding a `perform()` operation with a `TimeInterval`.

RealTimeLine models the "real" time, i.e., a monotonic sequence of time instants characterised by the non-decreasing `now()` value of the current time.

The current time splits the timeline into a past and a future timeline; the two real-time lines decorated by actions represent the past and the future temporal behaviour of the system, respectively. Actions can be inserted into the future timeline and the planned instants of their `TimeIntervals` can be specified. This allows the global temporal behaviour of the system to be *controlled* by planning actions on the future timeline in the proper sequence and with proper timing to suit the application's functional and temporal requirements.

Actual instants of a TimeInterval make sense for the past timeline only. They are used to *record and observe* the past temporal behaviour of the system. Both actual and planned instants are immutable for the past timeline.

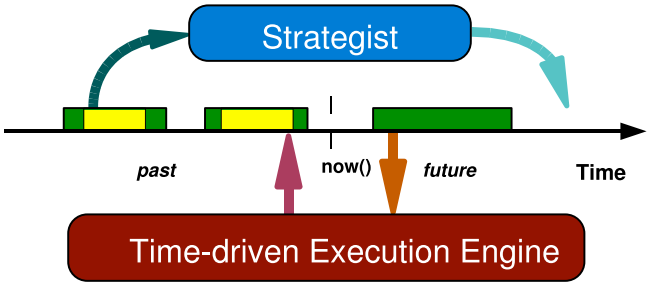

Figure 1. Reflective scheduler and timeline

Along with these abstractions we introduce two architectural components of the system, the *strategist* and the *execution engine*. Their purpose is to control and actuate the temporal behaviour, respectively. By separating the two roles, we achieve separation of policy and mechanism [9].

The execution engine triggers the execution of a computational operation whenever the corresponding action is enabled, i.e., the current time falls inside its planned interval. It also sets the `actualBegin` and `actualEnd` of the `TimeInterval` whenever a computational operation is actually started and completed, respectively. The execution engine provides the required reflective *causal connection* between the actual system temporal behaviour and the architectural abstractions that represents it, i.e. the actions, their time intervals and the real-time line.

The strategist is in charge of observing the past behaviour of the system and planning its future behaviour, respectively, by observing actions in the past timeline, and by inserting actions in the future timeline according to the application goal(s) and requirements. The strategist is an integral part of the application, namely the part in charge of time-related decisions.

In classic priority-based real-time software the equivalent of the strategist is the scheduling logic that statically or dynamically sets the tasks' priorities based on their deadlines and durations according to a given scheduling algorithm like rate-monotonic analysis. The equivalent of the execution engine is the dispatcher within the kernel.

The reflective scheduler has a number of advantages over other approaches: (i) *it clearly identifies and localises in separate components the policy and the mechanism that determine the overall system behaviour*, while traditional approaches leave implicit scheduling control rules scattered in different places and at various levels of abstractions, dispersed among application code, kernel scheduler, locks and low-level interprocess communication mechanisms (all possibly with priority-inversion avoidance algorithms), and so on; (ii) since it knows what scheduling information is relevant for the application, it can record only the bare minimum and abstract it in the most suitable form. This is possible because all the timing data is collected at the decision-making point (a consequence of the policy-mechanism separation); (iii) it is based on abstractions and components that represent and control time and behaviour more explicitly and accurately than ordinary schedulers, threads, priorities, and deadlines; (iv) it allows an application to keep abreast of the current state of scheduling with sufficient detail to spot scheduling faults that anticipate a failure, and possibly take preventive steps to avoid it.

Temporal Reflection The reflective scheduler, however, does not simply separate policy and mechanism. First, it models a time-sensitive system as a system "amenable to dynamic, feedback-based control" [8]. Second, and more importantly, it explicitly represents the system temporal behaviour as an object upon which the strategist can make arbitrary computations. It is then possibile for a computing system to reason about temporal aspects of its own computation. Accordingly, we define *temporal reflection* as "the ability of a system to self-represent (reify), observe and control its own temporal behaviour" as an object of the system itself [23].

3. The L4 microkernel

L4 is a family of second-generation microkernels that aims at high flexibility and maximum performance, but without compromising security. In order to be fast, L4 strives to be small by design [11], and thus provides only the least set of fundamental abstractions and the mechanisms to control them: address spaces with memory mapping operations, threads with basic scheduling and synchronous IPC. We based the reflective scheduler on NICTA::Pistachio-embedded (L4-embedded), an implementation of the N1 API specification [18].

Scheduler L4 contains a 256-level, fixed-priority, round-robin (RR) scheduler. The RR scheduling policy runs threads in priority order until they block in the kernel, are preempted by a higher priority thread that becomes runnable, or exhaust their timeslice. The standard length of a timeslice is 10 ms but can be set between ϵ (the shortest possible timeslice, currently 2–10 ms depending on the platform) and ∞ with the `Schedule()` system call. Once a thread exhausts its timeslice, the scheduler enqueues it at the end of the list of the running threads of the same priority, to give other threads a chance to run. RR achieves a simple form of fairness and, more importantly, guarantees progress.

