Extensions Temps-Réel pour Exo-Noyau Embarqué

Damien Deville, Alexandre Courbot, Gilles Grimaud

RENPAR'15 / CFSE'3 / SympAAA'2003
La Colle sur Loup, France, 15 au 17 octobre 2003

Extensions Temps-Réel pour Exo-Noyau Embarqué

Damien Deville, Alexandre Courbot, Gilles Grimaud

Université des Sciences et Technologies de Lille, LIFL, Bât. M3,
Cité Scientifique, 59655 Villeneuve d'Ascq - France
{deville,courbot,grimaud}@lifl.fr

Résumé
Les cartes à puces sont des petits objets portables axés principalement sur la sécurité (10^9 unités vendues principalement en Asie et en Europe). Afin de permettre aux logiciels encartés de supporter plus de services, les systèmes d'exploitation pour cartes ont évolué d'une plateforme d'exécution monolithique dédiée vers des architectures systèmes plus ouvertes qui supportent le chargement dynamique de code. Cet article présente les problèmes temps-réel du système d'exploitation pour carte Camille, qui présente les caractéristiques suivantes : chargement dynamique de code, vérification de type embarquée, compilation à la volée, et chargement dynamique de composants systèmes. D'une manière plus générale, il traite des difficultés à accorder les extensions temps-réel avec les principes des exo-noyaux.

Mots-clés : Systèmes embarqués, temps réels, code mobile, systèmes extensibles .

Introduction

Une carte à puce est un objet fortement contraint : processeur de faible puissance, capacités d'entrées/sorties limitées, mémoire de petite taille (généralement de 1 à 4 Ko de RAM, de 32 à 128 Ko de ROM et de 16 à 64 Ko de Flash RAM). Mais sa facilité d'utilisation et sa résistance physique aux attaques en font l'un des objets mobiles de choix. Cet article présente les travaux menés sur l'exo-noyau Camille RT. Afin de situer le contexte dans lequel notre travail a pris place, la section 1 résume les capacités matérielles des cartes à puces. La section 2 présente les motivations qui ont abouti au développement de Camille et de Camille RT. Enfin, les sections suivantes discutent des solutions utilisées pour le support du temps-réel dans l'exo-noyau Camille.
Les cartes à puce sont des petits objets portables et sécurisés (POPS) qui supportent aujourd'hui la *post-issuance* (chargement dynamique de code après émission de la carte). Leurs contraintes matérielles font l'objet de plusieurs définitions ISO [15] - leur but étant principalement de garantir à la carte un certain degré de résistance aux attaques physiques [16].

1. Contexte des cartes à puce

1.1. Microprocesseurs

Les classes de microprocesseurs encartés sont très diverses (tableau 1), allant du vieux CISC 8 bits (4,44 Mhz) au puissant RISC 32 bits (100 à 200 Mhz). Le type de processeur utilisé sur carte dépend en grande partie des normes ISO [15]. Les nouvelles applications embarquées demandent de plus en plus de puissance, ce qui amène les concepteurs de cartes à choisir des processeurs 32 bits (ou des variantes améliorées de processeurs CISC 8 ou 16 bits). Néanmoins, ces processeurs restent d'un fonctionnement simpliste (pas de cache, pas de pipeline, ...) ce qui simplifie les predictions de temps d'exécution.

Modèle	Architecture	Taille du bus de données	Taille et nb des registres	Fréquence
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 à 28.16 Mhz
R4KSC	RISC	32 bits	32 (32 bits)	4,77 Mhz à 100 Mhz

TAB. 1 – *Caractéristiques des processeurs les plus utilisés sur carte.*

1.2. Mémoires

Il existe différents types de mémoires sur une carte (tableau 2). La première est la RAM (Random Access Memory). On y trouve aussi de la ROM (Read Only Memory), ainsi que de l'EEPROM (Electric Erasable Programmable Read Only Memory) ou de la FlashRam, qui sont des mémoires persistantes réinscriptibles. Étant donné que le silicium de la carte doit être limité à 27 mm^2, le point mémoire (surface de silicium requise pour stocker un bit de mémoire) est un facteur important. La carte à puce classique dispose de 2 à 4 Ko de mémoire de travail, 32 à 128 Ko de mémoire persistante, et 64 à 128 Ko de ROM. La mémoire persistante a un inconvénient majeur, lié à ses propriétés électroniques. Son délai

Type	Point mémoire	Capacité	Temps d'écriture	Taille de page
ROM	référence	32 à 128 Ko	lecture seule	1 octet
FlashRAM	x 2-3	16 à 64 Ko	2,5 ms	64 octets
EEPROM	x 4	4 à 64 Ko	4 ms	2 à 64 octets
RAM	x 20	128 à 4096 octets	$\leq 0.2\mu s$	1 octet

TAB. 2 – *Caractéristiques des mémoires cartes.*

d'écriture est jusqu'à 10000 fois plus important que celui de la RAM. De plus, l'écriture en mémoire persistante peut endommager ses cellules (ce phénomène est appelé *stress*).

1.3. Entrées/Sorties

Les producteurs de cartes ont fourni des spécifications ISO qui définissent les protocoles d'entrées/sorties. La normalisation filaire est l'ISO 7816 et se décline en protocoles << T=x >> . La normalisation sans fil (pour les cartes sans contact physique) est définie dans l'ISO SC17 14443. Certains prototypes de carte à puce ont utilisé des protocoles et des liens physiques plus conventionnels. << T=0 >> et << T=1 >> sont les protocoles les plus utilisés dans l'industrie. Ils fournissent un taux de transfert allant de 9600 à 192000 bauds via une ligne série semi-duplex (ces taux garantissant le transfert d'1 Ko de données en moins d'une seconde). L'implémentation des protocoles d'entrées/sorties introduit des contraintes temps-réel. En fait, les ISO 7816-3 & 4 définissent un délai maximal avant transmission de l'*Answer To Reset* (ATR, *i.e.* le flux d'octets produit par la carte lors de la mise sous tension). D'autres échéances existent pour les réponses aux requêtes du terminal. Des contraintes RT pour les systèmes d'exploitation pour cartes sont donc explicitées dans les nomes ISO.

