Observation Tools for Effective Schedules in a RTOS.

Moustapha Bikienga
LIAS
Univ. Koudougou & ENSMA
01 BP 376
Koudougou, Burkina Faso
moustapha.bikienga@lisi.ensma.fr

Dominique Geniet
LIAS
Univ. Poitiers & ENSMA
1 av. Clément Ader, Téléport 2
F-86961 Chasseneuil, France
dominique.geniet@univ-poitiers.fr

Annie Choquet-Geniet
LIAS
Univ. Poitiers & ENSMA
1 av. Clément Ader, Téléport 2
F-86961 Chasseneuil, France
annie.geniet@univ-poitiers.fr

ABSTRACT

Real-time scheduling validation usually stands on emulators: the scheduling policy is validated, not the effective scheduler. We propose a strategy to calibrate scheduling observers, that aim to validate effective implementations of schedules.

Categories and Subject Descriptors

C.3 [**Spec.-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.4.1 [**Operating Systems**]: Process Management—*Scheduling*

General Terms

Scheduling, Real-Time Systems

Keywords

Scheduling, Real-Time Systems

1. INTRODUCTION

Real-Time Systems are mostly control-command systems that must satisfy both algorithmic correctness and specific time constraints. Real-Time scheduling focus on satisfying deadlines, and real-time validation on deciding whether the system can satisfy or not the time constraints. We deal with real-time validation, not with algorithmic validation.

Control-command systems must react to all incoming events. They are composed of a set of concurrent tasks $\left\{(\tau_i)_{i\in[1,n]}\right\}$ which may read signal values, compute how the system must react, and transmit engine activation signals. Each task is submitted to hard temporal constraints induced by the dynamic of the physical process: e.g. a late computed result, even if it is correctly computed, may be unexploitable because it is out-of-date. For that reason an operating system may host a real-time software if and only if it can guarantee that all deadlines are met. Such an operating system is called *Real-Time Operating System*: it is especially characterized by the use of specific scheduling policies [2].

Two approaches are commonly used in the litterature:

- **on-line scheduling:** a set of rules is used at run-time to chose the task to process among the pending tasks; several algorithms have been proposed in the litterature (e.g. RM, EDF [10]);
- **off-line scheduling:** a schedule is computed before run-time (either thanks to on-line scheduling policies like RM or EDF, or using a model-driven approach),and then must be followed by the dispatcher; such strategies are more powerfull than on-line strategies in the sense that they can produce valid schedules (i.e. for which all the time constraints are met) for a larger class of applications [7].

Motivation and Related works

Since the early sixties, many real-time scheduling policies have been proposed [10] [2] [7]. However, the real-time operating systems which may be used nowadays to host effective applications only propose fixed priority schedulers [13] [20]. Neither the other (more performant) on-line strategies nor the off-line strategies are implemented.

In [21], the Linux kernel is modified in order to guarantee the real-time constraints. It implements a priority driven scheduler within the kernel. In [16], the operating system structure is also modified, by the implementation of scheduling functions in both the hardware and the software. The proposed scheduling technique is also priority driven. This approach is extended in [19] and [5], where the proposed coprocessor is modelled in VDL.

A challenging issue for real-time systems would be to propose a methodology to implement scheduling strategies other than the native fixed-priority ones within a real-time kernel. Of course, such a methodology has to be validated. For that aim, some specific observation tools must be developed. Their definition is the aim of our paper.

