UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE

Numéro d'Ordre : 3558 Année : 2004

THÈSE

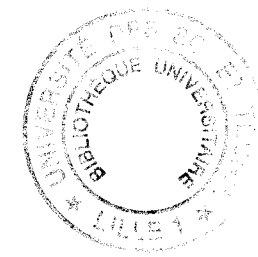
pour obtenir le grade de

DOCTEUR DE L'U.S.T.L.

DISCIPLINE : INFORMATIQUE

présentée et soutenue publiquement,
le 15/12/2004, par

DAMIEN DEVILLE



Titre :

CAMILLERT : UN SYSTÈME D'EXPLOITATION TEMPS RÉEL EXTENSIBLE POUR CARTE À MICROPROCESSEUR

Jury :

Président :	Pierre Paradinas	Professeur titulaire de chaire, CNAM
Rapporteurs :	Sacha Krakowiak	Professeur, Université Joseph Fourier — Grenoble
	Gilles Muller	Professeur, École des Mines de Nantes
Examinateurs :	Vincent Cordonnier	Professeur, Université de Lille 1
	Louis Grégoire	Responsable de la recherche en système, Gemplus SA
Directeur :	David Simplot-Ryl	Professeur, Université de Lille 1
Co-Directeur :	Gilles Grimaud	Maître de Conférences, Université de Lille 1

Résumé

Le logiciel carte est de plus en plus conçu pour supporter des contraintes temps réel. Par exemple, dans les cartes Java SIM, l'application principale est chargée de générer une clef cryptographique de session pour chaque unité de communication consommée faute de quoi l'infrastructure GSM rompt la communication. Actuellement, les systèmes d'exploitation pour carte à puce ne gèrent l'aspect temps réel qu'au cas par cas. Ils ne permettent pas aux applications «utilisateur» de signifier des besoins en termes d'accès au microprocesseur, ceci pour des raisons de sécurité.

Nous proposons une architecture logicielle embarquée autorisant le partage de la ressource microprocesseur entre les extensions de l'exo-noyau CAMILLE. Cette architecture permet aux extensions de supporter des tâches temps réel au dessus de l'exo-noyau garantissant la disponibilité du microprocesseur. Nous avons montré les faiblesses des solutions initialement préconisées pour supporter du temps réel dans les exo-noyaux et nous proposons un moyen de faire collaborer les extensions sous la forme d'un partage d'une de leurs politiques d'ordonnancement et d'une mutualisation de leurs accès au microprocesseur. Nous avons mis en avant les propriétés fonctionnelles que nous attendons de ces ordonnanceurs collaboratifs et nous avons proposé une architecture distribuée permettant de charger et de valider ces propriétés. Cette architecture de partage du microprocesseur a été validée expérimentalement dans CAMILLERT.

Mots-clef : Systèmes d'exploitation, Architectures extensibles, Temps réel, Systèmes embarqués, Carte à microprocesseur.

Abstract

Smartcards software is more and more designed to support real time constraints. For example, in a Java SIM card, the most important application is the one that generates cryptographic session keys. Those keys must be delivered within a firm deadline (one per communication unit used), otherwise the cell phone is disconnected from the network. Current smartcards operating systems only support real time constraints for applications designed by smartcard manufacturers. For security reasons, they are not available at the user application level.

We propose an embedded software architecture allowing to share the access to the microprocessor between all the extensions of the CAMILLE exokernel. This architecture enables extensions to support real time applications on top of the exokernel which guarantees the microprocessor availability. We have highlighted the weaknesses of previously proposed solutions to support real time applications in an exokernel. We propose to allow extensions to collaborate by sharing their access to the microprocessor and also their scheduling policies. We have formalized the functional properties we expect from these collaborative scheduling policies and proposed a distributed architecture that allows their validation. This architecture has been evaluated in the CAMILLERT prototype.

Keywords : Operating systems, Extensibles architectures, Real time, Embedded systems, Smartcards.

À ma famille,
et à mes amis.

Remerciements

Je remercie tout d'abord Vincent Cordonnier pour m'avoir accueilli dans son équipe pendant mon DEA *puis pendant la durée de ma thèse. Je le remercie aussi de sa présence dans le jury de cette thèse.*

Je suis très reconnaissant envers les professeurs Sacha Krakowiak et Gilles Muller pour avoir accepté de rapporter mes travaux. Je les remercie pour les nombreuses remarques formulées suite à la relecture de ce mémoire et qui ont permis de le faire progresser. Je les remercie aussi pour leur présence dans mon jury.

Je remercie Pierre Paradinas pour avoir accepté de présider ce jury de thèse. Je le remercie aussi pour m'avoir accepté au sein de son équipe de recherche pendant mon stage de DEA *et ainsi me donner goût à la programmation des cartes à microprocesseur. J'exprime ma gratitude à Louis Grégoire pour sa présence dans ce jury.*

Mes remerciements les plus sincères à David Simplot-Ryl et Gilles Grimaud pour avoir encadrer mes travaux pendant ces trois années. Je les remercie grandement pour leur soutient, les nombreuses discussions que nous avons menées, et enfin pour leur aide au quotidien.

Je remercie sincèrement tous les membres passé, et présent de l'équipe de recherche RD2P. *Ces trois ans de thèse passés dans cette équipe sont forts des bons moments que nous avons passés ensemble. Je remercie particulièrement mon binôme devant l'éternel, Michaël Hauspie avec qui j'ai partagé les années d'école d'ingénieur, le* DEA *puis la thèse. C'est un ami sur lequel je peux compter, et nous avons partagé beaucoup plus que le travail au cours de ces cinq années. Je ne peux pas oublier les nombreuses soirées pleines de discussions que nous avons passées ensemble. Nous nous sommes supportés, encouragés et guidés pendant nos thèses respectives. Je lui souhaite bonne chance dans ses projets à venir et espère avoir de nouveau l'occasion de travailler et de partager un bureau en sa présence. Je remercie aussi Gilles Grimaud qui non content d'être un très bon encadrant est aussi quelqu'un que je considère comme un véritable ami. Nous avons passé de nombreuses soirées riches en discussions autour de produits provenant de son terroir natal. Je le remercie pour la confiance qu'il m'a toujours accordée et aussi pour m'avoir confié Camille, j'espère en avoir été digne. Je remercie Julien Cartigny pour sa bonne humeur quotidienne et pour les nombreux fous rires résultant de certains épisodes de sa vie de thèsard. Je remercie dans un ordre qui n'a pas d'importance Sébastien Jean, Caroline Fontaine, Hervé Meunier, Rémy Obein, Jean Carle, Farid Naït, Nadia Bel Hadj Aissa, Vincent Benony, François Ingelrest, Mamhoud Taïfour, Kevin Marquet, Antoine Gallais et Alexandre Courbot.*

Tous mes remerciements à Isabelle Simplot-Ryl et Yann Hodique pour la collaboration fructueuse autour de la sûreté du partage sûr service. Je remercie aussi Isabelle Simplot-Ryl pour l'aide qu'elle m'a fourni au quotidien.

