# Supporting Real Time Extensions in an Embedded Exo-Kernel

Damien Deville Alexandre Courbot Gilles Grimaud
{deville,courbot,grimaud}@lifl.fr http://www.lifl.fr/~{deville,courbot,grimaud}
*Université des Sciences et Technologies de Lille, LIFL, Bât. M3,*
*Cité Scientifique, 59655 Villeneuve d'Ascq - France*

## Abstract

*Smart cards are massively deployed small portable devices which focus on security (*$10^9$ *units sold mainly in Asia and Europe). To increase the capability of the smart card software to support more and more services, smart card operating systems have evolved from "dedicated monolithic execution platforms" to more "standard" operating system architectures that support dynamic code loading. This paper presents real time issues in a particular smart card operating system named Camille which support features like: dynamic code loading, on board type verification, on board just in time compilation, and operating system component dynamic loading. More generally, it discusses on the difficulties to support real time extensions according to the exo-kernel principles.*

## Introduction

A smart card is a device with major hardware constraints: low-power CPU, low throughput serial I/O, little memory (typically $1-4$ kb RAM, $32-128$ kb ROM and $16-64$ kb Flash RAM). But its user friendly principle and tamper resistance makes it one of the mobile computing devices of choice.

This paper presents the work led on the Camille RT exo-kernel. In order to sense the context in which our work took place, Section 1 includes an overview of the hardware capabilities of the smart card. Section 2 presents the motivations for the Camille and Camille RT smart card operating system. Finally, the remaining Sections present solutions used to support real time in the Camille exo-kernel.

## 1. Smart card context

Smart cards are small, portable and secured device which nowadays allows post issuance (loading application on card after the card being emitted). Nevertheless, smart cards have some hardware limitations due to ISO [15] constraints that are mainly defined to enforce smart card tamper resistance [16].

### 1.1. Microprocessors

A wide class of microprocessors are used from old 8-bit CISC micro chip (4.44 Mhz) to powerful 32-bit RISC (100 to 200 Mhz). The type of CPU used for smart card is highly influenced by the ISO [15] constraints linked to the card. Historically, smart card manufacturers used 8-bit processors because operating systems and applications code are more compact. But smart cards now need to be more efficient and new embedded applications require more and more computing power. So card designers sometimes choose 32-bit RISC processors (or enhanced 8/16 bits CISC).

| Model | Architecture | Data BUS size | Registers | Frequency |
|---|---|---|---|---|
| 68H05 | CISC | 8 bits | 2 (8 bits) | 4.77 Mhz |
| AVR | RISC/CISC | 8 bits | 32 (8/16 bits) | 4.77 Mhz |
| ARM7TI | RISC | 32 bits | 16 (32 bits) | 4.77 to 28.16 Mhz |
| R4KSC | RISC | 32 bits | 32 (32 bits) | 4,77 Mhz to 100 Mhz |

**Table 1. Characteristics of common smart card microprocessors.**

### 1.2. Memories

Different types of memory exist on the card. The first one is the RAM (Random Access Memory); there is also some ROM (Read Only Memory); and finally EEPROM (Electric Erasable Programmable Read Only Memory) or FLASHRAM that are writable persistent memories. Because smart card silicon is supposed to be limited to 27 $mm^2$, the physical space needed for storing 1 bit is an important factor. A typical smart card dispose of 2-4KB of working memory, 32-128KB of persistent memory and 64-128KB of ROM.

Persistent memory has a major drawback, linked to its electronic properties. Its writing delay is up to 10000 times slower than RAM one. Furthermore, writing in persistent memory may damage the memory cells (this is called stress).

### 1.3. I/O

Smart card manufacturers have provided ISO normalization, defining I/O protocols. The wired normalization is called ISO 7816 and is declined in "T=x" protocols. The wireless normalization (for contact less smart card) is defined in ISO SC17 14443. Nevertheless some Smart Card prototypes have used more conventional protocols and physical links. "T=0" and "T=1" are the most used protocols by the smart card industry. They provide 9600 to 192000 bauds using a half duplex serial line (This rate means that 1 KB of data is transferred in less than 1 second). Implementation of I/O protocols introduces a lot of real time constraints. In fact, ISO 7816-3&4 define deadline for Answer To Reset (ATR, *i.e.* the byte stream produced by the card when the power is switched on). Other deadlines exist for delivering every answer to a question from the terminal. Smart card operating systems clearly have RT deadlines.

## 2. From open smart card operating systems to Camille

Since birth of smart cards in the early eighties, smart card software architecture has never stopped evolving. During this period, numerous standards have been proposed. They are known under the name ISO7816-x, and rule nearly all smart cards features from physical characteristics to application management.