Most of the time, temporal validation means the validation of the scheduling strategy. This is classically performed off-line, independently of the platform on which the application will run (see the *theoretical* side of Figure 1). It often relies on simulation. We are here interested in the actual behavior of the application. We want to verify that at run time, all the temporal constraints are actually met. This requires a further step after the validation of the scheduling strategy (see

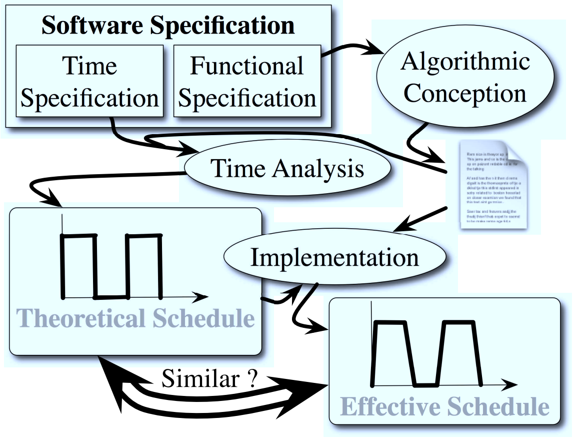

Figure 1: Theoretical/Effective scheduling validation.

the *effective* side of Figure 1), that consists in verifying that this strategy is correctly implemented within the scheduler, and thus, that the application behaves as expected, i.e. that its actual behaviour matches the schedule. This will rely on observation.We implement the program tracing system into the program itself [11][15].

Scope of the paper

The aim of this paper is to set up the basis for the design of an observation tool. Then we will present its implementation within the real-time development framework Xenomai which cooperates with the Linux kernel. And finally, we will illustrate its use through the observation of the native scheduling strategies.

Context

We adopt the classical modelling of tasks. We consider periodic hard real-time systems: for all i, τ_i is periodic and characterized by the following time attributes (see Fig. 2):

- r_i is the first release date;
- C_i is the worst case execution time;
- D_i is the relative deadline;
- T_i is the period.

Time 0 is defined as the first release date of the earliest released task. Tasks are assumed to be independent: they neither share resources nor exchange messages.

We use a PC architecture: a date is associated with each event, thanks to a clock called *real-time clock* in the sequel. We consider that the date values generated by the real-time clock match effective date values.

We use the real-time development framework Xenomai [18] for the following reasons:

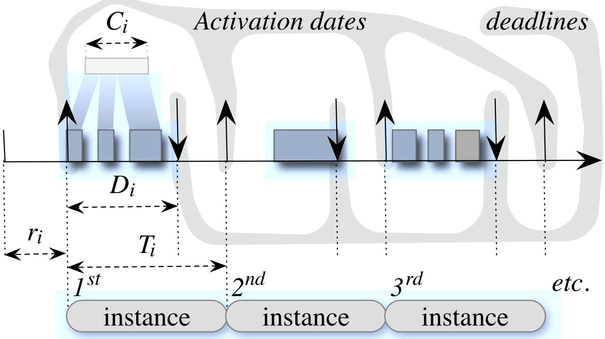

Figure 2: Time characteristics of task τ_i.

1. Xenomai stands on an open source operating system, what is required since we plan to modify the kernel of the operating system;
2. Xenomai runs on a PC architecture;
3. the system is alive: there are active users and a developer community, regular new versions, etc.

For these reasons, systems like LynxOS [17], QNX [8], RTLinux [20] could not be used whereas Xenomai [18] satisfies all requirements. This justifies our choice.

2. REAL-TIME SCHEDULING

2.1 Schedules

2.1.1 Theoretical aspects

A task owns the processor between two consecutive *context switchs*. The time interval between two consecutive context switchs is constant, it is called *quantum*. The quantum is a multiple of the period of the real-time clock of the computer. A *schedule* σ is a sequence of tasks that successively own the processor. $\sigma_t = \tau_i$ means that τ_i owns the processor from time $(t-1) \times q$ to time $t \times q$.

A schedule σ is *cyclic* with period P if $\exists t_0 \in \mathbb{R}^+$ such that $t \geq t_0 \Rightarrow \sigma_{t+P} = \sigma_t$.