Je remercie Marie-Agnes Enard et Jean-Jacques Vandewalle pour leur soutient et pour m'avoir accordé du temps pour discuter de la vie.

Une pensée pour Delphine Jennequin qui se prépare à rédiger sa thèse en mathématique, je te souhaite beaucoup de succès dans cette épreuve.

Je remercie les enseignants d'informatique de l'école d'ingénieur polytech Lille que j'ai d'abord fréquenté en tant qu'élève puis par la suite en tant que collègue : Bernard Carré, Nathalie Devesa, Xavier Redon, David Simplot-Ryl, Vincent Bachelet, Sylvain Janot et les autres ...

Je remercie aussi tous les gens de la société Gemplus avec qui j'ai été amené à travailler pendant ma thèse et mon DEA *: Jean-Louis Lanet, Antoine Requet, Lilian Burdy, Ludovic Casset, Laurent Lagosanto, Fabien Combret, Antoine Galland, Thierry Lamote et les autres ...*

Je remercie tous les stagiaires qui ont travaillé sur le prototype de Camille : Julien Janier, Simon Morvan, Younes Bouchama et Alexandre Courbot. Je remercie tout particulièrement Rémy Obein pour son excellent travail en tant qu'ingénieur, mon prototype de thèse ne serait rien sans lui.

Je tiens tout particulièrement à remercier mes parents qui m'ont toujours supporté et laissé libre de mes choix. Ils m'ont ainsi permis de me construire et d'en arriver où j'en suis.

Je remercie également tous mes amis membres de Melting-Pot, bande d'informaticiens déjantés et d'amis que j'ai plaisir à fréquenter. Merci donc à Nicolas Guillois, Sebastien Wachter, Jean-Paul De Lemos, Vianney Lecroart, Nicolas Lacour, Benjamin Legros, Guillaume Denry, Cécile Lesgoirres, Matthieu Cardon, Marie Weerts, Marie-Pierre Etienne, Christophe Carron, Samuel Ledjmi, Mathieu Durand, Yann Le Maner, Michaël Hauspie, Alban Lecocq, Gabriel Bizzotto, François Bonami, Frédéric Dhieux, Rémy Obein, Grégory Bouillez, Fabrice Lété, Grégory Guche, Hervé Meunier et les autres...

Je remercie enfin mes amis joueurs de snooker qui ont occupé mes lundi soir en me permettant de me libérer l'esprit. Merci donc à : Daniel, Chacal, Vincent, Guy, Renaud, Patrick, Gwenn, Juliette, Jacques, Denis, Yvan, Pascal, Thomas, Hervé, Jérôme, Fred, Christelle, Nicolas, Claire, Christophe et Aurore.

Avant propos

J'ai entendu parler pour la première fois d'exo-noyau au cours d'un stage de seconde année d'école d'ingénieur lorsqu'un de nos professeurs, David SIMPLOT, m'a proposé, ainsi qu'à mon binôme, de travailler pendant l'été à venir sur la réalisation du prototype de thèse d'un certain Gilles GRIMAUD. Ils ont su nous mettre l'eau à la bouche et nous avons décidé d'accepter leur offre en nous faisant aider de deux amis supplémentaires. Au tout début de ce stage, notre compréhension de cette architecture si particulière de système d'exploitation était très vague. Ils ont su nous distiller ponctuellement les bonnes informations, en bonne quantité, afin de nous rendre curieux et désireux d'en savoir plus. Ma compréhension des concepts architecturaux derrière les exo-noyaux s'est affinée de plus en plus au contact de Gilles et des autres membres de l'équipe RD2P. Ce stage nous a conforté dans l'idée de poursuivre nos études dans la branche de la recherche et nous nous sommes donc inscrits en DEA.

Pendant celui-ci, j'ai pu affiner ma compréhension des exo-noyaux en lisant les articles sur les différents travaux du MIT, notamment l'article polémique d'HOTOS sur les abstractions dans les systèmes d'exploitation [EK95], ainsi que le manuscrit de thèse de Dawson ENGLER [Eng98]. J'ai eu la chance d'effectuer mon stage de DEA dans l'équipe de recherche sur la carte à microprocesseur de la société GEMPLUS, où j'ai pu découvrir les différentes facettes de cette informatique si spécifique où un octet est souvent un octet de trop et où il faut mettre en balance performances et empreintes mémoire. Ce stage m'a conforté dans l'idée que l'informatique embarquée fortement contrainte allait être le cadre de mes travaux de recherche à venir.

Gilles GRIMAUD concluait sa thèse en posant cette question : «Comment représenter dans notre micro-noyau le matériel (la pile d'exécution et les mécanismes d'interruptions) qui permettrait de supporter simultanément différentes applications utilisant leur propre représentation du partage du temps machine ?». Ce mémoire est ma réponse à cette question.

Comment lire ce document

Ce document est composé de six chapitres :

Le **Premier chapitre** présente les contraintes du monde des systèmes embarqués. Il

décrit tout d'abord les matériels spécifiques à ce domaine qui sont les responsables directs des difficultés de réalisation des logiciels enfouis. Nous prenons comme exemple les cartes à microprocesseur pour illustrer les évolutions du logiciel enfoui. Les cartes à puce sont un des membres les plus répandu de la famille des Petits Objets Portables et Sécurisés (POPS). La deuxième partie de ce chapitre est consacrée à l'autre grande famille de logiciels enfouis qui est composée des systèmes temps réel. Ces systèmes ont pour but de garantir les temps de réponse de leurs applications face aux stimuli de leurs environnements.

Le **chapitre 2** présente les motivations qui nous ont amené à la définition d'une architecture extensible pour applications en temps réel. Cette architecture se focalise sur la capacité de supporter du temps réel extensible dans un système extensible. Ce chapitre dresse une liste des problèmes auxquels nous avons été confrontés et donne les stratégies que nous avons mises en place pour les résoudre.

Le **chapitre 3** commence par une discussion autour des avantages et inconvénients engendrés par l'isolation naturelle qui existe entre les extensions d'un exo-noyau. Il motive la nécessité de dépasser cette isolation et d'autoriser le partage de service entre extensions pour optimiser l'usage des ressources présentes sur le système physique. Pour rendre ce partage viable, il est nécessaire de rétablir la confiance entre les extensions qui par défaut ne se font pas mutuellement confiance. Ce chapitre se termine par une présentation du canevas que nous avons mis en place dans notre exo-noyau pour valider le partage d'un service entre plusieurs extensions.