There are five parties involved in the life cycle of the smart card. *Semiconductor manufacturers* are in charge of chip design and mass production. *Smart card manufacturers* (associated in this paper with smart card software producers) embed issuers' requirements. *Card issuers* traditionally have more business/behavioral considerations while deploying and managing smart card-based solutions. *Service providers* design and deploy (under the control of issuers) value-added services and *Users* benefit from those services.

The smart card software life cycle is composed of three steps. First, software is *produced* and *loaded* (*i.e.* embedded in the chip). Then, depending on its nature (ready-to-run or ready-to-load appli-

| Type | memory point | capacity | write time | page size |
|---|---|---|---|---|
| ROM | reference | 32-128 KB | read only | 1 byte |
| FlashRAM | x 2-3 | 16-64 KB | 2.5ms | 64 bytes |
| EEPROM | x 4 | 4-64 KB | 4ms | 2-64 bytes |
| RAM | x 20 | 128-4096 B | $\leq 0.2\mu s$ | 1 byte |

**Table 2. Card memory characteristics**

cations), this software is *initialized* or *instanced*. Last, embedded software is *operated* in client/server applications.

Smart card operating systems have evolved from *first-generation* smart card software architecture (a monolith given by smart card manufacturers to semiconductor manufacturer that meets client requirements and chip specifications) to open operating systems that allow dynamic code loading after the emission of the card to the end user (this is known as post-issuance). This trends allow end users to customize the applications to best fit their needs. These *ready-to-load application* platforms usually rely on a virtual machine, both for portability (a single application can be loaded into several different cards, *i.e.* relying on different hardware, without needing to be modified) and for security (it is usually easier to prove or ensure the safety with intermediate codes). Figure 1 describes the architecture of two of these OS: Multos[1] (a virtual machine supporting Mel byte codes) and a Java Card[1] (a dedicated Java configuration designed to suit best the smart card constraints) with a SIM support. Multos OS is mainly used in banking context like in MASTER CARD. Java SIM Card is used in GSM cell phones.

Multos and Java Card support dynamic code loading to provide multi-application services at the application level. Both of these OS are provided a particular application which is very close to the OS. Multos support a banking core management appli-

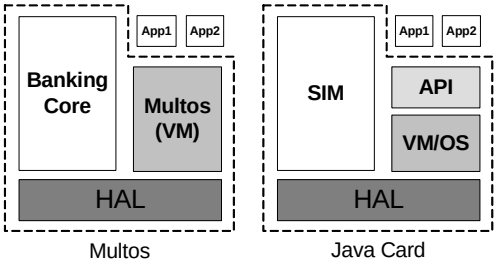


**Figure 1. Extensibility in open smart card operating systems**

cation used by MASTER CARD, Java SIM Card support an application named SIM. The SIM application (*i.e.* Subscriber Identity Module) is the bridge between the card and the digital devices of the phone that manage the communication. The most important treatment for the SIM card is to produce and deliver a cryptographic key periodically (used for authentication), otherwise the BSC (Base Station Controller which is part of the infrastructure of the GSM network) stop the communication. Both of these applications are so close to the OS that a programmer could not develop them like standard application using Mel or Java language. This means that it is impossible to update the code of these applications. Extensibility is only supported at the "user" application level. On Figure 1 the dashed line is the cross-border between what is extensible and what is loaded once by the smart card manufacturer.

The major drawback of these two OS is that they only support applications written in

[1]Maosco Ltd., `http://www.multos.com`.

their language over the abstraction they provide. Camille is the first prototype in "open" operating systems for smart card which support loading of new hardware abstractions.

### 2.1. Camille

Camille aims at supporting the various hardware resources used in smart cards. Memory pages, numerical values, microprocessor, and native code can easily be manipulated by applications while ensuring a high level of security. The Camille design approach can be compared to the MIT *Exo-Kernel* [11, 12] with identical principles and concepts (*i.e.* no hardware abstractions but a secure (de)multiplexed access). Embedded code is expressed using a dedicated intermediate language called FAÇADE [14], but source code can be written using several widespread languages. For instance, C code can be compiled into FAÇADE using GCC.

The Camille OS provides the following three basic characteristics. *Portability* is inherited from the use of an intermediate code and by a limited set of hardware primitives. *Security* is ensured by a code-safety checking (which uses a PCC-like algorithm [21]) at loading time [23]. *Extensibility* is provided through a simple representation of the hardware that at the *root* of the system does not predefine any abstractions. Thus, applications have to build or import abstractions which match their requirements. The Camille splitted architecture is described Figure 2.