2.1.2 Concrete aspects

The real-time clock regularly sends a signal, that increments a register that every program may read. The different times are computed from the start time of the system (time 0). Hence we represent the time by the set $\mathbb{N}$, and time=t ($t \in \mathbb{N}$) means t *clock cycles after starting the computer*. In the sequel, we define our time scale by a translation of the clock: time 0 is the start time of the real-time software.

The function $X : \mathbb{N} \rightarrow S$ gives the history of the operating system. Its graph is a set of pairs of the form $(t, X(t))$.

EXAMPLE 1. *Figure 3 presents a process history and the corresponding graph: at the beginning the system is in the state S_0 (the scheduler launches τ_1), as at time* 30.

In the sequel, the *Theoretical* (resp *Effective*) *schedule* corresponds to the theoretical (resp. effective) analysis of the real-time software.

Date	Event
0	$Start\,(\tau_1)$
10	$End\,(\tau_1)$
30	$Start\,(\tau_1)$
40	$End\,(\tau_1)$

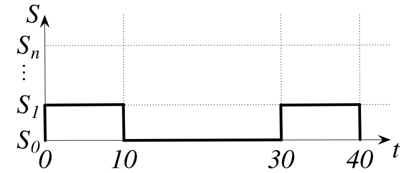

Figure 3: Graph of X for Example 1.

3. THE SCHEDULER OBSERVER

3.1 Why to observe a schedule

We have presented in §2.1 theoretical aspects of scheduling. Based on these concepts, one can validate a schedule (off-line) or a scheduling policy (on-line) in a *theoretical* way.

Theoretical means that the validation relies on the assumption that there is no difference between the theoretical and the effective schedules (matching assumption that the operating system follows the schedule very precisely). Now, the validation of the scheduler requires either to prove this assumption, or to give it up and to reason on the effective schedule rather on the theoretical one. To evaluate how far the operating system is from this matching assumption, we need to collect effective times corresponding to context switches. For that aim, we have developed a specific component, that we have called *scheduler observer*.

The role of a scheduler observer is summarized in Figure 4 [9] [6]. The temporal part of the requirements leads to the theoretical analysis of the software, that produces the theoretical behaviour(s) X of the software, and then the diagnosis of validity. Following the path *Conception* → *Implementation*, we obtain the effective software, whose execution produces the effective behaviour $\widehat{X}$.

Definition 1. *A* scheduler observer *is a component which models an effective system in order to produce an estimation $\bar{X}$ of its effective behaviour $\widehat{X}$.*

Our concern is to measure the *distance* between $\widehat{X}$ and X, which quantifies the quality of the observer. This distance may be estimated thanks to the specifications of both the real-time clock and the associated operating system functions. Thanks to the observer, we produce an estimation $\bar{X}$ of $\widehat{X}$: we have

$$distance\left(X,\widehat{X}\right) \le distance\left(X,\bar{X}\right) + distance\left(\bar{X},\widehat{X}\right)$$

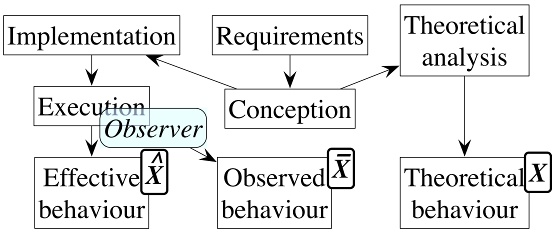

Figure 4: What a *scheduler observer* produces.

3.2 How to observe a schedule

An observer produces a sequence of pairs $(t, A(t))$, where t is a real-time clock value and $A(t)$ the action which has been observed at time t.

The observation process involves three objects: the observed system, the hardware it is running on, and the observer itself. The observer runs on the same computer that the observed system. The system, the hardware and the observer are characterized by specific attributes.

The observed system

- The **class** describes the characteristic of the real-time system (hard, soft, periodic, syncronous, etc.) [7].
- The **life** is the time interval on which the observation process must run. For our examples, we observe it along the loading period and one cyclic period, that are computed following [3].
- The **time scale** s.scale is the time unit: it specifies the smallest time interval between two consecutive events in the life of the system.