Synchronous IPC L4 threads can exchange messages via L4's synchronous IPC using a number of convenience functions implemented in terms of the basic `Ipc()` syscall. Besides the classic `Send()` and `Receive()`, for example, there is `Call()`, used by clients to perform a simple RPC to servers, composed by a `Send()`, followed by a `Receive()` from the same thread. Once the request is performed, servers

can reply and then block waiting for the next message using `ReplyWait()`, a `Send()` to a thread followed by a `Receive()` from any thread. To wait for an incoming message one can use `Wait()`, that is a `Receive()` from any thread, and so on. The complete list of IPC convenience functions is defined in the L4-embedded specification [18].

User-level interrupt handlers L4 delivers an interrupt as a synchronous IPC message to a normal user-level thread which registered with the kernel as the *handler thread* for that interrupt. The handler runs in user-mode with its interrupt disabled, but with the other interrupts enabled, and thus it can be preempted by higher-priority threads, which possibly, but not necessarily, are associated with other interrupts.

Asynchronous notification Asynchronous notification is used by a sender thread to notify a receiver thread of an event. While implemented via the IPC syscall, notification is neither blocking for the sender, nor requires the receiver to block waiting for the notification to happen. A sender notifies with `AsynchIPC(tid,bits_to_notify)`. A receiver first specifies with `Set_AsynchMask(bit_mask)` the bits it is interested in, then can poll the current state of its notification word with `Get_NotifyBits()`, or block waiting for the notification of any event *or* a normal, synchronous IPC message using `WaitAsynch(&recv_bits,&src_tid)`.

4. User-level reflective scheduler for L4

This section gives a detailed account of the implementation of the reflective scheduler for L4-based systems. First we list the various requirements and desiderata it should meet, then we show how we designed and implemented its components to meet them, and finally we analyse in detail the code of the execution engine and how it uses the scheduler and L4-embedded APIs.

4.1. Requirements

The reflective scheduler shall meet these requirements:

Timeliness of execution actions shall run at the planned time with minimal jitter. Scheduling accuracy is critical, for example, for applications such as multimedia, software-defined radios and time-sensitive communication protocols.

Accuracy of measurements the actual duration of actions shall be precisely measured, as the well functioning of adaptive strategies depends on the accuracy of the measurements.

Reactivity to changes both the strategist and execution engine shall react promptly to any change on the real timeline, to keep aligned the actual and the desidered temporal behaviour of the system.

Additionally, the reflective scheduler shall properly integrate in L4-based systems. First, real-time applications shall seamlessly run along with non-real-time ones without any consequence, besides the use of processor time spent executing time-critical actions. Second, the reflective scheduler will neither replace nor interfere with the standard L4 scheduler, but rather it will coexist with it. Finally, its implementation will not require any change to the microkernel itself. The latter requirement entails a user-level implementation, in line with the microkernel philosophy [12].

4.2. Design and implementation

We analysed the L4 microkernel API and the aspects of its implementation that affect the temporal behaviour of user-level software [24]. Based on this analysis we designed the reflective scheduler components, its API, and a supporting timer driver in terms of C language structures and the L4-embedded API so as to meet the requirements and desiderata discussed above. The results are described in the following paragraphs and summarised in Listings 2 and 3.

Low-level timer driver The timely execution of actions and the accurate measurement of their duration hinges on *lltimer*, a user-level device driver which defines some abstractions and primitives as C structures and macros to use a hardware timer in a platform-independent, yet very efficient way. The hardware timer and time measurements are modelled by `lltimer` and `lltimepoint`, respectively.

```
struct lltimepoint {
        union {
                uint32_t raw32; // 32 bit timers
                uint64_t raw64; // 64 bit timers
        } raw; // the value read from the hw timer
        usec_time_t usec; // from raw via C macros
};
struct timeinterval {
        struct lltimepoint start;
        struct lltimepoint end;
};
struct Action {
        int                 class;
        struct timeinterval planned;
        struct timeinterval actual;
        void                (* perform) (void);
        int                 clobber;
};
struct RealTimeLine {
        struct lltimer * refClock;
        struct mutex     past_tl_mutex;
        struct  Action * pastHead;
        struct  Action * pastTail;
        L4_ThreadId_t    pastObserverThread;
        int              pastObserverBit;
        struct mutex     future_tl_mutex;
        struct  Action * futureHead;
        struct  Action * futureTail;
        L4_ThreadId_t    futureObserverThread;
        int              futureObserverBit;
};
void            future_enqueue_action (struct Action *);
void            past_enqueue_action   (struct Action *);
void            future_add_observer   (L4_ThreadId_t, int);
void            past_add_observer     (L4_ThreadId_t, int);
struct Action * future_get_first_action (void);
struct Action * future_peek_first_action (void);
struct Action * future_dequeue_action (struct Action *);
struct Action * past_get_latest_action (void);
struct Action * Action_new (int /*class*/,
                void (* /*perform*/ )(void), struct timeinterval *);
void            Action_delete (struct Action *);
```

Listing 2. R. Scheduler abstractions and API

The current implementation of `lltimer` is 500 loc, half of which is the driver for the XScale `OSTMR`, a 32 bit timer clocked at 3.640 MHz.