2. Des systèmes d'exploitation pour cartes ouverts à Camille

Depuis la naissance des cartes à puce au début des années 80, l'architecture des logiciels pour carte n'a pas cessé d'évoluer. Durant cette période, un grand nombre de standards ont été proposés. Ils sont

regroupés dans les normes ISO7816-x, et réglementent toutes les fonctionnalités des cartes, des caractéristiques physiques jusqu'à la gestion des applications.
Cinq acteurs interviennent dans le cycle de vie d'une carte à puce. Les *fabricants de silicium* conçoivent et produisent les puces en masse. Les *producteurs de carte* (associés dans cet article aux producteurs de logiciels) embarquent sur carte ce que demandent les distributeurs. Les *distributeurs de cartes* ont des considérations plus commerciales et distribuent des solutions logicielles basées sur les cartes. *Les fournisseurs de services* conçoivent et déploient (sous le contrôle des distributeurs) des services à valeur ajoutée et les *utilisateurs* bénéficient de ces services.
Le cycle vie d'un logiciel carte compte trois étapes. Tout d'abord, le logiciel est *produit* et *chargé* (*i.e.* embarqué dans la puce). Ensuite, en fonction de sa nature (application *prête-à-exécuter* ou *prête-à-charger*), le logiciel est *initialisé* ou *instancié*. Enfin, le logiciel embarqué est *utilisé* dans des applications client/serveur.
Les systèmes d'exploitations pour carte ont évolué d'une architecture de première génération (un monolithe fourni par les producteurs de cartes pour les producteurs de puces qui correspond aux requêtes des utilisateurs et aux spécifications de la puce) vers des systèmes ouverts permettant le chargement dynamique de code après émission de la carte à l'utilisateur final (*post-issuance*). Ceci permet aux utilisateurs de personnaliser leurs applications afin de les adapter au mieux à leurs besoins. Les plates-formes *prêtes-à-charger* sont habituellement basées sur une machine virtuelle, à la fois pour des raisons de portabilité (une application peut être chargée sans modifications dans différentes cartes utilisant du matériel hétérogène) et de sécurité (il est plus facile de prouver ou de s'assurer de l'innocuité de code sous forme intermédiaire). La figure 1 décrit l'architecture de deux de ces OS : Multos (une machine virtuelle qui supporte les bytecodes Mel) et Java Card (une configuration Java dédiée conçue pour correspondre au mieux aux contraintes des cartes) avec support SIM (Subscriber Identity Module). Multos est principalement utilisé en milieu bancaire (Master Card). Les cartes Java SIM sont utilisées dans les téléphones portables.

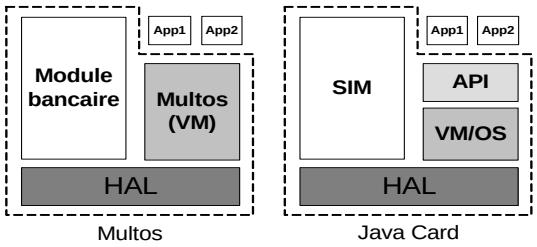

FIG. 1 – *Extensibilité dans les systèmes d'exploitation ouverts pour carte.*

Multos et Java Card supportent tous deux le chargement dynamique de code au niveau applicatif, afin de supporter des services multi-applications. Ces deux OS sont dédiés à une application particulière, très proche du système. Multos comprend un noyau de gestion d'applications bancaires utilisé par Master Card, tandis que Java SIM Card supporte une application nommée SIM qui sert de pont entre la carte et la partie du téléphone dédiée à la communication. Le traitement le plus important pour une carte SIM est de produire et fournir une clé cryptographique de façon périodique qui sera utilisée pour l'authentification, à défaut de quoi la BSC (*Base Station Controler*, partie de l'infrastructure d'un réseau GSM) coupera la communication. Ces deux applications sont si proches de l'OS qu'un programmeur ne peut les développer comme des applications standard en utilisant un langage comme Mel ou Java. Cela signifie que le code de ces applications ne peut être mis à jour, l'extensibilité n'étant possible qu'au niveau

des applications utilisateur. Sur la figure 1, la ligne pointillée délimite ce qui est extensible de ce qui est chargé une fois pour toute par le fabricant. L'inconvénient principal de ces deux OS est qu'ils ne supportent que les applications écrites dans leur langage au-dessus des abstractions qu'ils fournissent. Camille est le premier prototype d'un système ouvert pour carte à puce supportant le chargement de nouvelles abstractions matérielles efficaces. L'ouverture du système d'exploitation Camille se matérialise par ses APIs qui représentent le matériel tel qu'il est, sans imposer une quelconque abstraction. Le programmeur peut ainsi définir celles dont il a besoin alors que Multos ou Java Card imposent leur propre modèle d'abstraction (mémoire objet ou système de fichiers ISO vs. pages de mémoire physique, ...).