The hardware

- The **architecture** describes the characteristics of the target (uniprocessor, multiprocessor, etc.) [7].
- The **computer cycle** is the time interval between two consecutive real-time clock signals.

The observer

- Statements are embedded in the software. They collect the pairs $(t, X(t))$ that compose $\bar{X}$.

Definition 2. $\bar{X}$ *is an* accurate *view of* $\widehat{X}$ *if* $\forall t \in \mathbb{N}$, $\exists t^{'} \in \mathbb{N}$ *such that* $\left|t - t^{'}\right| \le S.scale \wedge \bar{X}\left(t^{'}\right) = \widehat{X}\left(t^{'}\right)$

So $\bar{X}$ is an *accurate* view of $\widehat{X}$ if at each time t, $\widehat{X}$ matches the value $\widehat{X}(t)$ for a time $t^{'}$ *near* from t: *near* means *less that s.scale*. Hence both graphs are the sames, if values are approximated to the nearest s.scale multiple.

We note this property $\bar{X} = \widehat{X}$. The view $\bar{X}$ obtained thanks to an observer Ω is noted $\bar{X}_\Omega$ if specifying Ω is required, $\bar{X}$ if there is no ambiguity.

Definition 3. *An observer Ω is* adequate *for a system S running on a hardware H if and only if $\widehat{X} = \bar{X}_\Omega$.*

3.3 Implementation

We implement the observer as a functionnaly-empty version of the program itself. Figure 5 presents the way this is performed. The events Start, Exec and End are explicit; the events Suspend and Reload are implicitely deduced from the context. This technique guarantees $\widehat{X} = \bar{X}$, since the program **is** the observer.

```
void τi() {
  do { /* Tracing τi's activation */
     WRITE(τi.Start) ;
    /* Body */
     for(j=1; j≤Ci; j+=S.scale) {
       OwnCPU(S.scale);
       WRITE(τi.Exec) ;
     }
    /* Tracing τi's completion */
     WRITE(τi.End) ;
    /* Waiting for τi's next activation */
     P(τi.Activ) ;
  } while (0==0);
}
```

Figure 5: Program body for τ_i.

3.3.1 Calibration

Running the observer on the same computer (and with real-time priority) impacts the time behaviour of the software (writing, disk buffering and storage, etc.). These operations may also involve noise at the operating system level (memory caching, external process loading/suspending, etc.). We have to evaluate how these effects impact $\widehat{X}$ and/or $\bar{X}$. This is achieved by means of a *Process Calibration*, that consists in associating an estimated duration to all statements added to the program, due to the observer.

$\mathbb{N}$ represents the sequence of dates corresponding to the computer real-time clock signals, hence all statements (e.g. clock signals) may occur only at integer dates. The number of *ticks* of the real-time clock which are associated with the time unit depends on the frequency of the real-time clock.

To evaluate this time, we execute N times[1] the code (see Figure 5) with no observer statement, and we get the total amount of time A_0 dedicated to its completion. Next, we replay the sequence with one observer statement s_1 (e.g. WRITE(τ_i.Start)). We again get the total amount of time A_1 which is greater than A_0. The difference $A_1 - A_0$ corresponds to $N \times Duration\,(s_1)$. This process is successively reproduced for all observer statements. Once all observer statements integrated, we have a sequence of computed times A_0, A_1, $\ldots A_k$ (k is the number of observer statements that had been integrated) such that $A_0 < A_1 < \ldots < A_k$. The value $B = \frac{A_s - A_{s-1}}{N}$ is an estimation of the amount of time dedicated to the s^{th} observer statement integrated into the software during each execution of τ_i. Using *S.scale*, we compute the time used for observation per time unit: $B_s = \frac{A_s - A_{s-1}}{NT_i}$ (T_i is the period of τ_i, expressed as a multiple of the time unit *S.scale*). Moreover, statement duration analysis shows that the observed values follow Gaussian distributions, hence the average value is the more accurate [4]. Hence to cancel this loss of time from observations, we modify the scale of the system thanks to $S.scale := S.scale - \underset{\tau_i}{Average}\, B_u$.