Le **chapitre 4** décrit les composants que nous avons mis en place dans CAMILLERT afin de supporter des applications temps réel au niveau des extensions. Nous décrivons tout d'abord les modifications que nous avons apportées afin d'exposer le microprocesseur et les contextes d'exécution des traitements aux extensions. Nous décrivons une première famille d'extension exploitant l'exposition du microprocesseur pour gérer des applications temps réel. Enfin, nous décrivons une deuxième famille d'extensions dites collaborative qui ont pour but commun une meilleure utilisation de leurs accès au microprocesseur, et qui pour y réussir acceptent de partager une politique d'ordonnancement et de mutualiser leurs accès au microprocesseur. Ces extensions sont fédérées par un composant particulier appelé coordinateur qui est chargé de valider le partage de la politique d'ordonnancement en utilisant le canevas présenté au chapitre précédent.

Le **chapitre 5** décrit l'implantation de ces différents travaux dans l'exo-noyau CAMILLE. Ceux-ci nous ont amenés à la réalisation d'un prototype qui a été le cadre de leurs évaluations.

Enfin, le **chapitre 6** conclut ce mémoire en donnant les enseignements que nous pouvons tirer de nos travaux puis en listant quelques pistes de recherches futures autour de CAMILLERT.

Bien que chaque chapitre puisse être lu de manière indépendante, ce document a été pensé comme un tout et nous invitons le lecteur à le lire comme tel.

Chapitre 1

Systèmes embarqués pour le temps réel : objectifs et contraintes

«Au commencement était le verbe. Puis arriva le traitement de texte et leur foutu processeur de pensée. La mort de la littérature s'ensuivit. Ainsi va la vie. Martin Silenius» – **Dan Simmons, Hyperion 1, 1989.**

Ce chapitre présente les différents élements technologiques des systèmes informatiques embarqués supportant des traitements temps réel. La première section de ce chapitre donne les clefs de cette informatique embarquée hétéroclite. En particulier, elle spécifie les contraintes matérielles spécifiques à la famille des Petits Objets Portables et Sécurisés (POPS) et présente les stratégies mises en œuvre par les logiciels enfouis pour les surmonter. Nous nous attachons, ensuite, à donner les principes de base du domaine du temps réel et nous discutons de leurs applications dans le contexte de l'informatique embarquée. Nous concluons cet état de l'art par une discussion autour des problèmes liés à l'extensibilité de ces systèmes embarqués temps réel.

1.1 L'informatique embarquée

Il est intéressant de noter que l'informatique change : elle tend à évoluer vers le développement d'une multitude d'objets informatiques accessibles et personnalisés pour et par les utilisateurs [Wei93, WGB99]. Les gros serveurs de calcul centralisant l'information cohabitent avec une informatique nomade dotée de capacité forte de communication. L'omniprésence se manifeste par le fait qu'actuellement nous trouvons des appareils électroniques dans beaucoup de produits de la vie courante : dans nos téléphones portables, dans nos montres, dans nos machines à laver, ... De plus, la tendance est d'interconnecter tous ces petits appareils les uns aux autres dans le but de les faire interagir. L'acceptation de ces objets mobiles et commu-

nicants par les utilisateurs a été favorisée surtout par leur grande ergonomie. Toutefois, ce facteur clef de l'intégration de ces objets portables et communicants amène des problématiques nouvelles quant au développement du logiciel embarqué et des systèmes d'exploitation embarqués permettant d'exploiter au mieux les spécificités et contraintes de ce domaine. Le logiciel embarqué doit à la fois être très proche du matériel sur lequel il fonctionne, mais aussi facile d'utilisation. De plus, pour pouvoir interagir avec les autres POPS présents dans son environnement, il doit supporter une multitude de normes différentes, en particulier pour les protocoles de communication (WIFI 802.x, BLUETOOTH). Enfin, pour les POPS utilisant des batteries, le logiciel enfoui doit supporter une gestion de l'énergie efficace lui permettant de fonctionner le plus longtemps possible.

Nous nous attacherons dans un premier temps à décrire les spécificités et contraintes matérielles qui caractérisent l'informatique embarquée. Nous nous arrêterons en particulier sur l'exemple de la carte à microprocesseur qui est l'incarnation par excellence de ces «*Petits Objets Portables de Sécurité*». Nous nous intéresserons ensuite à l'évolution des systèmes d'exploitation embarqués, depuis les plateformes fortement monolithiques vers les plateformes extensibles. Finalement, nous discuterons des problèmes de sûreté de fonctionnement de ces systèmes pour POPS.

1.1.1 Matériels de l'informatique enfouie

Les logiciels enfouis sont de plus en plus présents dans la vie de tous les jours. Ils sont utilisés dans des équipements allant du simple four à micro-ondes au téléphone portable de $3^{ème}$ génération. Leur déploiement dans la vie courante, en les rendant omniprésents, les amène à fonctionner sur des plateformes physiques très hétérogènes. Ces plateformes sont caractérisées par des contraintes matérielles fortes ainsi que par la grande diversité et l'aspect disparate des composants électroniques qui les composent. Le choix de ces composants est guidé par des critères de place disponible (facteur de forme), de coûts de production du support physique d'exécution, de consommation énergétique, mais aussi des coûts de développement du logiciel. Ces critères font de l'informatique enfouie un domaine à part entière et ils contribuent grandement à sa richesse. Les développeurs du logiciel embarqué sont donc amenés à gérer de nombreuses déclinaisons d'un même logiciel sur une multitude de configurations matérielles différentes.

Les processeurs utilisés dans l'informatique embarquée vont du simple microcontrôleur dédié à un usage précis au puissant microprocesseur RISC en passant par les circuits FPGA. Il est courant dans l'informatique embarquée d'utiliser des versions adaptées ou dégradées de vieux processeurs de type 68XX de MOTOROLA ou 8086 de INTEL. Ces vieux processeurs 8 bits ont l'avantage d'être économiques en consommation énergétique, en taille du substrat de silicium et sont surtout peu coûteux à produire en grande série. Leur puissance de calcul est suffisante pour la majeure partie des utilisations dans les contrôleurs ou programmateurs des

équipements électroménagers. La famille des assistants personnels portables (PDA) nécessite plus de puissance de calcul et utilise donc des processeurs plus puissants (souvent de type RISC) où la seule concession importante d'un point de vue architectural concerne la maîtrise de la consommation énergétique.