2.1. Camille

Le but de Camille est de supporter les différentes ressources matérielles utilisées dans les cartes. Les pages mémoires, valeurs numériques, le microprocesseur et le code natif peuvent facilement être manipulés par les applications tout en garantissant un haut niveau de sécurité. Camille est conçu à la manière des exo-noyaux [10, 11] du MIT, avec des principes et des concepts similaires (*i.e* pas d'abstraction matérielle mais un accès sécurisé et (dé)multiplexé à celui-ci). Le code embarqué est exprimé via un langage intermédiaire dédié nommé FAÇADE [14], mais les programmes peuvent être écrits dans un langage de haut niveau. Le code C peut ainsi être compilé vers du FAÇADE en utilisant GCC. Camille fournit trois caractéristiques basiques. La *portabilité* découle de l'usage du code intermédiaire et d'un ensemble limité de primitives matérielles. La *sécurité* est assurée par la vérification du code (en utilisant un algorithme similaire à PCC [20]) au moment où celui-ci est chargé. L'*extensibilité* provient de la représentation très simple du matériel qui ne définit aucune abstraction à la racine du système. Par conséquent, les applications doivent importer ou construire elles-mêmes les abstractions dont elles ont besoin. L'architecture en deux parties de Camille est décrite figure 2.

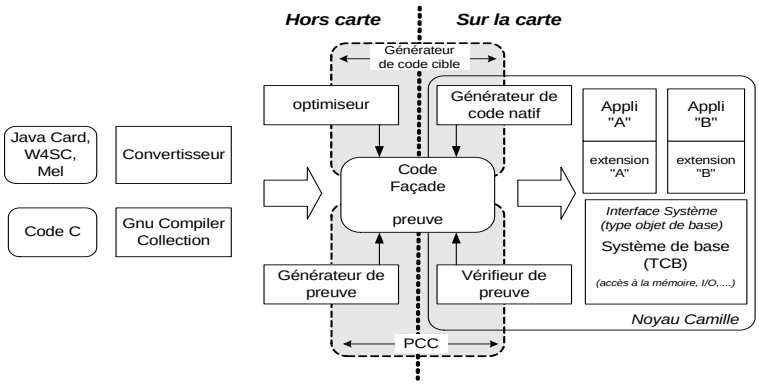

FIG. 2 – *Architecture et infrastructure logicielle de Camille.*

L'extensibilité se fait en général au détriment de la performance. Les parties du système qui demandent une efficacité maximale peuvent bénéficier d'un compilateur *à la volée* pour compiler le code intermédiaire en code natif. Une amélioration des performances découle également de l'approche des exo-noyaux qui n'entraînent aucune pénalité due à une abstraction dans le noyau de l'OS. Des optimisations indépendantes du matériel peuvent également être faites hors de la carte, au moment où le code source est transformé en code FAÇADE . Une description plus précise de Camille ainsi que des résultats expérimentaux peuvent être trouvés dans [6, 13]. Le prototype Camille démontre la faisabilité d'un OS extensible pour carte dont l'empreinte est raisonnable : 17 Ko de code natif, parmi lesquels 3,5 Ko pour la vérification de code, 8,5

Ko pour la génération de code natif et le reste pour le multiplexage du matériel.

2.2. Nouveau Problème

Le problème principal de Multos et de Java SIM vient du type particulier d'applications qu'ils supportent. SIM et le noyau bancaire sont des applications, mais sont en même temps tellement proches de l'OS et de la machine virtuelle qu'un développeur ne pourrait pas les écrire en utilisant les outils de développement classiques. SIM possède ainsi son propre code de commutation de contexte afin de s'assurer du respect de l'échéance temporelle pour la génération de la clé cryptographique. Une telle fonctionnalité serait difficile à implémenter en utilisant le langage Java Card, l'API ou les abstractions, et interfère tellement avec la machine virtuelle qu'elle peut être considérée comme une partie du système d'exploitation.
Camille apporte une solution à de tels problèmes. Le support d'applications Java Card est possible et a déjà été réalisé ; il est également possible de concevoir et d'implémenter une machine virtuelle Java Card au dessus du compilateur *à la volée* de Camille en utilisant le niveau d'expressivité minimal de l'exo-noyau. La même chose peut être faite pour le support des applications Multos dans Camille.

Le reste de cet article discute de nos travaux sur les problèmes temps-réel dans le contexte de l'exo-noyau Camille et se concentre sur deux points : le calcul du temps d'exécution au pire cas (*Worst Case Execution Time, WCET*) et la conception d'une architecture d'ordonnanceurs temps-réel pour exo-noyaux.

3. Camille RT

L'architecture standard des exo-noyaux, définie par le MIT, n'offre de primitives dédiées au temps réel ni pour les applications, ni pour les extensions. L'exo-noyau est conçu pour permettre l'extensibilité tout en garantissant une sécurité maximum. Il ne fait qu'offrir un accès équitable au processeur pour chaque << système >> [10]. Pour chaque quantum de temps CPU, le noyau utilise une politique de << *round-robin* >> pour élire une application (une primitive *yield* est proposée aux extensions qui veulent offrir leur temps restant aux autres). La motivation principale de Camille RT est de pouvoir offrir aux extensions système et aux applications utilisateur la possibilité d'implémenter des primitives temps réel et de faire cohabiter les applications standard avec les applications temps-réel.
Les points-clés pour permettre le support du temps-réel dans un système d'exploitation extensible basé sur une architecture de type exo-noyau sont : (*i*) quantifier le temps d'exécution de chaque primitive de l'exo-noyau (*par ex.*, le délai requis pour un accès à une TLB virtuelle) ; (*ii*) trouver un ordonnancement qui satisfait toutes les limites de temps des applications en cours d'exécution ; (*iii*) définir un moyen permettant aux applications d'informer l'exo-noyau de leurs exigences temps-réel (limite de temps, taux d'utilisation, temps de départ, ...) ; (*iv*) quantifier le temps d'exécution des programmes RTs exprimés dans le langage intermédiaire FAÇADE .
Les trois premiers points concernent la problématique temps-réel dans un exo-noyau quelconque, la dernière est spécifique à Camille qui utilise un langage intermédiaire pour améliorer la portabilité. Nous aborderons tout d'abord le problème *iv*, les problèmes *i*, *ii* et *iii* seront traités ensuite.