The usual downside of extensibility is performance. For some parts of the OS that require efficiency, Camille uses *Just-in-Time* techniques to compile intermediate code into native one. Increased performance also comes from the exo-kernel approach that does not introduce abstraction penalties in the core of the OS. Because smart cards have limited computing power, additional hardware independent optimizations are also performed out of the card, while the source code is translated to FAÇADE. A more precise description of Camille, and experimental results as well can be found in [7]. The Camille prototype demonstrates the feasibility of an extensible smart card OS that has reasonable footprint: 17 KB of native code in which 3.5 KB for code verification, 8.5 KB for native code generation, and the rest for hardware multiplexing.

### 2.2. New problem

The main problem with Multos and Java SIM is linked to the particular application they support. SIM and Banking Core are applications but are so close to the operating system and to the virtual machine that an application developer would not be able to write them using the standard application development scheme and tools. For example, SIM provides context commutation to ensure the respect of the deadline linked to cryptographic key generation. Such a feature would be difficult to implement using Java Card language, API or abstractions, and is so close to the virtual machine inner loop that can be consider as a part of the operating system.

Camille would be a solution to such problems. Supporting loading of Java Card application written in Java Card is possible and has already been done; Another solution is to design and implement a Java Card virtual machine on top of Camille *Just-In-Time* compiler using the minimalist level of expressiveness of the Camille exo-kernel. Same process can be done for supporting Multos applications within Camille.

The remainder of this paper is about our work on real time issues in the context of the Camille exo-kernel focusing on two points: on-card WCET computation in the Camille operating system and architectural design of a real-time schedulers in exo-kernels.

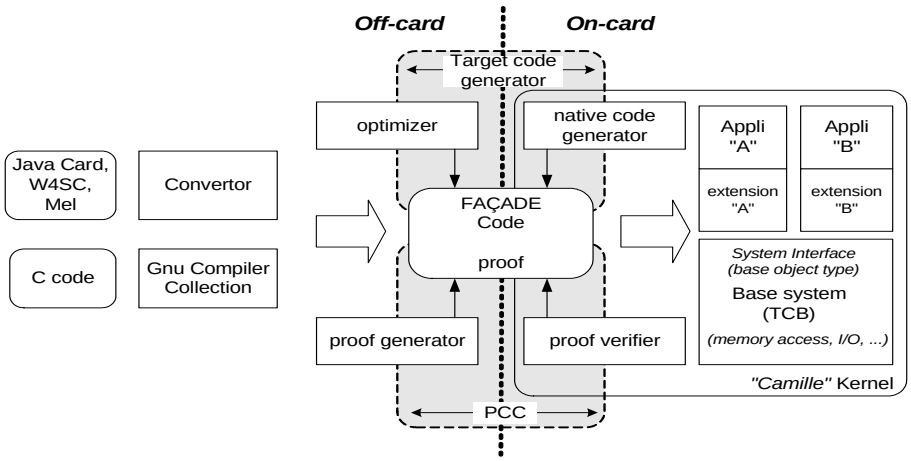


**Figure 2. Camille architecture and software infrastructure**

## 3. Camille RT

The standard exo-kernel architecture, as it has been defined by the MIT, does not offer any dedicated real time primitives neither to applications nor to extensions. The exo-kernel is designed to support extensibility while ensuring maximum security. It just offers "equity" for the access to the processor to each "system" [2] [11]. For each cpu time slice the exo-kernel uses a "round-robin" policy to elect an application (A *yield* primitive is given to extensions which want offering the remaining time of a slot to others). The main motivation in Camille RT is to offer to user applications and to system extensions the capability to implement real time primitives, and to make standard applications coexist with real time applications.

The key points to support real time in an extensible operating system bases on exo-kernel architecture are:

**(*i*)** quantify the execution time for each of the exo-kernel primitives (*e.g.*, delay related to a virtual TLB access),

**(*ii*)** find a schedule that succeeds all running applications deadlines,

**(*iii*)** define a way to allow applications to notify their real time requirements (deadline, rate, start time, ...) to the exo-kernel,

**(*iv*)** quantify the execution time for each of the RT code expressed in FAÇADE intermediate language.

The first three points concern the real time problematic in a standard exo-kernel, the last one is specific to Camille as it uses an intermediate language to increase the portability. Next Subsection focuses on on-card WCET computation, and the remainder of this article on the scheduling problem in a standard exo-kernel.