Actions C functions play the role of time-critical operations, and are referenced via function pointers in `Action`

structures, allocated by `Action_new()` and disposed by `Action_delete()`. The strategist uses the `class` field to quickly identify actions; the execution engine uses the `clobber` field to flag that the execution of this action was delayed by the late termination of the previous one.

Timeline The real timeline is split into a past and a future timeline, each one implemented by a linked list protected from concurrent access by a mutex. The strategist enqueues actions in the future timeline with `future_enqueue_action()` and with `past_get_latest_action()` fetches them from the past timeline as soon as they have been executed. With `future_peek_first_action()` the execution engine peeks at the first (earliest) action on the future timeline, with `future_get_first_action()` removes it, and with `past_enqueue_action()` enqueues an executed action in the past timeline. The execution engine peeks but does not remove the first action because the strategist can replace it any time before execution.

The requirement of reactivity is fulfilled by informing the interested parties of changes to the real timeline via the observer-observable design pattern (described as 'model-view' [3]). After the strategist and execution engine registered themselves as observers of the past and future timeline using, respectively, `future_add_observer()` and `past_add_observer()`, they will be informed of relevant changes of the timelines via L4's asynchronous notification. At the bottom of Listing 3 is the fragment of function `future_enqueue_action()` that notifies the observer if the first action changed. Similarly, `past_enqueue_action()` notifies the observer when it changes the past timeline.

Execution engine At the top of Listing 3 is the source code of the execution engine. Lines 12–14 register the execution engine as an observer of the future timeline and enable L4 IPC to deliver asynchronous notification in addition to ordinary synchronous messages. After this initialisation the execution engine performs its task in an infinite loop.

Line 18 peeks the first (earliest) action on the future timeline. If an action was present, line 20 clears the notification mask so that only new additions will result in notifications. Lines 22–25 use `lltimer` primitives to set up an interrupt for the planned start time of the peeked action. In case the interrupt setup at line 25 succeeds, or the peek of line 18 revealed that an action is not present, control passes to line 35 where it blocks waiting for an IPC message.

The IPC can be either an interrupt from the timer, or a notification that the head of the future timeline changed. If the IPC is an interrupt, line 38 cleans up the fired interrupt, line 39 fetches the first action on the future timeline, and control proceeds to line 43 that, if an action was on the timeline and thus `a` is not `NULL`, exits the loop to execute it. If the IPC was not an IRQ, it can be only a notification: line 42 disarms the interrupt and, since `a` stayed `NULL`, restarts from the beginning at line 16.

In case the interrupt setup at line 25 fails, it means that the current time is already past the planned start time (i.e., the action is late), and the interrupt will fire only when the timer counter reaches the same value after rolling over. Thus

```
1  void execution_engine_thread (void)
2  {
3   L4_Word_t retbits;
4   L4_ThreadId_t src;
5   L4_MsgTag_t tag = L4_Niltag;
6   struct lltimer * clock;
7   struct lltimepoint delay;
8   struct Action * a = NULL;
9   struct Action * pa = NULL;
10  struct timeinterval last_action_interval;
11
12  future_add_observer(L4_Myself()); // observe future timeline
13  L4_Set_AsynchMask((1 << FTL_OBSERVE_BIT));
14  L4_Accept(L4_AsynchItemsAcceptor);
15  while (1) {
16   do {
17    // peek (not fetch) the action to perform
18    pa = future_peek_first_action();
19    if (pa) {
20     L4_Set_AsynchBits(0); // no notifications for this action
21     // compute when this action should start
22     lltimepoint_zero(clock, &delay); // zero the delay
23     LLT_US_TO_RAW(clock, &pa->planned.start);
24     LLT_ADD_RAW(&delay, &pa->planned.start, &clock->start); // offset
25     if (LLT_SLEEP_SETUP(clock, &delay, L4_Myself())) {
26      // timer irq setup failed because "delay" is <= 0
27      // cleanup interrupt we registered for
28      LLT_SLEEP_CLEANUP(clock, &delay, L4_Myself());
29      a = future_get_first_action(); // action to run (assumes a==pa)
30      continue; // will exit the loop if "a" is not NULL
31     }
32     // timer irq setup was successful, fallthrough to wait for IRQ IPC
33    }
34    // wait for timer irq or asynchronous notify IPC
35    tag = L4_WaitAsynch(&retbits, &src);
36    if (tag.X.label == IRQ_LABEL) {
37     // cleanup interrupt we just received
38     LLT_SLEEP_CLEANUP_IRQ(src, clock, &delay, L4_Myself());
39     a = future_get_first_action(); // action to run (assumes a==pa)
40    }
41    else // cleanup interrupt we registered for
42     LLT_SLEEP_CLEANUP(clock, &delay, L4_Myself());
43   } while ( (a == NULL) );
44   // the loop exited with "a" containing the action to execute now
45   LLT_READ_TIMEPOINT(clock, &a->actual.start); // time action start
46   a->perform(a); // execute the action
47   LLT_READ_TIMEPOINT(clock, &a->actual.end); // time action end
48   // make the delay relative to the timer start
49   LLT_SUB_RAW(&a->actual.start, &a->actual.start, &clock->start);
50   LLT_SUB_RAW(&a->actual.end, &a->actual.end, &clock->start);
51   LLT_RAW_TO_US(clock, &a->actual.start);
52   LLT_RAW_TO_US(clock, &a->actual.end);
53   // was the execution of this action delayed by the previous one?
54   if (last_action_interval.end.usec > a->planned.start.usec)
55    a->clobber = 1;
56   last_action_interval = a->actual;
57   past_enqueue_action(a); // store in past timeline
58   a = NULL; // delete reference to "a"
59  } //while
60 }
61
62 void future_enqueue_action
63 (struct RealTimeLine * rtl, struct Action * a) {
64  int first = 0;
65  if ((timeline is empty) || ("a" is the earliest action)) {
66   // enqueue "a" at the head of the list
67   first = 1; // set the first action flag
68  } else { /* add "a" later in queue */ }
69  if (first == 1) // future timeline head changed, notify the observer
70   L4_AsynchIpc(rtl->futureObserverThread,(1<<rtl->futureObserverBit));
71 }
```