Les capacités et les types de mémoire utilisés dans l'informatique enfouie sont fonction de critères variés comme par exemple la taille de silicium disponible, les besoins de l'application embarquée et enfin la quantité d'énergie fournie (batteries ou alimentation fixe). Les fabricants de plateformes d'exécution enfouies ne mettent souvent que le strict minimum requis par l'application (par exemple quelques bancs de registres pour des capacités de mémoire de travail de l'ordre de quelques octets) afin de diminuer les coûts du silicium.

Beaucoup de systèmes enfouis sont dotés d'une interface de communication propriétaire type bus de terrain ou plus généralement d'une simple ligne série ou parallèle. Les logiciels enfouis exploitent rarement des interfaces de communication vers des humains mais plutôt vers des capteurs ou actionneurs (moteur, manipulateur, alarme ...) leur permettant d'interagir avec leur environnement. Les fabricants fournissent souvent une interface de communication succinte permettant au développeur d'interagir avec le système embarqué en cas de dysfonctionnement de celui-ci. Les objets mobiles nécessitant une plus forte capacité de communication utilisent des interfaces infrarouge ou radio comme les technologies WIFI 802.11 ou BLUETOOTH.

Malgré ces contraintes fortes relatives aux matériels utilisés dans l'informatique embarquée, les utilisateurs demandent de plus en plus de services (client mail, annuaire, recherche d'un service comme par exemple d'impression dans l'entourage de l'objet ...) et d'ergonomie à ces objets communicants. Les développeurs du logiciel embarqué fournissent de plus en plus des systèmes d'exploitation embarqués complets supportant des classes de services proches de celles des systèmes d'exploitation de l'informatique traditionnelle. Ils doivent supporter une interaction entre objets malgré les nombreuses différences de matériels mais aussi de protocoles de communication. Une pratique courante de l'informatique embarquée est appelée *co-design*. Elle consiste à fusionner les phases de conception de la plateforme physique d'exécution et de la plateforme logicielle afin d'obtenir la plateforme dédiée la plus adaptée à une application unique. Le système TINYOS issu des travaux de recherche de l'université de Berkeley [HSW+00] est un exemple de ce type de conception orientée suivant les deux axes logiciel et matériel. Le périphérique obtenu possède une unité de positionnement GPS, une interface de communication radio, un système d'exploitation et un langage de programmation événementiel. Il est utilisé par exemple dans les réseaux de capteurs qui sont des réseaux spontanés formés par dispersion de nœuds communicants chargés de surveiller certains paramètres (température, pression, radioactivité ...) d'un milieu souvent hostile à l'homme.

Un exemple caractéristique de ces objets que nous acceptons et utilisons quotidiennement sans même y prêter attention est celui de la carte à microprocesseur. Les logiciels des cartes à

puce ont évolué pour supporter de plus en plus de services tout en facilitant leur intégration dans les systèmes d'information modernes.

1.1.2 Exemple de la carte à puce

La carte à puce est un des membres les plus représentatifs et les plus répandus de la famille de l'informatique embarquée. Les cartes à puce se caractérisent par des contraintes matérielles fortes. Elles sont usuellement pourvues de processeurs de faible puissance, de capacités d'entrées/sorties limitées, de mémoires de petite taille (allant 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 leur usage intuitif par leur porteur ainsi que leur résistance physique aux attaques en font l'un des objets mobiles de choix (cartes bleues, cartes SIM des téléphones GSM, carte santé ...).

Les cartes à microprocesseur sont des petits objets portables et sécurisés. Elles permettent le chargement dynamique de code, c'est-à-dire après émission de la carte auprès de son porteur[1]. Leurs contraintes matérielles font l'objet de plusieurs définitions au sein de l'ISO [ISO87] dont le but est principalement de garantir à la carte un certain degré de résistance aux attaques physiques [KK99, MDS99]. Les normes ISO ont pour objectifs l'homogénéité, la fiabilité et la sécurité des cartes à microprocesseur.

Modèle	Architecture	Bus de données	Nb (taille) registres	Fréquence (Mhz)
68H05	CISC	8 bits	2 (8 bits)	4.77
AVR	RISC/CISC	8 bits	32 (8/16 bits)	4.77
ARM7TI	RISC	32 bits	16 (32 bits)	4.77 à 28.16
R4KSC	RISC	32 bits	32 (32 bits)	4.77 à 200

TAB. 1.1: Caractéristiques des processeurs les plus utilisés sur carte.

Les classes de microprocesseurs encartés sont très diverses, allant du vieux CISC 8 bits (4,44 Mhz) au puissant RISC 32 bits (100 à 200 Mhz) comme illustré tableau 1.1. Le type de processeur utilisé sur carte dépend en grande partie des normes ISO [ISO99] (notamment celles concernant la résistance de la carte aux torsions et aux flexions). 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 cœurs CISC 8 ou 16 bits). Néanmoins, ces processeurs restent d'un fonctionnement simpliste, ils sont pour la majeure partie dépourvus de cache d'instructions et de données, de circuit évolué de gestion de la mémoire type MMU ou TLB. Ces circuits occupent en effet trop de place sur le substrat de silicium qui est limité à $27mm^2$. Toutefois, beaucoup de cartes sont pourvues de coprocesseur additionnel de cryptographie car elle est d'un usage primordial pour les domaines d'utilisation des cartes à microprocesseur (accélération des algorithmes de cryptographie RSA ou DES).

[1] De l'anglais *post-issuance*.

Il existe différents types de mémoire sur une carte. La première est la RAM (Random Access Memory) qui est utilisée comme mémoire de travail. On y trouve aussi de la ROM (Read Only Memory) pour stocker le code, ainsi que de l'EEPROM (Electric Erasable Programmable Read Only Memory) ou de la FlashRam, qui sont des mémoires persistantes réinscriptibles. Étant donné que la surface de silicium de la carte est limitée à $27mm^2$, le point mémoire (unité élémentaire de surface de silicium requise pour stocker un bit de mémoire) est un facteur important. Une carte à puce classique dispose de 2 à 4 Ko de mémoire de travail, 32 à 128 Ko de mémoire persistante et 64 à 256 Ko de ROM. Le tableau 1.2 résume les caractéristiques principales de ces différentes mémoires.

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	1 à 64 octets
RAM	x 20	128 à 4096 octets	$\leq 0.2\mu s$	1 octet

TAB. 1.2: Caractéristiques des mémoires cartes.

La mémoire persistante a un inconvénient majeur lié à ses propriétés électroniques. Son délai d'écriture est jusqu'à 10000 fois plus important que celui de la RAM. De plus, l'écriture répétée en mémoire persistante peut endommager ses cellules (ce phénomène est appelé *stress* de la mémoire).