### 3.1. On-Card WCET Computation

We place ourselves in the context of hard real-time systems, i.e. the deadlines are critic and the system might not continue to run if even a single one is not met. In these conditions, it is imperative to know the Worst Case Execution Time (or, at worst, an execution time that is known to be superior to the actual WCET) of the RT code to be run. This is commonly done by performing a *static analysis* of the task's code, that is, analyzing the code to make a pessimist estimation of its WCET. In practice, the WCET

[2] extension in the exo-kernel architecture.

obtained by static analysis will always be superior to the actual WCET, meaning that we can ensure that the deadlines will be met if the scheduler gives the task at least this time to finish. On the other hand, static analysis techniques are often very pessimistic, which leads to a waste of the resources consumed by the difference between the actual WCET and the computed WCET. For this reason, many methods have been developed to reduce the pessimism by accurately simulating the architecture the program will run on [2, 3, 4].

Other methods to calculate the WCET are also known as *dynamic analysis*, however they are not suitable for our work: being based on runs of the task with particular input data that are supposed to make it last as long as possible, they cannot completely guarantee that the given time is superior to the actual WCET of the task. For this reason, next sections will only concentrate on static analysis methods.

#### 3.1.1. RT and downloaded code

Dealing with real time constraints requires the knowledge of execution time for every RT code. As Camille is an open extensible operating system, applications are converted to or written in an intermediate language named FAÇADE . In its current form, it offers extensibility at both user and OS level, but does not support WCET computation. Algorithms for WCET [4] and languages allowing easy WCET computation are well known [5, 24] but have the drawbacks of dramatically constraining application programming. We then have to propose a FAÇADE sub-model, more restrictive, used for developing real time tasks and extensions and use the existing one for generic applications. Thus standard and real time applications can coexist in Camille. WCET computation need to be performed in the card for different reasons. The main one is that FAÇADE is compiled "on the fly" by the card, thus only the card exactly knows the time needed for executing each FAÇADE instruction because it depends on the physical platform. Computing WCET outside the card would be possible by exporting a "chip profile" containing the exact code generated by the on fly compiler and the cycle number for each CPU's instructions. Smart card manufacturers do not like these informations to be public for industrial and economical purpose such as keeping secret the detail of technologies of smart card chip, and also to prevent timing attack or DSA [20] of cryptographic softwares/hardwares.

#### 3.1.2. Overview of Static Analysis Techniques

There are three main static-analysis techniques that we will quickly explain here:

- *Tree-Based* analysis,
- *Path-Based* analysis,
- *Implicit Path Enumeration Technique*.

*Tree-Based analysis* [3] uses the syntaxic tree of the program (built from the source code by the compiler) to get a detailed view of the program's structure. This method give good results and allow optimizations that reduce the pessimism of the WCET thanks to the good knowledge of the program's structure. However, it requires a part of the analysis to be done at the source code level. Moreover, a link must be done between the syntaxic tree and the basic blocks of the assembler code that is generated by the compiler, which isn't always trivial since the code's structure might be changed during compilation, most likely during the optimization phase. Still, there are solutions that address this last issue [9]. This analysis is not suitable for on-card WCET computation, since it requires some high level informations for the source code that we don't want to provide to the card due to the low amount of working memory and also due to the slow

I/O serial line. Moreover, the processing power it needs is far behind the capabilities of the card.

The two other static analysis techniques [10] need to have a representation of the control flow of the program. A *control-flow graph* is made of *basic blocks* (totally sequential block of instructions with a single entry point and a single exit point) as nodes. These nodes are connected to each other by the control-flow break instructions. Since it is very easy to compute the WCET of basic blocks (it is the sum of the execution time of the instructions it is made of), all the problematic of these techniques resides in finding the "longest path" in the control-flow graph. Figure 3 shows how a FAÇADE program is changed into a control-flow graph.

*Path-Based techniques* make usage of the classical Graph algorithms (*e.g.* Dijkstra [8]) to find the longest path of the graph. This path is then checked for feasibility - if not feasible, the path is removed from the graph and the algorithm is run again until a longest feasible path is found. Although better suited for Camille than Tree-Based analysis, this method still requires some heavy computation to be done, with remembering of incorrect paths, and so on. Therefore, it can't reasonably be run on-card.

*The Implicit Path Enumeration Technique* (*IPET* [18]) also uses the control-flow graph to generate a set of constraints that must be respected. Then another set of constraints is created from the loop bounding information (either provided by the programmer, either deduced from the program or the language). These constraints allow the elimination of infeasible paths. Using these two sets of constraints, a constraint solving or integer linear programming algorithm is used to maximize the WCET expression:

$$WCET = \sum_i n_i \times w_i$$

Where $n_i$ is the number of times Basic Block *i* is run, and $w_i$ is the cost of Basic Block *i*.