Listing 3. E. engine & future_enqueue_action()

lines 28–30 disarm the interrupt, fetch the action and proceed to execute it immediately via the `perform` function pointer.

Policy/mechanism Executing actions unconditionally, even when they are late, avoids hardwiring arbitrary policies in the execution engine mechanism. Appropriate policies to handle the consequences of actions executed late, just like any other scheduling fault, are left to the strategist.

Simplification The execution engine in listing 3 has been simplified for clarity and space reasons. For example, a race condition can occur because for a notification to be received and acted upon, the execution engine must be waiting at line 35, but in some corner cases (e.g, SMP) a strategist can change the first action between the peek at line 18 and the fetch at line 29 or 39, and cause the execution engine to operate on an invalid action. There are simple workarounds (e.g., copy the peeked action and, if after the fetch `a!=pa`, reenqueue `a` in the future timeline and restart from beginning), but they are not included for the above-mentioned reasons.

L4 integration Since actions are atomic they should not be preempted, so the execution engine thread is the only one at its priority level (set to 240, lower than kernel interrupt threads at 255, but higher than standard applications, typically 100), and its timeslice is set to ∞. With this arrangement L4 schedules applications composed by lower-priority threads in the free time between time-critical actions.

Protected scheduling Actions run in the same protection domain as the strategist and execution engine. While we expect real-time adaptive strategies to require domain-specific information available only from within an application, protected scheduling can be performed by encapsulating in an action a synchronous `Call()` IPC to threads in other protection domains (but possibly increasing the activation jitter).

Scheduling latency Since the execution engine is always ready to receive the timer IPC, and has higher priority than normal applications, L4 can receive the interrupt and perform a *direct process switch* [10] from kernel mode to the execution engine at user level without running the kernel scheduler [24]. As a consequence, the latency of the reflective scheduler is low and roughly constant, and can be compensated by the strategist as shown in Section 5.5.

5. Experimental evaluation

We evaluate the performance of the reflective scheduler by implementing and benchmarking a real-time application that exploits the ability of the scheduler to adapt the future application behaviour based on the observation of its recent past.

5.1. The problem

We choose an adaptive real-time image analysis problem that occurs in real-world security monitoring systems. It involves a digital video signal encoded in MPEG that must be monitored for anomalous behaviour. In our case this behaviour is the presence of sparks or lightning. A scenario where this might be required is an industrial plant that utilises flammable gas and where a spark or lightning might cause an explosion. The application decodes the encoded video stream and applies the image analysis algorithm to the stream. Upon finding a spark or lightning, it reports its position to the operator. The analysis must occur in real-time.

Time-adaptive image-analysis algorithm We selected an algorithm to find lightning and sparks which scales with respect to the spatial resolution of image analysis. The algorithm takes RGB frames and looks for pixels with luma (Y) exceeding a given threshold. It outputs a bounding rectangle that marks the area of interest so that it can be further processed. An example of the resulting frame is shown in Figure 2. The scalability is achieved by varying the horizontal and vertical step at which the image is scanned.

Figure 2. The analysis locates a lightning

5.2. Real-time requirements

The application described above has the following real-time requirement: it must process all video frames without skipping or delaying any of them. In fact, a spark or a lightning strike is a phenomenon with high temporal resolution: skipping a single frame can lead to missing it. Delayed processing of video frames would result in late alarms. Therefore, unlike conventional soft real-time media-processing applications that can skip or buffer frames, our scheduler should satisfy a firm real-time constraint.

The timely analysis of a frame poses the following problems. First, the time available before the deadline is known only at run time; it depends on video rate, I/O speed (platform-, network-, and device-dependent), decoding times (CPU-dependent) and so on. While variations in frame rate and decoding times could in principle be managed, I/O makes things extremely unpredictable. Benchmarks we performed show that, even from a RAM-based filesystem, read times varies with a factor of 3. Second, the duration of image analysis itself can be difficult to estimate in advance. It has a complex dependency on the algorithm and the image

Figure 3. Read-parse-analyze-display real-time plan

size, but also on CPU and memory performance, which can change at run-time for many reasons (e.g., power management and DMA traffic). Static methods like WCET are pessimistic, and would lead to too many frames skipped and underutilised available time. Therefore we want to use adaptive scheduling.