3.1. Calcul de WCET sur la carte

Dans notre contexte (systèmes temps-réels durs, les échéances critiques et le non-respect de l'une d'elles peut compromettre l'intégrité du système), il est impératif de connaître le WCET (*Worst Case Execution Time*, temps d'exécution au pire cas) de chaque programme RT à exécuter, ou au moins d'un temps qui est connu pour être supérieur à ce WCET. Pour obtenir un tel temps, la méthode usuelle est de faire une *analyse statique* du code de la tâche, c'est à dire analyser le code afin de faire une estimation pessimiste de son WCET. En pratique, le WCET obtenu par analyse statique sera toujours supérieur au << vrai >> WCET, ce qui signifie que nous pouvons être sûrs que les échéances seront satisfaites si

l'ordonnanceur affecte à la tâche un délai supérieur à ce temps pour se terminer. L'inconvénient de l'analyse statique est qu'elle est souvent très pessimiste, ce qui entraîne un gaspillage des ressources égal à la différence entre le WCET calculé et le WCET réel. Beaucoup de méthodes ont été développées afin de réduire cette différence le plus possible, notamment en simulant précisément l'architecture sur laquelle la tâche va être lancée [2, 3, 1].
D'autres méthodes de calcul de WCET sont également connues sous le nom d'*analyse dynamique*, cependant elles ne sont pas utilisables dans le cadre de nos travaux : étant basées sur des exécutions de la tâche avec des données d'entrées sensées la faire durer le plus longtemps possibles, elles ne peuvent garantir complètement que le temps donné sera supérieur au WCET réel de la tâche. Pour cette raison, les sections suivantes se concentrent exclusivement sur les méthodes d'analyse statique.

3.1.1. RT et code chargé

La gestion des contraintes temps-réel demande la connaissance du temps d'exécution de chaque programme RT. Camille étant un système d'exploitation ouvert, les applications sont compilées vers ou écrites dans un langage intermédiaire nommé FAÇADE . Dans sa forme actuelle, Camille est extensible au niveau système et applicatif, mais ne supporte pas le calcul de WCET. Les algorithmes pour le calcul du WCET [1] et des langages permettant un calcul du WCET aisé sont bien connus [5, 23] mais contraignent fortement le développement. Il faut donc proposer un sous-modèle de FAÇADE plus restrictif et utilisé pour le développement d'applications et d'extensions temps-réel, et utiliser le modèle existant pour les applications classiques. Ainsi les applications classiques et temps-réel peuvent cohabiter. Le calcul de WCET doit être effectué sur la carte pour différentes raisons. La principale est que FAÇADE est compilé durant son chargement par la carte, et qu'elle seule peut ainsi connaître le temps d'exécution de chaque instruction, étant donné que celui-ci dépend de la plate-forme physique. Le calcul de WCET hors-carte serait possible en exportant un << profil de la puce >> contenant le code natif généré pour chaque instruction par le compilateur << à la volée >> ainsi que le temps d'exécution de chaque instruction native. Mais les fabricants de cartes à puce sont hostiles à l'idée de rendre ces informations publiques pour des raisons économiques et industrielles, comme garder secrète la technologie employée sur la carte ou réduire les risques d'attaque temporelles (DSA [19]) pour les opérations cryptographiques.

3.1.2. État de l'art des techniques d'analyse statique

Nous allons détailler ici les trois techniques principales d'analyse statique : l'analyse *Tree-Based*, l'analyse *Path-Based*, et la technique d'énumération implicite des chemins (*Implicit Path Enumeration Technique*).
L'analyse *Tree-Based* [3] utilise l'arbre syntaxique du programme (construit par le compilateur à partir du code source) afin d'obtenir une vue détaillée de sa structure. Cette méthode donne de bons résultats et permet de mettre en oeuvre des optimisations visant à réduire le pessimisme du WCET calculé grâce à une bonne connaissance de la structure du programme. Cependant, une partie de l'analyse doit être faite au niveau du code source (C ou tout autre langage de haut niveau). De plus, une correspondance doit être établie entre l'arbre syntaxique et les blocs de base du code assembleur généré par le compilateur, ce qui n'est pas toujours évident car la structure du code peut avoir changé durant la compilation (le plus souvent durant les phases d'optimisations). Des solutions existent pour répondre à ces problèmes [8]. L'utilisation de cette analyse sur FAÇADE ne fonctionnerait que sur un sous-ensemble du langage, et nécessiterait la transmission de l'arbre syntaxique du programme à la carte, opération qui n'est pas rentable en raison des faibles capacités mémoire et processeur de cette dernière, et du faible débit de sa ligne série. En dehors de ces contraintes, les techniques *Tree-Based* présentent des caractéristiques intéressantes pour le calcul sur carte : complexité et pessimisme faibles, calcul de WCET à la volée, ce qui permet de réduire l'utilisation mémoire.
Les deux autres techniques d'analyse statique [9] utilisent une représentation du flot de contrôle du

```
                .Cst CardCode powerOf {
                .Link Math
                .Return CardInt
                .Argument CardInt number
                .Argument CardInt p
                .Local CardInt result_0
BB0             result_0 <- #1 asIs
        LL2:
                p <- p + #-1
                .Temp temp3
BB1             temp3 <- p != #-1
                Jumpif temp3, LL4
BB2             Jump LL3
        LL4:
BB3             result_0 <- result_0 * number
                Jump LL2
        LL3:
                .Temp temp0
BB4             temp0 <- result_0 asIs
                Return temp0
                }
```