3.3.2 Quality of effective schedules

The theory defines X as the ideal behaviour that the implementation can produce. We observe $\bar{X}$. On the structural level, X and $\bar{X}$ are lists of pairs of the form $(t, A\,(t))$. The ideal $X = \bar{X}$ corresponds to $\forall t \in S.Life$, $(t, A\,(t)) \in X \Leftrightarrow (t, A\,(t)) \in \bar{X}$. If this ideal is not reached, then there exists $t \in S.Life$ such that $(t, A\,(t)) \in X \setminus \bar{X} \cup \bar{X} \setminus X$. The more frequently this situation appears, the worst is the quality of the implementation. The *Hamming distance* [12] has been designed to quantify the differences between vectors. It is defined in the following way: let $A = (a_i)_{i \in [1,n]}$ and $B = (b_i)_{i \in [1,n]}$ be vectors of size n, the Hamming distance $d\,(A, B)$ is equal to $|\{i \in [1,n] \text{ such that } a_i \neq b_i\}|$.

Therefore $d\,(A, B)$ is bounded by 0 ($A = B$) and n (there is no i such that a_i matches b_i). In our context, we prefer to use a percentage-based quality indicator, where 0% means *bad* and 100% means *perfect*.

Definition 4. *We call* similarity level *of an observed schedule* $\bar{X}$ *the value* $\delta\left(X, \bar{X}\right) = \frac{S.Life - d\left(X, \bar{X}\right)}{S.Life}$, *where* $d\,(\bullet, \bullet)$ *is the Hamming distance.*

3.3.3 Xenomai implementation of the observer

The Linux kernel is not a real-time operating system. To get a real-time Linux, one may modify the non-preemptive kernel[2] into a preemtive one. The alternative is to enrich the Linux kernel with a second kernel, named *co-kernel*. This co-kernel is a real-time kernel.

The Xenomai implementation is based on the co-kernel approach, which is implemented using the ADEOS patch [1]. ADEOS stands on the concept of *domain*, that embeds a set of processes supposed to share the same criticity level. For the Xenomai implementation, three domains have been defined (see Figure 6):

1. the Xenomai domain, also called primary domain;
2. the Linux domain, also called secondary domain;
3. the interrupt shield domain, which is an intermediate domain between the two others.

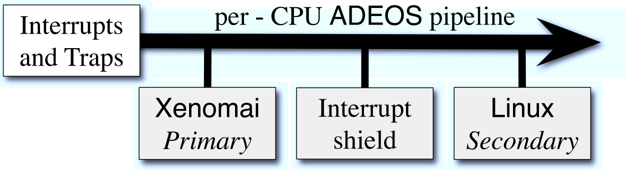

Figure 6: ADEOS domain organization.

We have experimented using the native Xenomai API (v. 2.5.6). Statements like start() and end() have been developed thanks to both the Linux input/output procedures and the Xenomai specific system primitives. The Figure 7 presents the way τ_i is implemented, Figure 8 describes the programming details for the functions start() and end(). The others observer statements (e.g.busy_cpu) are developed following the same approach.

[1] N is an arbitrary great number (10^6 here).