5.3. Adaptive strategy

To meet the requirements it is necessary to plan the timing of both the MPEG video decoding and the analysis of each frame. Figure 3 shows the typical read-parse-analyse-display cycle as it appears on the timeline.

Strategist The core strategist logic schedules four types of actions on the timeline, `read`, `parse`, `analyse` and `display`, and handles four types of events corresponding to completion of these actions. Listing 4 shows the main application actions planned by the strategist. Listing 5 shows the "action handlers", the functions called by the strategist when it receives the notification that an action has been executed. An action handler plans future action(s) according to the action just executed, its actual duration, the current time, and the system state. In other words, while actions are application operations, action handlers are subroutines of the strategist, called right after the action has been executed.

Adaptation tactic We achieved the required adaptivity to the unpredictable available time and analysis duration by exploiting our knowledge of the algorithm. As the analysis duration has a positive (although unknown) correlation with the resolution, this information is collected and recorded at run-time in an adaptation table (Table 1). Each line contains the longest execution time of the analysis algorithm observed so far at the given quality setting, and the last frame where it was observed. The table is used by `lookup()` to select the best quality level given the available time as follows. If, for example, the available time is 9000 μs, `lookup()` will select the pair (1,2); if the available time is 18000 μs, the pair (1,1) will be selected. If, instead, the available time is only 5000 μs, `lookup()` will return the (1,3) quality setting, in the hope that the resulting speedup will be enough. The quality of the adaptation improves over time, as `update_adaptive_table()` refines the longest observed duration at a given quality setting with the actual execution time of the `analyse` action provided by the execution engine. Algorithms with more complex timing behavior can be accommodated with correspondingly sophisticated tactics.

frame	analysis length (μs)	dx	dy	quality (1/dxdy)
16	15211	1	1	1
18	7373	1	2	1/2
0	(no data)	1	3	1/3
0	(no data)	2	2	1/4
0	(no data)	2	3	1/6

Table 1. The adaptive table after 50 frames

Actions and actions handlers The actions are found in Listing 4 and perform the following operations: `mpeg2_read_action()` reads the raw bitstream from a file and stores it in the parser buffer; `mpeg2_parse_action()` feeds data from the IO buffer into the MPEG2 decoder; `analyze_action()` analyses the frame for sparks and lightning, subsampling the image along the x and y axes by `dy` and `dy` pixel, respectively (both global variables for convenience of exposition); finally, `display_action()` displays a frame.

```
void mpeg2_read_action () {
  size = fread (buffer, 1, BUFFER_SIZE, mpgfile);
  mpeg2_buffer (decoder, buffer, buffer + size);
}
void mpeg2_parse_action () {
  mpeg2_parse_state = mpeg2_parse (decoder);
}
void analyze_action () {
#define BRIGHTNESS_TRESHOLD 120 // dx. dy are global
  rectangle_rgb (info->sequence->width, info->sequence->height,
    info->display_fbuf->buf[0], BRIGHTNESS_TRESHOLD, dx, dy);
}
void display_action () {
  // display the frame
}
```

Listing 4. Actions

The action handlers are found in Listing 5 and perform the following operations: `handle_read()` if some data was read, schedules a new `parse` action; `handle_parse()` is the core function of the strategist. If the MPEG decoder status code indicates that a new frame has just been decoded, `handle_parse()` schedules two actions. First, it calls `plan_display()` (line 37) to plan the `display` action, which constitutes the final step of processing of the video frame. Second, if there is some time remaining before the display of the frame, it schedules the `analysis` action (line 38–43). The `lookup()` function (line 41) selects

