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 $\{t_i\}_{i \in [1,n]}$ 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the physical process: e.g. a late computed result, even if it is correctly computed, may be unexploitable because it is out-of-date.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 offline, 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
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].

Figure 1: Theoretical/Effective scheduling validation.

Scope of the paper
The aim of this paper is to set up the basis for the design of the program itself [11][15]. We implement the program tracing system into the program itself [11][15]. 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 $i$, $\tau_i$ 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:

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 switches. The time interval between two consecutive context switches is constant, it is called quantum. The quantum is a multiple of the period of the real-time clock of the computer. A schedule $\sigma$ is a sequence of tasks that successively own the processor. $\sigma_i = \tau_i$ means that $\tau_i$ owns the processor from time $t_0$ to $t_0 + q$.

A schedule $\sigma$ 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).

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 $\tau_1$), as at time 30.

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

The observed system

- The class describes the characteristic of the real-time system (hard, soft, periodic, synchronous, 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 \(\hat{X}\).

Definition 2. \(\hat{X}\) is an accurate view of \(\hat{X}\) if \(\forall t \in \mathbb{N}, \exists t' \in \mathbb{N} \text{ such that } |t - t'| \leq \text{s.scale} \land \hat{X}(t') = \hat{X}(t')\)

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

We note this property \(\hat{X} = \hat{X}\). The view \(\hat{X}\) obtained thanks to an observer \(\hat{\Omega}\) is noted \(\hat{X}\) if specifying \(\hat{\Omega}\) is required, \(\hat{X}\) if there is no ambiguity.

Definition 3. An observer \(\hat{\Omega}\) is adequate for a system \(S\) running on a hardware \(H\) if and only if \(\hat{X} = \hat{X}\).

3.3 Implementation

We implement the observer as a functionally-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 implicitly deduced from the context. This technique guarantees \(\hat{X} = \hat{X}\), since the program is the observer.
The Linux kernel is a real-time kernel. Therefore, no
\( A_i \) correspond to \( N \times Duration(s) \). 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 \( B = \frac{A_k - A_0}{N} \) is an estimation of the amount of time dedicated to the \( s \)th observer statement integrated into the software during each execution of \( \tau_i \). Using \( S.scale \), we compute the time used for observation per time unit: \( B_x = \frac{A_k - A_0}{N} \) (\( T_x \) is the period of \( \tau_i \), 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 \( S.scale := S.scale - \text{Average} B_x \).

### 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 preemptive 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 \( \text{start()} \) and \( \text{end()} \) have been developed thanks to both the Linux input/output procedures and the Xenomai specific system primitives. The Figure 7 presents the way \( \tau_i \) is implemented, Figure 8 describes the programming details for the functions \( \text{start()} \) and \( \text{end()} \). The others observer statements (e.g. \( \text{busy.cpu} \)) are developed following the same approach.

\(^2\)A Linux kernel process can not be preempted [20].
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 1ms. 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 $\frac{\text{observer primitive quantum}}{\text{quantum}} < 0.1\%$. We have experimented on the following task system:

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>No primitive</td>
<td>40 ns</td>
</tr>
<tr>
<td>rt_sem_p</td>
<td>130 ns</td>
</tr>
<tr>
<td>rt_sem_v</td>
<td>157 ns</td>
</tr>
<tr>
<td>rt_timer_read</td>
<td>42 ns</td>
</tr>
<tr>
<td>rt_task_inquire</td>
<td>205 ns</td>
</tr>
<tr>
<td>start</td>
<td>576 ns</td>
</tr>
<tr>
<td>end</td>
<td>576 ns</td>
</tr>
</tbody>
</table>

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

The scheduling policy is Rate Monotonic ($\text{Priority}(\tau_i) < \text{Priority}(\tau_j) \Leftrightarrow T_i > T_j$). 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 $\delta(X, \hat{X}) = 100\%$, 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 §3.2 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 experimentations. 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 planned [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


[3] An extracted sequence only!