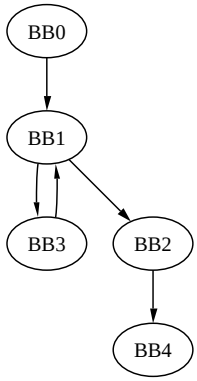

FIG. 3 – *Un programme* FAÇADE *et le graphe de flot de contrôle correspondant.*

programme. Un *graphe de flot de contrôle* est composé de *blocs de base* (blocs d'instructions totalement séquentielles et n'ayant qu'un seul point d'entrée et un seul point de sortie) qui servent de sommets. Ces sommets sont connectés entre eux par leur séquence et les instructions de rupture de flot de contrôle. Le calcul de WCET est extrèmement simple pour les blocs de base (il s'agit de la somme des temps d'exécution des instructions qui les composent), ainsi toute la problématique de cette technique est de trouver un << chemin >> du sommet d'entrée vers le sommet de sortie du graphe. La figure 3 montre comment un programme FAÇADE peut être changé en graphe de flot de contrôle. Les techniques *Path-Based* appliquent les algorithmes de graphes classiques (*ex.* Dijkstra [7]) afin de trouver le plus long chemin du graphe. La faisabilité de ce chemin est alors testée - s'il n'est pas valide pour le programme, le chemin est supprimé du graphe et l'algorithme est relancé jusqu'à ce que le plus long chemin faisable soit trouvé. Bien que cette technique soit plus abordable pour Camille que la méthode *Tree-Based* en termes de surcoût de transmission de données pour le calcul de WCET, elle nécessite une certaine puissance de calcul et la maintenance d'une importante liste de chemins tabous, et ne peut pas conséquent pas être utilisée sur une carte.

La technique d'énumération implicite des chemins (*Implicit Path Enumeration Technique*, IPET) se sert également du graphe de flot de contrôle afin de générer un ensemble de contraintes qui doivent être respectées par le programme (par exemple, si le bloc B1 aboutit à deux blocs B2 et B3 en raison d'une condition *if*, on aura $N1 = N2 + N3$ (N1, N2 et N3 étant le nombres de fois que les blocs B1, B2 et B3 sont exécutés). Un autre ensemble de contraintes est alors créé avec les informations de bornage de boucles (qui sont soit fournies par le programmeur, soit déduites automatiquement du programme quand cela est possible). Ces contraintes permettent l'élimination de tous les << chemins infaisables >> . En utilisant ces deux ensembles de contraintes et un algorithme de programmation linéaire entière, on tente de maximiser l'expression du WCET :

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

Où n_i est le nombre d'exécutions du bloc de base *i*, et w_i est le coût du bloc de base *i*.

Bien entendu, les calculs requis par cette méthode sont encore énormes comparé à ce que la carte nous propose. Cependant, ses propriétés laissent supposer qu'un calcul en deux phases pourrait être effectué, la partie la plus coûteuse hors-carte et une partie abordable sur carte.

3.1.3. Analyse statique, Camille, et FAÇADE

L'intégration du temps-réel dans Camille pose de nombreuses questions, sur la meilleure manière de remodeler FAÇADE afin d'obtenir des estimations de WCET satisfaisantes et sur le contournement des problèmes posés par le schéma en deux phases de chargement de code dans Camille.

Adaptation de FAÇADE au temps-réel

FAÇADE est un langage intermédiaire extrèmement simple, qui ne dispose que de 5 instructions dans sa forme courante. Le support du temps réel nécessite le respect de certaines contraintes (boucles bornées, etc.), par conséquent une extension de FAÇADE à certaines instructions dédiées au temps-réel, garantissant à coup sûr un calcul de WCET faisable, est envisagée.

Pour permettre le bornage des boucles nécessaire à la méthode IPET, FAÇADE se voit doté d'une instruction `UNTIL` qui délimite une boucle bornée et prend le nombre maximum d'itérations autorisées (ou une expression de celui-ci) en paramètre. Cette instruction sera utilisée en lieu et place des instructions de saut usuelles et son bornage se fera hors-carte par les techniques classiques, afin de pouvoir fournir à la carte une expression constante (mais qu'il faudra vérifier). Les programmes qui ne peuvent être compilés en faisant usage de cette instruction ou qui ne respectent pas les contraintes RT de base comme l'absence de récursion ne pourront simplement pas faire usage du temps-réel.

Les méthodes *Tree-Based* imposent des contraintes plus importantes sur les programmes à analyser, réduisant plus fortement l'ensemble des programmes RT exprimables : les programmes doivent avoir une structure forte en plus d'être bornable, qui doit être transmise à la carte. Celle-ci peut alors calculer le WCET par une simple interprétation abstraite du couple code/structure. L'envoi séparé de la structure du programme peut-être évité si ce dernier est remodelé afin d'exprimer cette structure. Cela nécessite de remplacer les instructions de saut usuelles par des conditions de plus haut niveau, semblables aux conditions *if...then...else* du langage C.

Instanciation et vérification de WCET

Le principal atout de Camille vis-à-vis de la sécurité réside dans son mode de chargement dynamique de code : le programme binaire chargé sur la carte porte sa preuve, qui permet de vérifier son innocuité au chargement (cette technique est connue sous le nom de *Proof Carrying Code* [20]). Ceci permet de faire l'inférence de type hors-carte durant l'assemblage du programme, et la carte n'a plus qu'à vérifier la validité de la preuve durant le chargement. Cette dernière opération est suffisamment légère pour être faite sur la carte, et nous avons prouvé formellement que le processus de vérification de preuve offre le même degré de sécurité que si l'inférence de type était entièrement faite par la carte [22].