the quality of the analysis algorithm based on available time by performing lookup into the adaptation table, as described above; `handle_analyse()` updates the quality adaptation table based on the actual execution time of the image analysis algorithm on the current frame; `handle_display()` restarts the read-parse-analyze-display cycle with a parse.

```
1  int handle_analyse(struct Action * a) {
2   update_adaptive_table (adaptive_table,
3    a->actual.end.usec - a->actual.start.usec, dx, dy, a->frameno);
4   Action_delete(a);
5   return (ST_STEADY);
6  }
7  int handle_read(struct Action * a) {
8   usec_time_t now;
9   int newstate = ST_STEADY;
10  Action_delete(a); // dispose executed action
11  if (size) {
12   now = lltimer_now_usec(clock);
13   plan_parse (now, now + PARSE_LEN);
14  }
15  else
16   newstate = ST_END; // EOF
17  return (newstate);
18 }
19 int handle_parse(struct Action * a) {
20  struct Action * newa;
21  usec_time_t now, when;
22  static usec_time_t first_frame_time;
23  switch ( mpeg2_parse_state )
24  {
25   case STATE_FRAME_READY: {
26    usec_reltime_t avail_time; // can be negative
27    framenum++;
28    // compute when the display of this frame should happen
29    now = lltimer_now_usec(clock);
30    if (framenum == 1) { // the 1st frame starts at round second
31     first_frame_time = SEC_TO_USEC((now / 1000000))+SEC_TO_USEC(2);
32     when = first_frame_time;
33    } else
34     when = first_frame_time + (framenum-1) * frame_interval[framerate];
35     // frame_interval is display period
36    // plan display even if 'when' is past 'now' (i.e., missed deadline!)
37    if ((newa = plan_display (when, when + DISPLAY_LEN))) {
38     now = lltimer_now_usec(clock); // refresh 'now'
39     avail_time = when - now; // avail. time for image analysis
40     if ((avail_time > 0) &&
41       (lookup (adapt_table,ADAPT_TABLE_SIZE,avail_time,&dx,&dy))) {
42      // sets analysis quality in dx/dy, based on the available time
43      newa = plan_analyze (now, now + ANALYZE_LEN);
44     }
45    }
46    Action_delete(a); // parse action is not necessary anymore
47    break;
48   }
49   case STATE_BUFFER: {
50    now = lltimer_now_usec(clock);
51    newa = plan_read (now, now + READ_LEN);
52    Action_delete(a); // discard old parse
53    break;
54   }
55  }
56  return (ST_STEADY);
57 }
58 int handle_display (struct Action * a) {
59  now = lltimer_now_usec(clock);
60  new_cycle = plan_parse (now, now + PARSE_LEN);
61  Action_delete(a); // discard old display
62  return (ST_STEADY);
63 }
```

Listing 5. Actions handlers of the Strategist

5.4. Experimental results

We run the resulting system on PLEB2 [19], an embedded platform based on an Intel XScale PXA 255 CPU clocked at 400 MHz and running the L4-embedded microkernel. The results of the execution are shown in Figure 4. It shows the accuracy with which the scheduler executes the display after the frame analysis at a given quality level. More precisely, it shows the amount of jitter (in μs) between the *planned* and the *actual* start time of the display action caused by the previous frame analysis action performed at a quality level selected in the adaptation table. In the first five frames the table is empty, and the analysis action overruns or is skipped. After the fifth frame, however, the table is filled, and the scheduler meets the display deadline within 52–57 μs.

5.5. Closed-loop feedback calibration of the scheduler

The reflective scheduler allows a *system-wide* tuning of execution timing by applying a correction factor at *a single place in the system*. In Figure 4, even after the adaptive scheduling started to operate correctly, there is still a visible delay of 52–57 μs in the execution of a `display` action. It is probably caused by a combination of the latency of user-level interrupt handling and the overhead of the reflective scheduler, but we have not characterised them further at the moment. Listing 6 shows a fragment of the strategist that attempts to reduce this delay and thus improve scheduling accuracy by using an adaptation tactic based on closed-loop feedback calibration. A simple algorithm measures the error, and corrects the anticipation of the timer interrupt in the `lltimer` driver. The function bases its computation on the raw timer ticks, less practical but typically more accurate than microseconds used for actions planning. It considers only actions whose execution was not delayed (clobbered) by previous actions. As the source indicates the calibration is applied after the 25th frame for illustration purposes. The results are visible in Figure 5. Just three frames after frame number 25, the feedback-calibrated scheduler meets display deadlines with an accuracy oscillating between -4 and +3 μs, which is an improvement of an order of magnitude. It also

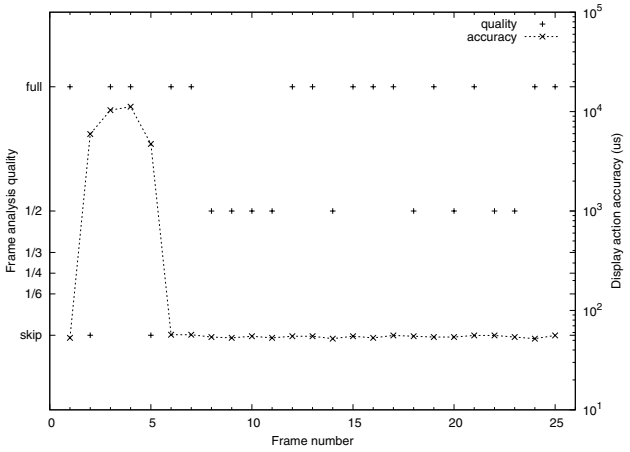

Figure 4. R. scheduler: accuracy vs quality

```
handle_display (struct Action * a) {
  // ... prologue of handle_display() ...
  if ((a->clobber == 0) && (framenum > 25)) {
    int error, newcal, oldcal;
    error = a->actual.start.raw.raw32 - a->planned.start.raw.raw32;
    switch ( abs(error) ) {
      case 0: // timely: no correction necessary
        break;
      case 1:
        error = error * 2; // intentional fall-through
      default: {
        oldcal = lltimer_get_calibration(clock);
        newcal = - (error) / 2 + oldcal; // feedback correction
        lltimer_set_calibration (clock, newcal);
      }}}
  // ... epilogue of handle_display() ...
}
```