Les recherches en électronique proposent des innovations marquantes en particulier dans le domaine de la mémoire. La FeRAM (Ferro-Electric Random Access Memory) et la MRAM (Magnetic Random Access Memory) sont des nouveautés majeures pour les mémoires carte. Ce sont des mémoires persistantes dont le point mémoire (taille sur le substrat de silicium) s'approche de celui de la ROM et leur délai d'écriture est celui d'une mémoire RAM. La seule contrainte pour la FeRAM [AMO+98], est que celle-ci *oublie* les données : à chaque lecture d'une donnée, celle-ci est perdue. Ainsi, chaque donnée doit être explicitement réécrite après chaque lecture. Cette contrainte soulève des problèmes de sécurité importants. Un autre problème est que la FeRAM est bâtie sur une unique couche d'aluminium. Cette technologie est plus facilement attaquable d'un point de vue physique. Son intégration dans le bloc monolithique impose aussi une perte de place pour faire coexister couche d'aluminium et de cuivre (technologie CMOS). Aussi, l'utilisation de la technologie FeRAM dans les cartes à puce est encore en cours d'évaluation.

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 de transport «T=x». La normalisation sans fil (pour les cartes sans contact physique) est définie dans l'ISO SC17-14443. «T=0» et «T=1» sont les protocoles filaires 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).

Au dessus de ces protocoles de transport se trouve le protocole APDU qui sert de protocole applicatif orienté client-serveur. La carte est vue comme un serveur dont la seule fonction est de répondre aux questions que pose le terminal. Ce protocole permet en particulier au terminal de commander les traitements réalisés par la carte. En effet, les temps de réponse aux questions du terminal sont bornés par la norme ISO. Si la carte n'a pas fini en temps et en heure elle doit envoyer un octet signalant qu'elle est encore en vie faute de quoi le terminal la considère comme morte et coupe son alimentation. En particulier, la carte doit produire, à chacun de ses démarrages, une suite d'octets appelée ATR[2]. L'ATR contient des données permettant d'identifier la carte (bancaire, SIM, santé ...). Il sert aussi à négocier les paramètres physiques de transmission des données (bits de parité, ordre des bits ...) utilisés par la couche de liaison des protocoles de communication.

1.1.3 Systèmes d'exploitation enfouis

Au cours des vingt dernières années, le logiciel enfoui a subi des évolutions majeures. Le logiciel enfoui était initialement rigide et monolithique. Il a, par la suite, évolué vers des architectures plus flexibles basées sur une séparation entre les préoccupations présentes au niveau du système d'exploitation et celles présentes au niveau des applications. Nous utilisons la classification des systèmes embarqués pour carte à microprocesseur proposée dans [DGGJ03b, DGGJ03a] pour décrire l'évolution du logiciel enfoui. Cette classification est rappelée figure 1.2 et les familles qu'elle distingue pour le domaine de la carte à puce restent d'actualité pour l'informatique embarquée.

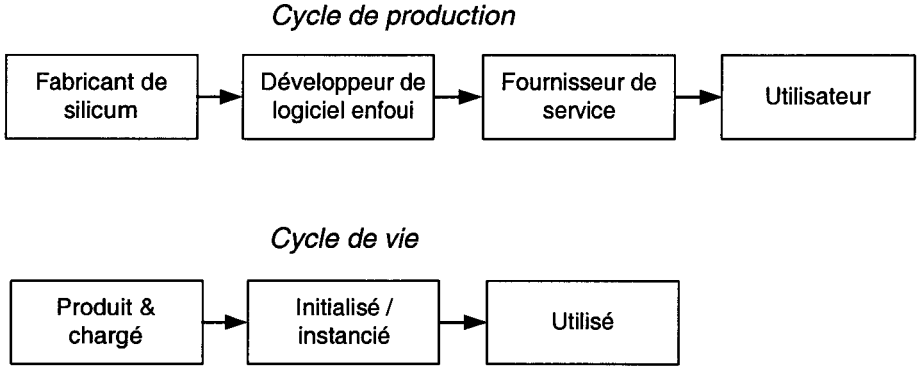


FIG. 1.1: Cycle de vie et de production du logiciel embarqué.

Le logiciel enfoui a son propre cycle de vie et de production, comme illustré sur la figure 1.1, mettant en œuvre différents acteurs allant du fabriquant de silicium au développeur de logiciel ou fournisseur de service. Le logiciel enfoui est tout d'abord *produit* et *chargé* dans un support physique d'exécution. Par la suite, il sera *initialisé* et *instancié* pour supporter une ou plusieurs

[2]De l'anglais Answer To Reset, qui peut se traduire par réponse au redémarrage.

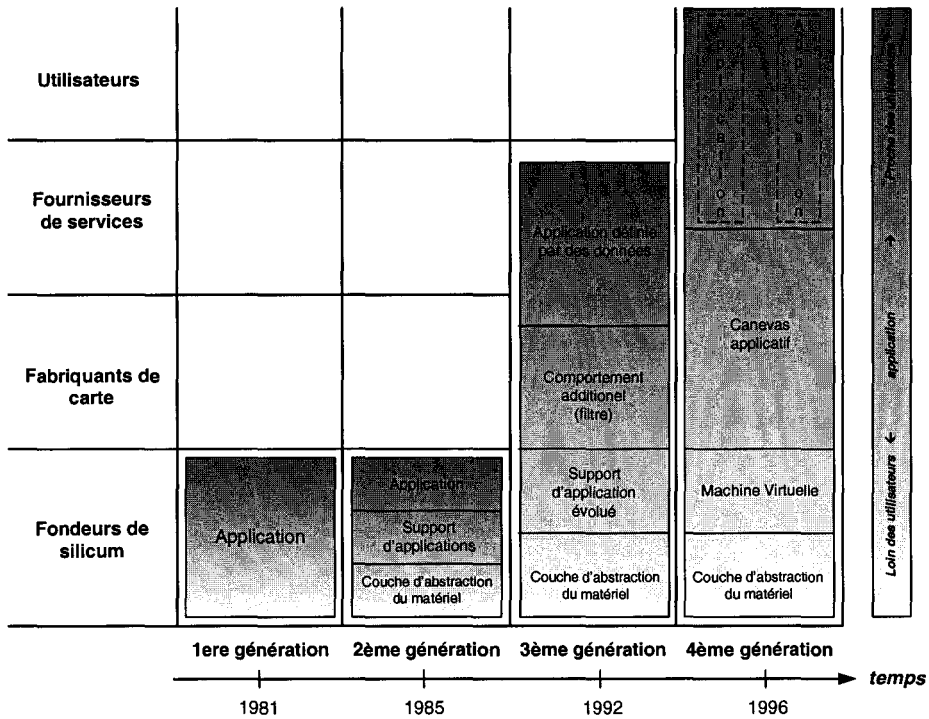


FIG. 1.2: Modèle d'évolution des systèmes embarqués.

applications données. Finalement, il sera *utilisé* en fonction de son domaine d'application.