L'implémentation du temps-réel dans Camille se doit de suivre la même philosophie : le programme binaire contient la << preuve >> que son temps d'exécution au pire cas est inférieur à une limite. La carte n'a qu'à vérifier cette preuve durant le chargement. Bien sûr, le système de preuve doit être aussi sûr que si la carte avait fait le calcul de WCET elle-même - la preuve n'est qu'un guide permettant d'éviter la plus grosse part de calculs à la carte, et ne doit en aucun cas être falsifiable.

Pour les méthodes IPET, deux problèmes émergent de cette idée : Tout d'abord, l'algorithme de vérification qui est choisi pour maximiser $WCET = \sum_i n_i \times w_i$ doit être vérifiable, *i.e* son résultat doit être formellement et rapidement vérifiable, au besoin avec quelques informations intermédiaires obtenues lors du traitement hors-carte. Ensuite, le WCET doit être << instancié >> sur la carte, car le coût des blocs de base ne peut être connu du monde extérieur. En effet, le WCET d'une instruction FAÇADE dépend directement de l'architecture matérielle cible sur laquelle le programme va être lancé (et ne sera de toute évidence pas le même sur un processeur AVR et sur un StrongArm), ainsi que sur la manière dont le compilateur Camille va traduire et optimiser le code FAÇADE. Cela signifie que les w_is de la formule de maximisation de WCET ne pourront pas être connus. Comment alors maximiser la formule? La connaissance des

n_is serait suffisante pour permettre à la carte d'instancier le WCET, étant donné qu'elle peut calculer rapidement les w_is pendant le chargement du programme. Cependant, les n_is dépendent également de l'architecture et ne peuvent pas non plus être calculés hors-carte : en effet, pour une condition *if* dont les blocs de base A et B sont les alternatives, A peut être plus coûteux en temps que B sur une architecture donnée, tandis que sur une autre le contraire sera parfaitement possible. Dans ce cas, les n_is seront différents pour les deux architectures et ne peuvent pas être calculés par le monde extérieur.

La solution envisageable consiste à donner à la carte suffisamment d'informations pré-calculées supplémentaires pour lui permettre faire le calcul des n_is et des w_is elle-même. En d'autres termes, faire le plus gros du travail hors carte et laisser la carte terminer avec sa connaissance des w_is. Dans ces conditions, il est plus correct de dire que la carte termine un travail commencé à l'extérieur plutôt qu'elle vérifie une preuve. La méthode du Simplex permet de tester rapidement si un résultat est maximal ou pas. Par conséquent, la preuve consiste en un ensemble de données qui donnent des résultats prometteurs dans les cas les plus extrêmes, permettant à la carte de choisir celui qui lui convient le mieux comme point de départ pour terminer la maximisation. Ce faisant, le premier problème est également résolu, étant donné qu'il n'y a aucune vérification à faire, le résultat venant de la carte. Cependant, la sûreté de cette méthode reste aujourd'hui à démontrer (est-il possible de faire calculer de faux résultats à la carte en lui fournissant des données falsifiées?), et le surcoût de transmission de données ainsi que le coût réel pour la carte à estimer.

Si le programme est exprimé sous une forme structurée, les méthodes *Tree-Based* présentent des propriétés intéressantes : l'instanciation est possible et immédiate après avoir calculé le coût des blocs de base, et la validité du WCET calculé est assurée par le fait que l'arbre syntaxique soit contenu dans le programme. Ces avantages couplés à ceux cités précédemment font de cette approche la plus attractive[4].

3.2. Ordonnanceurs temps-réel collaboratifs

Les systèmes temps-réel ont besoin de connaître le WCET des traitements qu'ils supportent (soit par mesure expérimentale, soit par analyse statique). Dans tous les cas, le support du temps-réel dans un exo-noyau nécessite de s'attarder sur les trois points déjà listés auparavant : (*i*) quantifier le temps d'exécution de chaque primitive de l'exo-noyau, (*ii*) trouver un ordonnancement qui satisfait les échéances de toutes les applications en cours d'exécution, (*iii*) permettre aux applications de faire connaître leurs exigences temps-réel.

3.2.1. Temps-réel et primitives d'exo-noyau

Afin d'étendre Camille vers un système d'exploitation temps-réel, nous devons quantifier les temps d'exécution de chaque primitive de l'exo-noyau. Les exo-noyaux n'offrent que des primitives simples pour accéder efficacement et de manière sûre au matériel. Cette simplicité permet de les borner facilement en temps. Des analyses ont été effectuées sur le code du noyau temps-réel RTEMS [1]. Il est également envisageable de faire l'analyse de WCET en compilant le noyau Camille avec GCC et en analysant le code FAÇADE produit avec nos algorithmes et outils de calcul de WCET.

Le principal challenge avec la carte à puce est lié à la mémoire persistance (EEPROM) qui présente des propriétés électroniques ennuyeuses. L'écriture d'une page est lente (env. 5 ms) et empêche tout accès à la page physique courante tout au long du processus d'écriture. Les opérations d'écriture EEPROM impliquent certains blocages matériels qui peuvent poser problème pour estimer le temps d'exécution des primitives. Les récents travaux de [12] proposent des solutions pour faire face à ce problème.