Listing 6. Timer feedback calibration

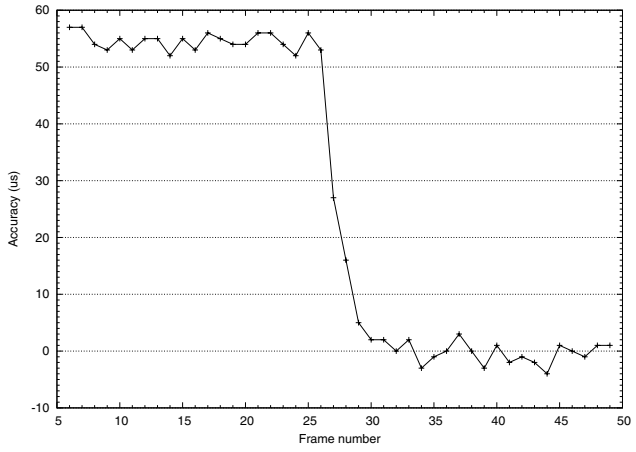

Figure 5. Scheduler feedback calibration

improves by an order of magnitude over a different approach to reflective scheduling, in which a reaction of the scheduler (on a Cyrix 233 MHz processor) takes "...on an average not more than 30 μs in the worst case..." [21]. Note that this tactic dynamically self-adapts to changes in operating system overhead or hardware performance. Of course, investigating the actual reasons of the delay could lead to a better tactic.

5.6. Discussion

Some closing comments: (i) the reflective scheduler achieves a clean separation of functional and timing concerns both at conceptual and implementation level [13]: writing and testing different scheduling policies, then, simply amounts to writing and testing the strategist's functions, which can be easily evolved independently from functional code, and even replaced at run-time; (ii) for the example at hand the reflective scheduler allowed well-localised solutions for both coarse-grained and fine-grained adaptivity problems: the code dealing with changes in the speeds of algorithm, platform and I/O is not only entirely contained in the strategist, but also very localised within it, and if necessary it can be replaced by more sophisticated heuristics to cope with more complex actions' timing behaviour with very little or no changes to the rest of the system; (iii) to the extent that strategies can be made adaptive to the speed of the execution platform, they are (in principle) *portable* across architectures with different performance characteristics; (iv) the decomposition of the application code into actions does not require anything new or special: in most cases the actions' code is the tasks' code of traditional tasks-based schedulers; (v) the atomicity of actions is what makes the real-time line a correct model of system behaviour: if inconvenient it may be relaxed by introducing preemption in the model (but complicating it), or addressed refining a coarse action into a sequence of finer-grained actions; (vi) the purely reactive design of this paper's strategist is not prescriptive, but simply convenient for illustration purposes and the specific example at hand: the design of strategists is a new area of real-time and time-sensitive systems research.

6. Related Work

We presented a proof-of-concept implementation of the reflective scheduler in [16]. Its main disadvantage is the Java implementation, which introduces jitter in the order of 10s of milliseconds.

Reflective scheduling is a recent and active branch of real-time research [7,20,21,27–29]. In general other reflective approaches to real-time scheduling leverage techniques developed for procedural reflection (see Section 2.2). In the same manner as our reflective scheduler, they allow the designer to disentangle and cleanly separate the application code dealing with functional computations (base-level) from the code dealing with temporal behaviour (meta-level). However, unlike our reflective scheduler, their implementations typically involve an interpreted language [7], require changes to existing systems, or even a new ad-hoc operating system [20, 21], with a major impact on complexity and performance. More importantly, their meta-level controls the base-level temporal behaviour with ordinary threads and priority-based schedulers, shown in this paper to be both less expressive and less accurate than their reflective counterparts.

Fiasco [6] is a real-time variant of L4 that sports sophisticated kernel-level support for periodic real-time tasks [30]. The reflective scheduler achieves flexible real-time scheduling both for periodic and non-periodic tasks without modifications to the L4-embedded microkernel.

Finally, other approaches to adaptive real-time scheduling exist. The framework presented in [14], for example, aims at regulating real-time scheduling using as controlled variable the miss ratio, which is "the number of deadline misses divided by the total number of completed and aborted task", while the controlling (manipulated) variable is the "total estimated CPU utilisation". The level of control achieved with this approach is more coarse-grained than what achieved via reflective scheduling.

7. Conclusions

In this paper we presented the design and the implementation of a reflective real-time scheduler for the L4-embedded microkernel. We demonstrated that it is possibile to do fine-grained, microsecond-level accurate real-time adaptive scheduling on top of a standard, general-purpose microkernel without changing its implementation. We also showed that a reflective scheduler models a time-sensitive system as a system amenable to dynamic, feedback-based control. Future work includes tackling more complex real-time system designs, research on the design of strategists and their proofs of correctness, analysis of scheduler performance with CPU and IO bound tasks and under interrupt overload, application to power management and extensions for SMT/SMP systems.