Les architectures logicielles enfouies étaient initialement des blocs logiciels monolithiques parfaitement adaptés aux besoins du client final ainsi qu'aux caractéristiques physiques d'un matériel cible. On regroupe tous ces systèmes monolithiques dans la *1ère génération* de logiciels embarqués. Une fois le logiciel produit par le développeur embarqué, celui-ci était donné sous forme de masque (*i.e.* image binaire du logiciel et de ses données) au fabricant de silicium qui produisait un bloc de silicium final comprenant le code associé au microprocesseur ou microcontrôleur. Jean-Jacques QUISQUATER a décrit l'utilisation de cette génération de logiciels enfouis dans les premières générations de carte à microprocesseur [Qui97]. Au fur et à mesure de la standardisation des architectures matérielles utilisées, les producteurs de logiciels enfouis ont réalisé que de plus en plus d'éléments de logiciels, en particulier les éléments liés à la gestion du matériel, étaient communs à différents logiciels produits. Cette réutilisation de logiciel caractérise le logiciel enfoui de *2ème génération* qui est construit sur une architecture en trois couches :

- La première couche comprend des primitives de gestion du matériel ;
- La seconde couche se compose des modules communs à plusieurs classes d'application (gestion de code d'identification personnel[3], gestion du code natif, allocation mémoire, ...) ;

[3]De l'anglais PIN : Personal Identification Number.

– Finalement, la dernière couche regroupe le code propre à l'application embarquée.

Les deux premières couches traitent des préoccupations logicielles que partagent tous les producteurs de système d'exploitation. Cette *2ème génération* de logiciels enfouis est aussi appelée architecture à base de bibliothèques logicielles[4]. Toutefois, ces bibliothèques sont toujours gravées par le fondeur sur le silicum. Ces deux premières générations sont encore fortement présentes dans beaucoup d'équipement de consommation courante (machine à laver, four à micro-ondes [GMGA02], domotique, ...) du fait de leur faible coût de production et aussi de la faible empreinte mémoire du logiciel. Les coûts de développement de ce type de logiciels sont très élevés mais ils sont rendus négligeables parce qu'ils sont dupliqués en de nombreux exemplaires.

La *3ème génération* de logiciels enfouis a émergé principalement pour répondre à des considérations de temps de développement. Les producteurs de logiciels enfouis ont pris conscience que 90% des applications qu'ils produisaient pouvaient être dérivées à partir de plateformes dédiées comme des systèmes de fichiers ou des moteurs de base de données. Dans ce cas précis leur motivation première a évolué pour devenir : «Quelle plateforme est la mieux adaptée aux besoins de mon application ?». Le logiciel enfoui s'est donc séparé en deux parties distinctes :

– La première partie est constituée d'une plateforme d'exécution dédiée qui peut être vue comme une combinaison de bibliothèques de gestion du matériel et de bibliothèques orientées applications ;

– La seconde partie est composée du code de l'application et de ses données propres.

Les fournisseurs de services peuvent par la suite paramétrer et ainsi configurer le fonctionnement des applications qui ont été préalablement chargées. La carte à microprocesseur compte beaucoup de ces systèmes basés sur des bases de données ou systèmes de fichiers. Les deux exemples les plus répandus sont les cartes SIM des téléphones portables GSM et aussi les cartes bancaires EMV (consortium de définition de la carte bancaire formé par `Europay`, `Mastercard` et `Visa`).

Par la suite, les développeurs de logiciel enfoui ont cherché à pousser ces concepts plus loin et ont proposé des plateformes d'exécution ouvertes à base de machine virtuelle qui forment la *4ème génération*. L'évolution ultime et la plus utilisée dans le contexte des cartes à puce est la configuration Java pour carte (JAVA CARD) qui a été définie par Sun [Che00, SM]. La JAVA CARD permet de décliner une plateforme d'exécution commune à une large classe d'applications et par la suite d'étendre les capacités de cette plateforme en chargeant le code nécessaire à une ou plusieurs applications. La carte à puce compte d'autres exemples de plateformes ouvertes comme par exemple MULTOS [Mao], une plateforme permettant de charger des applications écrites en C et compilées en P-code MEL, ou les cartesW4SC [Mic] de MICROSOFT.

Plus généralement les logiciels enfouis de *4ème génération* proposent un degré de flexi-

[4]De l'anglais LIBOS.

bilité plus fort quant à leur utilisation. La même plateforme de base peut être déclinée en plusieurs configurations supportant différents services au niveau applicatif (module SIM pour les téléphones portables, porte-monnaie électronique ...). Même si cela apparaît attrayant, l'utilisation de ces plateformes reste très limitée en pratique. Si l'on prend l'exemple de la JAVA CARD, son utilisation reste confinée de par les fortes spécificités de sa programmation (bibliothèques de programmation dédiées *à la Java* mais qui n'en sont pas : pas d'allocation dynamique ni de ramasse miette, ...). De plus, la carte est difficile à intégrer au sein des systèmes d'information de l'ubiquité numérique à cause de son protocole de communication spécifique et marginal (APDU [ISO89, ISO94]). Finalement, ces plateformes de *4ème génération* se limitent à être multi-applicatives. Une fois la plateforme déployée sur le système embarqué, aucun support ne permet de modifier fortement son fonctionnement.

1.1.4 Extensibilité des systèmes embarqués

Pour supporter la capacité de modifier le fonctionnement du système d'exploitation après son déploiement, les chercheurs se sont intéressés aux architectures à base de noyaux. Ces noyaux supportent une extensibilité au niveau système en autorisant le chargement dynamique de services système. Nous allons détailler trois de ces architectures.

a) HIPERSIM [5] est un système d'exploitation pour carte à microprocesseur proposé conjointement par la société MOBILE-MIND et le fabricant de silicium FUJITSU [Gut]. HIPERSIM est basé sur un noyau multitâche de type MACH [ABB+86]. C'est un des premiers systèmes d'exploitation pour carte supportant l'exécution concurrente et simultanée de plusieurs applications. HIPERSIM est principalement orienté vers une utilisation dans le secteur des téléphones de troisième génération, principalement dans les modules d'authentification de type SIM ou USIM. Le multitâche permet de dépasser le modèle d'exécution de la carte à puce (requête du terminal, traitement par la carte, émission d'une réponse) mais aussi de faciliter l'intégration de celle-ci dans les réseaux de communication GSM. Toutefois, HIPERSIM exploite des technologies mémoire de type FERAM qui, comme expliqué précédemment section 1.1.2, diminuent fortement la résistance de la carte aux attaques physiques. Dans l'état actuel, les cartes l'embarquant ne respectent pas les critères de sécurité du standard ISO [ISO87, ISO88].

b) CAMILLE est un système d'exploitation ouvert pour carte à microprocesseur issu des travaux de thèse de Gilles GRIMAUD [Gri00]. CAMILLE est basé sur les principes architecturaux des exo-noyaux [Eng98]. Le noyau, en particulier, n'impose aucune abstraction du matériel et a pour unique vocation de démultiplexer et sécuriser son exploitation [EK95]. CAMILLE fournit un accès sécurisé aux ressources matérielles et logicielles (microprocesseur, pages de mémoire, interface de communication série, blocs élémentaires de code natifs ...) et permet aux applications de gérer leurs ressources de manière directe.