3.2.2. Ordonnanceur et exo-noyau

Connaissant le temps d'exécution requis pour chaque primitive de l'exo-noyau et pour chaque application chargée, le problème est maintenant de trouver un ordonnancement pour un ensemble de tâches standards et RT. Les exo-noyaux n'offrent que l'équité comme stratégie d'accès au microprocesseur en

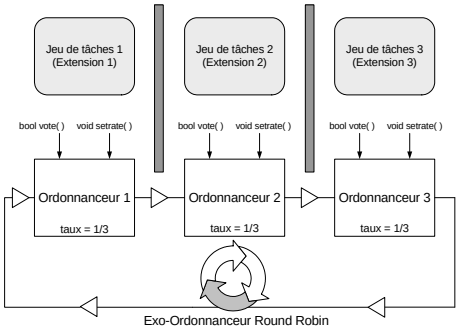

FIG. 4 – *Implémentation d'un ordonnanceur hiérarchique naif pour exo-noyau.*

utilisant une politique << round-robin >> (simple et impartiale). Les quantums de temps sont partages entre les extensions du système qui choisissent alors une application à qui les donner en fonction de leur propre politique d'ordonnancement. Ce processus de partage de CPU à deux niveaux est proche des ordonnanceurs hiérarchiques [21] ou du scheduler RT pour exo-noyau de Zeng et Liu, mais introduit des solutions sous-optimales. Par exemple, la figure 4 montre une implémentation naïve qui permet aux extensions de l'exo-noyau d'utiliser leur propre ordonnanceur pour gérer leurs tâches. Chaque extension a la garantie d'avoir accès au CPU avec un taux de $\frac{1}{N}$, N étant le nombre d'extensions présentes sur la carte. Quand une nouvelle extension est chargée, un vote est organisé pour déterminer si les extensions présentes acceptent le nouveau taux de $\frac{1}{N+1}$, et s'il satisferait toujours leurs échéances s'il était appliqué. Si le vote aboutit, le nouvel ordonnanceur est accepté avec son ensemble de tâches, sinon l'utilisateur est averti de l'échec du vote. Ce modèle n'est pas optimal pour des taches temps-réel car chaque extension essaie d'ordonnancer ses tâches sans chercher à collaborer avec les autres, ce qui peut les amener à rejeter des extensions dont les tâches pourraient être ordonnancées si les extensions collaboraient. Considérons par exemple deux extensions RT avec l'ensemble de tâches suivant : $\{$A (taux $\frac{1}{2}$), B (taux $\frac{1}{5}$)$\}$ comme première extension et $\{$C (taux $\frac{1}{4}\}$ comme deuxième extension. L'ordonnanceur 1 est appelé par la politique de << round-robin >> de l'exo-ordonnanceur et active la tâche A. L'ordonnanceur 2 reçoit alors le quantum suivant et active la tâche C. Puis l'ordonnanceur 1 élit la tâche A et ne dispose plus de temps pour la tâche B qui rate son échéance. Nous présentons dans la section suivante notre politique d'ordonnancement collaboratif qui résoud ce type de problèmes.

3.2.3. Notre approche

Reconsidérons l'exemple de l'ensemble $\{$A,B,C$\}$. Une politique de type *rate monotonic* [18] est capable de l'ordonnancer (l'ordonnancement sera ACABA CABAC ABACA BACA-vide avec une période de 20). La solution présentée précédemment a échoué car l'extension Camille gérant A et B est isolée de celle qui gère C. Notre solution consiste à autoriser le regroupement des extensions RTs qui acceptent de partager leur quantum de temps CPU dans un exo-ordonnanceur collaboratif virtuel, qui fournit certaines garanties quant au niveau de sécurité (*i.e.*, les échéances seront respectées même si les extensions ne se font pas mutuellement confiance). L'architecture et l'interface de l'ordonnanceur sont présentés figure 5. Un des ordonnanceurs faisant partie des ordonnanceurs temps-réel collaboratifs (*Collaborative Real-Time Schedulers*, CRT) se voit donner le droit d'ordonnancer l'ensemble complet de tâches et est désigné comme ordonnanceur actif. Quand l'exo-ordonnanceur collaboratif (*Collaborative Exo-Scheduler*, CES) reçoit un quantum de temps (*i.e.* quand le round-robin invoque la méthode *void pre( )*), le CES appelle l'interface *int plan(Task task[], int t0)* du CRT actif , qui retourne l'*id* de la tâche à exécuter pour le quantum *t0*. Le CES invoque alors *void crtpre(Task t)* sur l'ordonnanceur de l'extension correspondant

à la tâche *id*. A la fin du quantum, le round-robin invoque la méthode *void post( )* afin d'effectuer le changement de contexte de la dernière tâche active (le CES transmet cet évènement à l'extension à laquelle appartient cette tâche). Le problème principal est alors de garantir aux extensions que leurs échéances seront garanties par l'ordonnanceur actif, qui vient d'une extension extérieure à leur domaine de confiance. Cette garantie est possible grâce à un algorithme de vérification qui est appelé durant la phase de chargement et d'installation.