Acknowledgments

The author wants to thank Gernot Heiser, Ihor Kuz, Godfrey van der Linden, Marco Ruocco, Leonid Ryzhyk, Patryk Zadarnowski and the anonymous reviewers. Their feedback improved both the content and style of this paper.

References

[1] F.-N. Demers and J. Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *IJCAI WS on Reflection and Metalevel Architectures and their Applications in AI*, Montréal, Canada, Aug 1995.

[2] F. DeRemer and H. H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, 2(2):80–86, Jun 1976.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[4] D. Garlan and M. Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, Jan 1994.

[5] D. Hildebrand. An architectural overview of QNX. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 113–126. USENIX, 1992.

[6] M. Hohmuth. The Fiasco kernel: requirements definition. Technical Report TUD-FI98-12, Dec 1998.

[7] Y. Honda and M. Tokoro. Time-dependent programming and reflection: experiences on R^2 architecture. Technical report, SCSL-TR-93-017, Sony Computer Science Laboratory, 1993.

[8] C. Karamanolis, M. Karlsson, and X. Zhu. Designing controllable computer systems. In *Proceedings of USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, pages 49–54, Santa Fe, NewMexico, USA, Jun 2005.

[9] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in Hydra. In *Proceedings of the 5th ACM Symposium on Operating systems principles (SOSP)*, pages 132–140. ACM Press, 1975.

[10] J. Liedtke. Improving IPC by kernel design. In *14th SOSP*, pages 175–88, Asheville, NC, USA, Dec 1993.

[11] J. Liedtke. μ-Kernels must and can be small. In *5th IWOOOS*, pages 152–161, Seattle, WA, USA, Oct 1996. IEEE.

[12] J. Liedtke. Towards real microkernels. *CACM*, 39(9):70–77, Sep 1996.

[13] C. V. Lopes and W. L. Hursch. Separation of concerns. Technical report, College of Computer Science, Northeastern University, Boston, Feb 1995.

[14] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao. Feedback control real-time scheduling: framework, modeling, and algorithms. *Real-Time Systems*, 23(1-2):85–126, 2002.

[15] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of OOPSLA'87*, Orlando, Fl., USA, Oct 1987. ACM Press.

[16] D. Micucci, S. Ruocco, F. Tisato, and A. Trentini. Time sensitive architectures: a reflective approach. In *Proceedings of 7th International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, Vienna, Austria, May 2004. IEEE Computer Society Press.

[17] J. C. Mogul. Emergent (mis)behavior vs. complex software systems. In *Proceedings of EuroSys2006*, Leuven, Belgium, Apr 2006.

[18] National ICT Australia. *NICTA L4-embedded Kernel Reference Manual Version N1*, Oct 2005. http://ertos.nicta.com.au/Software/systems/kenge/pistachio/refman.pdf.

[19] National ICT Australia. PLEB, 2005. http://www.ertos.nicta.com.au/hardware/pleb/.

[20] A. Patil and N. Audsley. An application adaptive generic module-based reflective framework for real-time operating systems. In *Proceedings Work in Progress session of Real-time Systems Symposium*, Lisbon, Portugal, Dec 2004. Computer Society, IEEE.

[21] A. Patil and N. Audsley. Implementing application specific RTOS policies using reflection. In *Proceedings of 11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS)*, pages 438–447, Mar 2005.

[22] G. E. Reeves, D. Wilner, and M. B. Jones. What really happened on Mars? Dec 1997. http://research.microsoft.com/~mbj/Mars_Pathfinder/.

[23] S. Ruocco. *Temporal Reflection*. Ph.D. Thesis, Università degli Studi di Milano, Milano, Italy, Feb 2004.

[24] S. Ruocco. Real-Time Programming and L4 Microkernels. In *2006 WS Operat. System Platforms for Embedded Real-Time applications*, Dresden, Germany, Jul 2006.

[25] B. C. Smith. Reflection and semantics in a procedural language. Technical Report 272, MIT LCS, 1982.

[26] B. C. Smith. Reflection and semantics in Lisp. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 23–35, Salt Lake City, Utah, United States, 1984.

[27] J. Stankovic and K. Ramamritham. A reflective architecture for real-time operating systems. In Sang Son, editor, *Advances in Real-Time Systems*. Prentice-Hall, 1994.

[28] J. A. Stankovic. On the reflective nature of the Spring kernel. Technical Report UM-CS-1990-119, University of Massachusetts, 1990.

[29] J. A. Stankovic. Reflective real-time systems. Technical Report UM-CS-1993-056, University of Massachusetts, 1993.

[30] U. A. Steinberg. Quality assuring scheduling. Diploma thesis, Dresden University of Technology, Mar 2004.

[31] F. Tisato, A. Savigni, W. Cazzola, and A. Sosio. Architectural reflection: realising software architectures via reflective activities. In W. Emmerich and S. Tai, editors, *EDO*, volume 1999 of *Lecture Notes in Computer Science*, pages 102–115. Springer, 2000.

[32] V. Yodaiken. Against priority inheritance. Available at http://www.fsmlabs.com/against-priority-inheritance.html.

[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 Systems Principles (SOSP)*, pages 149–163, Bolton Landing, NY, USA, 2003. ACM Press.