Of course, the computation needed by this method is still enormous compared to what the card provides us. However, its properties let think than a two-phases computation can be done, the heavy part outside the card, and the light part on-card. For this reason, this method looks the most promising to address the issues that are listed below.

#### 3.1.3. Static Analysis, Camille, and FAÇADE

Implementing real-time in Camille raises a lot of questions, about the best way to re-think FAÇADE to have the less pessimist WCET computations possible and to face the issues related to Camille's two phases program loading.

##### 3.1.3.1. Adapting FAÇADE to face RT issues

FAÇADE is a dramatically simple intermediate language that only has 5 instructions in its current form. Since making real-time programs involves complying to some constraints (bounded loops, etc.), the idea is to extend FAÇADE to make it have some real-time safe instructions that can ensure the WCET computation is possible when they are used.

For the example to face the problem of bounded loops, FAÇADE is provided an `UNTIL` instruction that delimits a bounded loop block and takes the maximum number of iteration, or an expression of it as a parameter. This instruction is used whenever possible in place of the classical jumping instructions to form a loop.

Since Camille aims at running both real-time and non real-time tasks, this solution looks both reasonable and efficient. Programs that can't be compiled using the RT-safe instruction or that doesn't respect the RT constraints like the absence of recursion will simply not be suitable for RT, and programs that comply will then take advantage

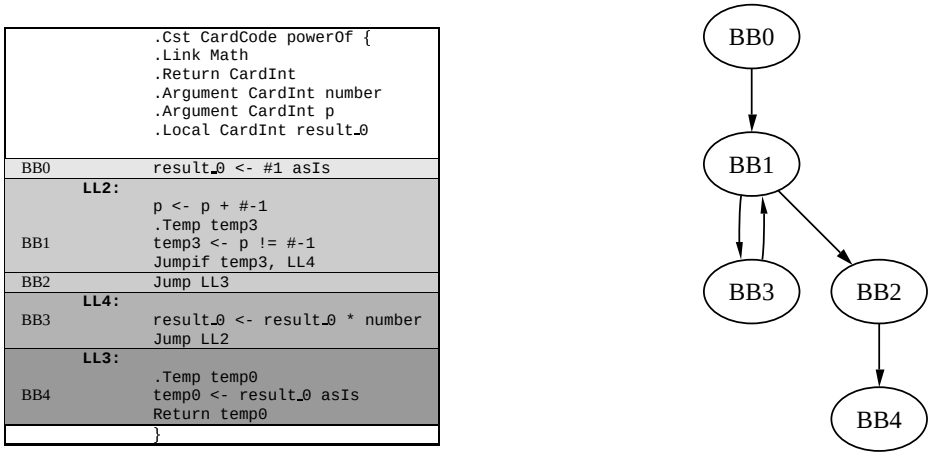


**Figure 3. A FAÇADE program and its associated control-flow graph**

of the higher-level structural information for their WCET computation (less pessimism for nested loops, etc.).

In practice, the Gnu Compiler Collection [3] does support the generation of such looping instructions. This would make it possible even for compiled programs to take advantage of the bounded-loop instruction.

#### 3.1.3.2. WCET instantiation and verification

The main principle of Camille is that the binary program to be loaded into the card contains the proof of its correctness, using a Proof Carrying Code technique [21]. Thanks to this, the heavy type-inference operation is made outside by the assembler program and the card only needs to check the validity of the proof, which is perfectly feasible while the card loads the program. We have formally proved that this solution gives the same level of security as if the card did the type-inference itself [23].

Implementation of real-time in Camille respect the same philosophy, that is, the binary program contain the "proof" that it will run within a given time at worst case. Of course, the card is able to verify this proof while loading the program, and the proof system must be as safe as if the card did all the external checking itself. The proof is only here to "show the way" to the verifier, so it doesn't need to perform heavy computations.

There are two problems that arises from this idea:

1. The maximization algorithm that will be chosen to maximize $WCET = \sum_i n_i \times w_i$ will have to be verifiable, *i.e.* the result it gives must be formally and quickly verifiable with some additional datum that the ones produced during off-card computation,

2. The WCET will have to be *instantiated* on-card, because the cost of the basic blocks is not known by the external world. The WCET of a FAÇADE instruction directly depends on the hardware architecture the program will run on (it will not be the same on an AVR processor and on a StrongArm) and also on compilation patterns and optimizations used by the on board compiler. This means that in the WCET computation formula, we will not be

[3] GNU Compiler Collection Internals, `http://gcc.gnu.org`.

able to know the $w_i$s. How can we thus maximize the formula?. Knowing the $n_i$s would be sufficient for the card to easily instantiate the WCET, since it can then quickly calculate the $w_i s$ while the program is loaded. However, the $n_i$s are also architecture-dependent, so they can't be calculated off-card neither. Indeed, in an *if* statement in which Basic Blocks A and B are the alternatives, A can cost more than B on one architecture while the opposite can also be true on different hardware. In this case, the $n_i$s will be different for the two architectures and therefore can't be computed off-card.