[5] De l'anglais High-Performance Smart IC Manager.

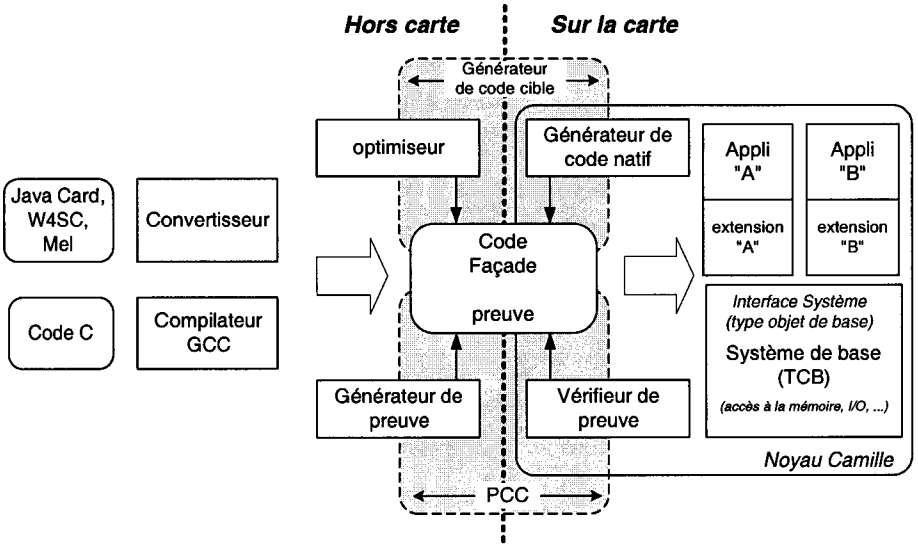


FIG. 1.3: Architecture logicielle de CAMILLE.

CAMILLE a été conçu pour garantir la portabilité, l'extensibilité, la confidentialité et l'intégrité des applications et des extensions système. La portabilité est assurée par l'utilisation du langage intermédiaire orienté objet FAÇADE [GLV99]. FAÇADE est un langage intermédiaire simple et compact composé de seulement cinq instructions[6] : trois instructions de branchement (`jump`, `jumpif` et `jumplist`), un `return` et une instruction `invoke`. L'instruction `invoke` permet d'appeler une des opérations exportées par les composants formant le système CAMILLE.

Ainsi, un programme FAÇADE peut être vu comme une suite de liens et d'interactions entre les différents composants qui forment le système embarqué [DRG04]. Les applications et extensions système peuvent être programmées à l'aide de langages de haut niveau comme le C ou le Java. Par la suite, elles sont converties en FAÇADE en utilisant des convertisseurs de code ou un compilateur dédié. La version actuelle de CAMILLE propose un convertisseur de code Java vers FAÇADE ainsi qu'une version de la suite de compilation GCC permettant de générer du FAÇADE à partir de codes source écrit dans un sous-ensemble du langage C.

Le terminal peut ensuite optimiser certains aspects du code FAÇADE (durée de vie des variables, réduction de la taille du code, ...). Les plateformes d'exécution construites à l'aide de CAMILLE sont extensibles au niveau applicatif mais aussi au niveau système. Le code FAÇADE une fois chargé sur l'équipement est traduit en code natif par un générateur de code à la volée [GD01] lorsque celui-ci a des besoins forts en termes de performances d'exécution (ceci est vrai en particulier pour les extensions système). Ces performances s'obtiennent au prix d'une expansion de la taille du code.

[6] On peut parler d'approche RISC.

La confidentialité et l'intégrité des applications sont garanties à l'aide d'une vérification de typage [RR98]. Une preuve de typage est calculée par le terminal sur le programme FAÇADE. Les extensions sont ensuite validées par le système embarqué à l'aide d'une vérification linéaire [RCG00a] effectuée pendant la phase de chargement. La figure 1.3 reprend l'architecture répartie de CAMILLE.

c) Les différents travaux de recherche menés autour de l'architecture THINK ont pour but de fournir des outils d'aide à la construction de systèmes d'exploitation flexibles. Les systèmes THINK reposent sur un nano-noyau qui comprend le code minimal nécessaire au démarrage, à l'initialisation et à la gestion du matériel. Ce nano-noyau n'impose aucune abstraction du matériel, il ne fait qu'en donner une vue au programmeur système.

THINK fournit aussi une bibliothèque de composants système supportant les différents services nécessaires à la programmation de systèmes d'exploitation (gestionnaire de disque et de mémoire, pilote de carte réseau, ...). Ces composants sont écrits de manière indépendante (utilisation des interfaces des composants donc possibilité de remplacer un composant par un autre supportant les mêmes opérations) et respectent le modèle de composant de THINK. THINK propose un canevas logiciel (modèle de programmation de composants systèmes), permettant de faciliter la construction et la mise en oeuvre de systèmes flexibles.

THINK, dans sa version initiale décrite dans [FSLM02], introduit les concepts de *composants*, *interfaces*, *liaisons*, *noms* et *domaines*. Les composants THINK sont proches des composants présents dans le modèle ODP. Un composant est une unité logique regroupant des données et des traitements. Un composant supporte une série d'opérations qui sont publiées aux autres composants sous la forme d'interfaces. Les liaisons sont les canaux de communication permettant de lier plusieurs composants dans le but de les faire interagir. Une liaison permet de lier un composant à une interface d'un autre composant en utilisant le nom symbolique de celle-ci. Les noms sont regroupés au sein de contextes de nommage et servent à désigner les interfaces des composants. Les liaisons sont créées et mises en oeuvre à l'aide de composants particuliers appelés usines à liaison. Les composants partageant une même propriété sont regroupés au sein d'entités appelées domaines.