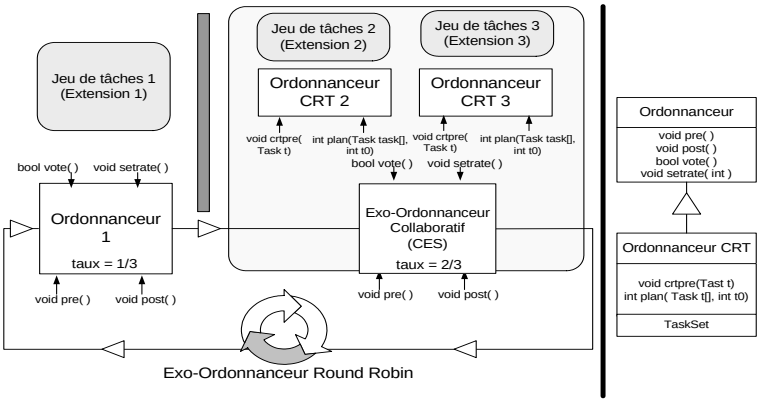

FIG. 5 – *L'ordonnanceur collaboratif de Camille RT et son interface.*

Lors du chargement d'une nouvelle extension collaborative temps-réel, un vote est organisé. En cas de succès, le CES vérifie s'il est possible d'installer la nouvelle extension par rapport à ses extensions et tâches. Le premier problème est de trouver un nouvel ordonnanceur dans l'ensemble des ordonnanceurs collaboratifs qui respecte toutes les échéances des tâches. Le second problème, qui est le plus important, est de vérifier que cet ordonnanceur va effectivement réussir à tout ordonnancer. La méthode d'interface *int plan(Task task[], int t0)* est utilisée pour trouver un ordonnanceur qui correspond. Chaque ordonnanceur potentiel se voit soumettre une liste de tâches qui contient toutes les tâches courantes ainsi que celles provenant de la nouvelle extension. Si l'ordonnanceur n'est pas capable d'ordonnancer les tâches il retourne -1 et le CES demande à un autre ordonnanceur. Une fois qu'un ordonnanceur valable est trouvé, le CES vérifie qu'il va effectivement réussir en lui demandant son plan d'ordonnancement pour l'hyperpériode. La vérification est de complexité $O(\mathtt{hyperperiode})$ et consiste à vérifier que les tâches vont toutes respecter leurs échéances. Le problème est alors de s'assurer que la fonction *plan* qui est une *fonction déterministe non-prouvée fiable* (*i.e.* pas d'état interne, pas d'accès à l'horloge matérielle ou à une quelconque fonction aléatoire), ne puisse biaiser l'ordonnancement des tâches. Une telle propriété peut être obtenue en utilisant un langage de domaine spécifique (Domain Specific Language, DSL) tel que Bossa [17] pour le développement des ordonnanceurs temps-réel. Un problème similaire a été résolu au niveau du code natif des exo-noyaux pour la fonction *own()* qui sécurise l'accès aux méta-données pour la gestion des disques (Chapitre 4 de [10]). Ces techniques pourraient vraisemblablement être utilisées pour le langage intermédiaire FAÇADE .

Conclusion

Cet article a présenté une introduction à l'extension temps-réel de l'exo-noyau pour cartes à puces Camille. Il identifie quatre points clés pour l'utilisation du temps-réel dans une architecture à exo-noyau. Les exo-noyaux conventionnels supportent ce genre d'extensions, mais cet article démontre un exemple

de l'inefficacité des ordonnanceurs temps-réels dans l'approche classique des exo-noyaux. L'approche présentée dans cet article résoud ce problème et détaille les stratégies utilisées pour améliorer le système d'exploitation Camille.
Cet article introduit également deux problèmes ouverts. Le premier est lié au plan d'ordonnancement : prédiction des blocages matériels, impact du multiplexage des interruptions et transmission des évenements matériel aux extensions. Le second est lié au calcul du WCET, le formalisme et la sûreté des algorithmes décrits dans l'article restant encore à prouver.

Bibliographie

1. 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*.
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. Courbot. Calcul réparti de temps d'exécution au pire cas pour petits objects portables et sécurisés. Master's thesis, LIFL/Université de Lille 1, 2003.
5. K. Crary and S. Weirich. Resource Bound Certification. In *the 27th ACM Symposium on Principles of Programming Languages*, 2000.
6. 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.
7. E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, (1):269–271, 1959.
8. J. Engblom, P. Altenbernd, and A. Ermedahl. Facilitating worst-case execution time analysis for optimized code, 1998.
9. Jakob Engblom, Andreas Ermedahl, and Friedhelm Stappert. Comparing different worst-case execution time analysis methods.
10. D. R. Engler. *The Exokernel operating system architecture*. PhD thesis, Massachusetts Institute of Technology (MIT), 1999.
11. 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.
12. Antoine Galland and Mathieu Baudet. Économiser l'or du banquier. In *3ème Conférence Française sur les Systèmes d'Exploitation (CFSE) – French Chapter of ACM-SIGOPS*, La Colle sur Loup, France, October 2003. (to appear).
13. G. Grimaud and D. Deville. Evaluation d'un micro-noyau dédié aux cartes à microprocesseur. *Deuxième Conférence Française sur les Systèmes d'Exploitation*, Mai 2001.
14. G. Grimaud, J.-L. Lanet, and J.-J. Vandewalle. FAÇADE: A Typed Intermediate Language Dedicated to Smart Cards. In *Software Engineering–ESEC/FSE*, 1999.
15. International Standard Organisation: ISO. Integrated circuit(s) cards with contacts, parts 1 to 9, 1987-1998.
16. Oliver Kömmerling and Markus 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, September 2002.
18. C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *ACM*, 20(1):46–61, January 1973.
19. 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.
20. George 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, January 1997.

21. J. Regehr. *Using Hierarchical Scheduling to Support Soft Real-Time Applications on General-Purpose Operating Systems*. PhD thesis, University of Virginia, 2001.
22. A. Requet, L. Casset, and G. Grimaud. Application of the B formal method to the proof of a type verification algorithm. *HASE*, 2000.
23. S. Thibault, J. Marant, and G. Muller. Adapting Distributed Applications Using Extensible Networks. In *the 19th IEEE International Conference on Distributed Computing Systems*, 1999.