A solution to address the last issue is to require the terminal that loads the program onto the card to compute the RT proof at load time. The terminal send the program to the card, which send it the $w_i$s as a response. The terminal then generate the RT proof and send it to the card for verification. This solution is unfortunately not suitable because asking such processing power from the terminal increase the complexity of software deployment in the embedded device. Moreover, as said above, industrials are not willing to let any part of their specifications going outside the card: knowing the execution time of the instructions could open the way to timing attacks.

The only solution that seems realistic consist in computing neither the $n_i$s nor the $w_i$s off-card, but instead give the card enough additional informations so it can safely perform "the trick" itself. In other words, doing the most constraining part of the work off-card and letting the card finish it with its knowledge of the $w_i$s. In these conditions, the card does not verify a proof, but instead finish a work that was started outside. The Simplex method offers a way to know whether a result is optimal of not. Thus, the "proof" consist of a set of datum that look promising in most extreme cases, so the card can choose the one that is the most well suited and finish the computation on this good basis. Doing so, it solves the first issue in the same row, since there is no verification to be done. However, the safety of this method has yet to be demonstrated (can the outside world fool the card with wrong datum?) and other ways have still to be formalized as well.

### 3.2. Cooperative real time schedulers

Real time system needs a knowledge of the WCET of the treatments they support (by experimental measurement or analytical computation). Nevertheless, supporting real time in an exo-kernel require to focus on the three previously listed key points: (*i*) quantify the execution time for each of the exo-kernel primitives, (*ii*) find a schedule that succeeds all running applications deadlines, (*iii*) define a way to allow applications to notify their real time requirements.

#### 3.2.1. RT and exo-kernel primitives

To extend Camille into a real time operating system, we need to quantify the execution time for every primitives of the exo-kernel. Exo-kernels only offers simple primitives to access efficiently and safely the hardware. Thus, due to their simplicity, exo-kernel's primitives are easy to bound in time. Similar analysis have been performed on other real time kernel code like RTEMS [4]. Another "tricky" way is to compile the code of the Camille kernel using our GCC and to analyze the resulting FAÇADE code using our WCET algorithms and tools.

The major technological challenge on a smart card is linked to persistent memory (EEPROM) which presents annoying electronic properties. Writing one page is slow (it takes about 5ms) and deny any access to the current physical page within the whole writing delay. EEPROM writing operations implies hardware locks. This challenge may create difficulties to predict primitives duration to compute their WCET. Recent results

introduced by [13] promotes solutions for solving this kind of locks in the smart card context.

### 3.2.2. Scheduler in exo-kernel

Provided with the knowledge of the execution time required for each of the exo-kernel primitives and also by each of the downloaded applications, the problem is now to find a schedule for a set of standard and real-time tasks. Exo-kernels only support "equity" in term of access to the micro-processor using a "round-robin" policy (simple and impartial). The time slices are granted to operating system extension which elect a runnable application according to their own scheduling policy. This two levels CPU sharing process is close to hierarchical schedulers [22] or to the Zeng and Liu approach described in [6]. It is this sufficient to build a RT scheduler over an exo-kernel, but it introduces sub-optimal solutions.