[2] A Linux kernel process can not be preempted [20].

```
rt_task_set_periodic(NULL, TM_NOW, T[0]*E);
while(0==0) {
 start();
 for(i=0;  i<C[0];  i++)
   busy_cpu();
   end();
   rt_task_wait_period(NULL);
}
```

Figure 7: Use of Xenomai real-time primitives by τ_i.

```
extern sem un_sem ;
extern int evt_nr ;
void start(RT_TASK_INFO *task_i) {
 rt_sem_p(&un_sem, TM_INFINITE);
 observation[evt_nr].begin=
      rt_timer_tick2ns(rt_timer_read());
 rt_task_inquire(rt_task_self(), task_i);
 observation[evt_nr].transition='start';
 observation[evt_nr].task=task_i->name;
 observation[evt_nr].end=
      rt_timer_tick2ns(rt_timer_read());
 evt_nr++;
 rt_sem_v(&un_sem);}
void rt_task_end(RT_TASK_INFO *task_i){
 rt_sem_p(&un_sem, TM_INFINITE);
 observation[evt_nr].begin=
      rt_timer_tick2ns(rt_timer_read());
 rt_task_inquire(rt_task_self(), task_i);
 observation[evt_nr].transition='end';
 observation[evt_nr].task=task_i->name;
 observation[evt_nr].end=
      rt_timer_tick2ns(rt_timer_read());
 evt_nr++;
 rt_sem_v(&un_sem);}
```

Figure 8: The start() and end() function bodies.

4. RESULTS

The computer experimentations have been performed on an Intel Pentium 4 whose real-time/ clock frequency is 2791.44 MHz. The real-time clock period is 1ns. The memory size is 491336 MB, and the hard disk size 31 GB. The cache memory is disabled for experimentations.

Figure 9 presents the values observed for completing the calibration process. The scale of the system (i.e. the scheduler quantum) is set to 1ms, hence $\frac{observer\ primitive}{quantum} < 0.1\%$. We have experimented on the following task system:

Primitive	Execution time
No primitive	46 ns
rt_sem_p	130 ns
rt_sem_v	157 ns
rt_timer_read	42 ns
rt_task_inquire	205 ns
start	576 ns
end	576 ns

Figure 9: Average execution time of Xenomai primitives and observer functions.

Evt	Start / End (ns)
S τ_1	0 / 1306
Q τ_1	1961 / 1002521
E τ_1	1003100 / 1003404
S τ_2	1006459 / 1006818
Q τ_2	1007211 / 2007601
Q τ_2	2008002 / 3008339
Q τ_2	3008672 / 4008989
⋮	⋮
Q τ_2	67012972 / 68013271
E τ_2	68013637 / 68013908
S τ_3	68015461 / 68015794
Q τ_3	68016191 / 69016595
E τ_3	69017012 / 69017290

Legend
S $\langle\tau\rangle$ Start task τ
Q $\langle\tau\rangle$ Quantum for task τ
E $\langle\tau\rangle$ End task τ

Figure 10: Observation results.

Times are in ms.

Task	r	C	D	T	Priority
τ_1	0	10	50	50	20
τ_2	10	30	70	70	14
τ_3	40	10	100	100	10

The scheduling policy is *Rate Monotonic* ($Priority\,(\tau_i) < Priority\,(\tau_j) \Leftrightarrow T_i > T_j$). We have observed the system on an hyperperiod [3]. The results[3] are presented in Figure 10. Considering the approximations involved by the scale (1ms), we have $\delta\left(X, \bar{X}\right) = 100\%$, hence the effective scheduling matches the theoretical scheduling.

5. CONCLUSION

We have defined a methodology to observe the effective schedules produced by real-time schedulers. The *similarity level* proposed in §3.3.2 enables us to evaluate the quality of a specific scheduler implementation and to compare different implementations of the same scheduling policy. A tool has been developed and validated by means of exprimentations.

This methodology will be helpful for implementing specific (on-line and/or off-line) policies into real-time kernels: we will be able to evaluate scheduler implementations relatively to their theoretical behaviour, and also to compare different implementations of a specific scheduling policy. We plan to address static scheduling, that is classically used for real-time systems. Dynamic scheduling may also be planed [14].

The next step of our research is the implementation of scheduling policies not yet implemented into real-time kernels. This research is ongoing.