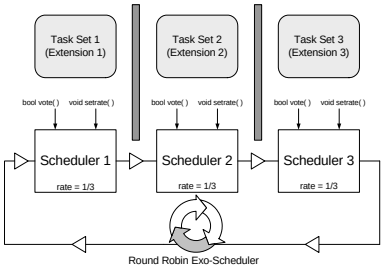


**Figure 4. Naive hierarchical scheduler implementation for exo-kernel**

For example, Figure 4 shows a naive implementation that enables exo-kernel extensions to bring their own scheduler to schedule their tasks. Each scheduler has the guarantee to have access to the CPU with a rate of $\frac{1}{N}$, where $N$ stands for the number of schedulers on the card. Once we load a new one, a vote is organized to determine if every present scheduler accepts and still meets its deadlines with the new rate of $\frac{1}{N+1}$. If the vote succeed, the new scheduler is accepted with its set of tasks, otherwise, the user is notified of the vote failure. This model is not optimal for real time tasks as extensions try to schedule their set of tasks without collaborating with each other. This could lead to reject extensions whose tasks could be scheduled if RT extensions collaborate. For example, lets consider two RT extensions that have the following task sets: $\{$A (rate $\frac{1}{2}$), B (rate $\frac{1}{5}$)$\}$ for first and $\{$C (rate $\frac{1}{4}$)$\}$ for second. The round-robin will schedule extensions one after another. Scheduler 1 is called by the "round-robin like" exo-scheduler and activates the task A. Scheduler 2 is now granted the time slice and elects task C. Scheduler 1 now elects task A and has no more remaining slot for task B which miss its deadline. We introduce our collaborative scheduling policies in the next section that solves such problems.

### 3.2.3. Our approach

Lets us reconsider the example of the task set $\{$A,B,C$\}$. It can be scheduled using a simple rate monotonic [19] policy (the schedule plan is ACABA CABAC ABACA BACA-Free with a period of 20). The previously described solution has failed because the Camille extension supporting task A and B is isolated from the one supporting task C. Our solution consists in allowing grouping RT extensions that accept to share their cpu slice into a virtual collaborative exo-scheduler which provides guarantees concerning the security level (*i.e.*, the deadlines will be met even if extensions do not trust each other). Architecture and scheduler's interfaces are presented Figure 5. One scheduler of the collaborative real-time schedulers (CRT) is granted the right to schedule the whole task sets and is named the active scheduler. When the collaborative exo-scheduler (CES) is given a time slice (*i.e.*, when the round robin invoke the *void pre( )* method), CES calls the active CRT's *int*

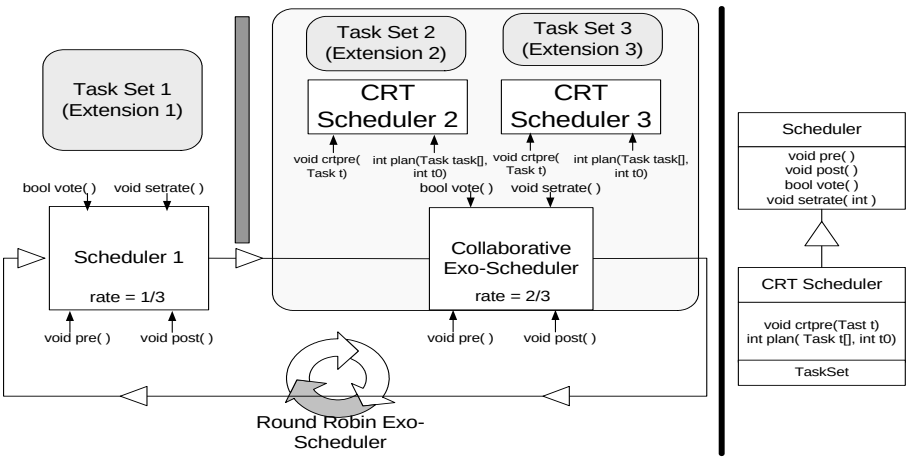


**Figure 5. CamilleRT Collaborative Scheduler and Scheduler Interface**

*plan( Task task[], int t0)* interface which return the *id* of the task to be executed for time slice *t0*. CES now invokes *void crtpre(Task t)* on the scheduler of the extension corresponding to the task *id*. At the end of the time slice the round-robin invokes the method *void post( )* to perform the context switching of the last active task (CES dispatches this event to the extension corresponding to this task). The main problem is to guarantee to extensions that their deadline will be met by the active scheduler which is brought by an extension which other extensions do not trust. This guarantee is achieved thanks to a checking algorithm performed during the loading and installation phase.

At loading time of a new collaborative real time extension, a new vote is organized. In case of success, the CES check the possibility to install the extension with its scheduler and tasks. The first problem is to find a new scheduler from the pool of collaborative schedulers which will meet the deadlines for all tasks. The second problem, which is the most important, is to check that this scheduler will effectively succeed. The interface method *int plan(Task task[], int t0)* is used to find a working scheduler. Each potential scheduler is submitted a tasks list containing all the current tasks and the ones coming with the new extension. If the scheduler is not able to schedule this task set it simply return -1 and the CES asks another scheduler. Once the CES has found a scheduler, the CES verifies that it will effectively succeed by asking for the scheduling plan on the hyperperiod. The verification is of complexity $O(hyperperiod)$ and only consists in checking that every task meets its deadline. Problem is to be sure that the *plan* function is an *untrusted deterministic function* (*i.e.* no internal state, no access to the hardware clock or any random like functions) which can help detecting when CES asks for the installation of a new extension or for the election of a task and can lead to break the guarantee that each RT task will succeed its deadline. Such a property can be achieved by using Domain Specific Language (DSL) like for example Bossa [17] for the development of real time schedulers. A similar problem was solved at native code level in exokernel for the *own()* function which secure the access to meta-data for disks management (Chapter 4 in [11]). These techniques can obviously be used on the FAÇADE dedicated intermediate language.

## Conclusion

The paper has presented introduction of real time extension in the Camille exo-kernel for smart card. It identify four key points to import real time extension in exo-kernel architecture. Conventional exo-kernels support this kind of extensions. But the paper shows one example of un-efficiency for real time schedulers in standard exo-kernel approach. The approach presented in this paper simply solve this problem and detail strategies used to improve the Camille operating system.

This paper also introduces two open problems. The first one is linked to schedule plan: predicting impact of hardware locks, impact of interruption multiplexing and its dispatching from hardware to extensions. The second one is linked to the WCET computation, the safety of the algorithms described in this article need to be proved.

## References


[1] Z. Chen. *Java Card Technology for Smart Cards*. Addison Wesley, 2000.

[2] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2):249–274, april 2000.

[3] A. Colin and I. Puaut. A modular and retargetable framework for tree-based wcet analysis. In *13th Euromicro Conference on Real-Time Systems*, Delft, The Netherlands, June 2001.

[4] A. Colin and I. Puaut. Worst-case execution time analysis of the RTEMS real-time operating system. In *13th Euromicro Conference on Real-Time Systems*, pages 191–198, Delft, The Netherlands, june 2001.

[5] K. Crary and S. Weirich. Resource Bound Certifi cation. In *the 27th ACM Symposium on Principles of Programming Languages*, 2000.

[6] Z. Deng and J. Liu. Scheduling real-time applications in an open environment. In *Procedings of the 18th IEEE Real-Time Systems Symposium (RTSS'97)*, pages 308–319, December 1997.

[7] D. Deville, A. Galland, G. Grimaud, and S. Jean. Smart Card operating systems: Past, Present and Future. In *the 5th NORDU/USENIX Conference*, Västerås, Sweden, February 2003.

[8] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, (1):269–271, 1959.

[9] J. Engblom, P. Altenbernd, and A. Ermedahl. Facilitating worst-case execution time analysis for optimized code, 1998.

[10] J. Engblom, A. Ermedahl, and F. Stappert. Comparing different worst-case execution time analysis methods.

[11] D. R. Engler. *The Exokernel operating system architecture*. PhD thesis, Massachusetts Institute of Technology (MIT), 1999.

[12] D. R. Engler and M. F. Kaashoek. Exterminate All Operating System Abstractions. In *the 5th IEEE Workshop on Hot Topics in Operating Systems*, Orcas Island, USA, 1995.

[13] A. Galland and M. Baudet. Controlling and optimizing the usage of one resource. To appear.

[14] G. Grimaud, J.-L. Lanet, and J.-J. Vandewalle. FAÇ ADE: A Typed Intermediate Language Dedicated to Smart Cards. In *Software Engineering–ESEC/FSE*, 1999.

[15] I. S. O. ISO. Integrated circuit(s) cards with contacts, parts 1 to 9, 1987-1998.

[16] O. Kömmerling and M. G. Kuhn. Design principles for tamper-resistant smartcard processors. In *USENIX Workshop on Smartcard Technology*, pages 9–20, 1999.

[17] J. L. Lawall, G. Muller, and L. P. Barreto. Capturing OS expertise in an event type system: the Bossa experience. In *Tenth ACM SIGOPS European Workshop 2002 (EW2002)*, pages 54–61, St. Emilion, France, Sept. 2002.

[18] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Workshop on Languages, Compilers, & Tools for Real-Time Systems*, pages 88–98, 1995.

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

[20] T. S. Messerges, E. A. Dabbish, and R. H. Sloan. Investigations of power analysis attacks on smartcards. In *USENIX Workshop*

*on Smartcard Technology*, pages 151–162, 1999.

[21] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, Jan. 1997.

[22] J. Regehr. *Using Hierarchical Scheduling to Support Soft Real-Time Applications on General-Purpose Operating Systems*. PhD thesis, University of Virginia, 2001.

[23] A. Requet, L. Casset, and G. Grimaud. Application of the B formal method to the proof of a type verifi cation algorithm. *HASE*, 2000.

[24] S. Thibault, J. Marant, and G. Muller. Adapting Distributed Applications Using Extensible Networks. In *the 19$^{th}$ IEEE International Conference on Distributed Computing Systems*, 1999.