6. REFERENCES

[1] http://gna.org/projects/adeos. web site, october 2011.

[2] G.C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Kluwer Academic Publishers, Boston, USA, 2000.

[3] An extracted sequence only !

[3] A. Choquet-Geniet and E. Grolleau. Minimal schedulability interval for real-time systems of periodic tasks with offsets. *Theoretical Computer Science*, 310:117–134, 2004.

[4] A. Colin, I. Puaut, C. Rochange, and P. Sainrat. Calcul de majorants de pire temps d'exécution: état de l'art. *Technique et Science Informatiques*, 22(5):651–677, 2003.

[5] C.M. Ferreira and A.S.R. Oliveira. *RTOS Hardware Coprocessor Implementation in VHDL*, chapter RTOS Hardware Coprocessor Implementation in VHDL, pages 6–11. Embedded Systems Conference, 2009.

[6] J.L. Hellerstein, D.M. Tilbury, and S. Parekh. *Feedback Control of Computing Systems*. John Wiley and Sons, 2004.

[7] J. Carpenter S. Funk P. Holman, A. Srinivasan J. Anderson, and S. Baruah. *A Categorization of Real-Time Multiprocessor Scheduling Problems and Algorithms*, chapter Handbook of Scheduling: Algorithms, Models, and Performance Analysis, pages 30–1—30–19. Chapman and Hall/CRC, 2004.

[8] R. Krten and C. Herborth. *The QNX CookBook*. PARSE Software Devices, 2003.

[9] W.S. Levine. *The Control Handbook*. CRC Press, 1996.

[10] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[11] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the* 2005 *ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 40, pages 190–200. ACM, June 2005.

[12] F.J. MacWilliams and N.J.A. Sloane. *The Theory of Error-Correcting Codes*, volume 1. North-Holland Mathematical Library, 1977.

[13] R. Mall. *Real-Time Systems: Theory and Practice*. Dorling Kindersley, 2007.

[14] T. Megel, D. Chabrol, V. David, and C. Fraboul. Dynamic scheduling of real-time tasks on multicore architectures. In *Colloque du GdR Soc/SiP*, Orsay, France, June 2009. CEA. http://hal-cea.archives-ouvertes.fr/docs/00/45/12/84/PDF/GDR-SOCSIP.pdf.

[15] T. Moseley, N. Vachharajani, and W. Jalby. Hardware performance monitoring for the rest of us: a position and survey. In *Proceedings of the* 8th *IFIP International Conference on Network and Parallel Computing*, volume 6985 of *Hardware performance monitoring for the rest of us: a position and survey*, pages 293–312, Changsha, China, 2011. Springer-Verlag.

[16] T. Nakano, A. Utama, M. Itabashi, A. Shiomi, and M. Imai. Hardware implementation of a real-time operating system. In *Proc. of the The* 12th *TRON Project International Symposium*, pages 34–42, Tokyo, Japan, December 1995. IEEE Computer Society.

[17] L.M. Surhone, MM.T. Tennoe, and S.F. Henssonow. *LynxOS*. VDM Verlag Dr. Mueller AG & Co. Kg, 2010.

[18] L.M. Surhone, M.T. Tennoe, and SS.F. Henssonow. *Xenomai*. VDM Publishing House, 2010.

[19] M. Vetromille, L. Ost, C.A. Marcon, C. Reif, and F. Hessel. Rtos scheduler implementation in hardware and software for real time applications. In *Proc. of the* 7th *IEEE International Workshop on Rapid System Prototyping*, pages 163–168, Chania, Crete, Greece, June 2006. IEEE Computer Society.

[20] K. Yaghmour, J. Masters, G. Ben-Yossef, and P. Gerum. *Building Embedded Linux Systems*. O'Reilly Media, 2008.

[21] V. Yodaiken and M. Barabanov. A real-time linux. *Linux Journal*, 34:5, 1997.