L'architecture THINK a été modifiée afin de respecter le modèle de composant FRACTAL en vue de construire des systèmes dynamiquement reconfigurables [SCS02, Sen03]. Ce modèle ajoute en particulier la notion de *contrôleur* permettant de capturer l'état interne d'un composant et, ce faisant, d'adapter dynamiquement le fonctionnement d'un système THINK par modification ou reconfiguration des composants qui le forment. Une version simplifiée de THINK basée sur un modèle à base d'évènements permet de construire des systèmes embarqués reconfigurables. Un système reconfigurable basé sur ce modèle événementiel de THINK à été évalué expérimentalement sur des briques LEGO RCX [CS02, Cha04].

Ces trois systèmes ont des différences et des points communs. HIPERSIM est la réponse des industriels de la carte à microprocesseur pour augmenter la réactivité de la carte en ga-

rantissant la disponibilité du microprocesseur. C'est une évolution temps réel des systèmes d'exploitation pour carte à microprocesseur. CAMILLE et THINK abordent des problématiques similaires concernant l'extensibilité système en exploitant des modèles et architectures à base de composants logiciels. CAMILLE propose des solutions pour permettre le chargement dynamique de composants système. CAMILLE exploite un compilateur embarqué et un mécanisme de vérification de code qui ont fait leurs preuves dans l'informatique fortement contrainte qu'est le domaine de la carte à puce. THINK est un peu plus qu'un système car il s'intéresse à proposer des solutions permettant d'aider à la construction du système d'exploitation par assemblage de composants. Toutefois, THINK était jusqu'à peu orienté gros systèmes, son utilisation sur des petits objets portables et sécurisés reste encore à valider.

1.1.5 Sûreté de fonctionnement des systèmes enfouis

Les systèmes enfouis peuvent être classés en plusieurs familles suivant leurs besoins en termes de sûreté de fonctionnement. Les systèmes enfouis depuis la *4ème génération* supportent le chargement dynamique d'applications et donc peuvent par ce même processus charger des applications dites malignes visant à compromettre l'intégrité du système ou à récupérer les données confidentielles du système ou de l'utilisateur. Il s'agit donc d'un problème de sûreté de fonctionnement des codes mobiles qui se décompose en trois sous-problèmes :

- la confidentialité qui consiste à garantir qu'une application est la seule et unique à pouvoir lire ou écrire ses données privées ;
- l'intégrité qui garantit que les données ou traitements d'une application ne seront pas violés par une autre application ;
- la disponibilité qui consiste à garantir qu'un traitement aura suffisamment de ressources pour arriver à fonctionner.

Les mécanismes de protection permettant de garantir la confidentialité et l'intégrité des données et traitements peuvent être classés en deux familles. On trouve tout d'abord les mécanismes dits statiques qui sont mis en œuvre pendant la phase de chargement du code. On trouve ensuite les mécanismes dynamiques qui nécessitent un support côté système permettant de contrôler l'exécution des traitements.

Les systèmes d'exploitation pour cartes à microprocesseur, de par leur domaine d'utilisation, nécessitent de fournir au minimum les garanties de confidentialité et d'intégrité. Les systèmes industriels MULTOS et W4SC exploitent un mécanisme d'isolation logicielle dynamique [WLAG93] au niveau de l'interpréteur.

Les JAVA CARD intègrent le même mécanisme que JAVA concernant les droits d'accès aux champs et méthodes des classes (privé, publique, sous-classe ...). La machine virtuelle doit donc effectuer des contrôles dynamiques à l'exécution des bytecodes de lecture et d'écriture d'un champ d'une classe ou d'appel d'une méthode car elle est dépourvue d'un vérifieur de type. Pour permettre une plus grande souplesse de programmation des classes, les JAVA

CARD exploitent aussi un pare-feux permettant à une classe de s'isoler des autres classes. Une interface de l'API JAVA CARD permet à une classe de demander la liste des références partagées par une autre classe; celle-ci pouvant refuser de lui donner accès à l'un de ses membres ou à une de ses méthodes.

Les JAVA CARD étaient jusqu'ici dépourvues de vérifieur de bytecode en grande partie pour des raisons de contraintes en termes de capacité mémoire et de puissance de calcul [Dev01]. Des solutions alternatives ont été proposées pour palier à ce manque. La première consiste à embarquer dans le code un certificat (à la Eva ROSE [RR98]) permettant de valider le typage de celui-ci en appliquant les principes des codes auto-certifiants (*Proof Carrying Code* [NL97]). CASSET et al ont utilisé la méthode B dans le cadre du développement sûr d'un vérifieur de bytecode pour JAVA CARD [CBR02, Cas02]. Xavier LEROY propose une technique alternative utilisant des règles de réécriture au niveau du bytecode permettant de garantir qu'une variable de travail ne change pas de type au cours des différents chemins possibles du programme [Ler02]. Cette technique consiste à dupliquer les variables polymorphes et contribue donc à une augmentation sensible du nombre des variables de travail. Toutefois, la phase de vérification par la carte s'en trouve simplifiée. CAMILLE utilise le langage intermédiaire fortement typé FAÇADE permettant d'utiliser une vérification de typage embarqué à la Eva ROSE.

Ces solutions, mêmes si elles répondent aux problèmes liés à la confidentialité et à l'intégrité du code, compliquent la chaîne de production et de déploiement du logiciel en introduisant des phases supplémentaires pour générer le certificat ou transformer le code. Une approche orientée système utilisant un système performant de caches de données entre les mémoires volatile et persistante de la carte et exploitant un codage des informations de types non stressant [DGR01, BCDR01] pour la mémoire persistante a permis d'embarquer un vérifieur complet dans une JAVA CARD. Cette solution a été évaluée sur une plateforme de type AVR et possède une empreinte mémoire ainsi qu'une consommation mémoire tout à fait acceptable dans le monde de la carte à puce [DG02, CDL02].

L'autre facette importante de la sécurité concerne les garanties de disponibilité des différentes ressources présentes sur un système. Le système d'exploitation doit pouvoir garantir à un traitement qu'il disposera de suffisamment de ressources pour fonctionner. Les ressources dont on cherche à garantir la disponibilité sont principalement la mémoire et le microprocesseur. De telles garanties sont complexes à obtenir dans un système multitâche.

Les systèmes embarqués exploitent souvent des mécanismes par contrat [SG02]. Chaque traitement est pourvu d'un contrat décrivant ses besoins en termes de ressources. Le système embarqué peut ainsi vérifier à l'admission d'un nouveau traitement qu'il a la capacité de lui fournir les ressources nécessaires à son fonctionnement. Le système s'engage à fournir au traitement ces ressources. Toutefois, il doit contrôler dynamiquement que le traitement ne va pas utiliser plus de ressources que spécifié dans son contrat. Pour éviter cette surveillance

dynamique, il est possible d'utiliser des analyses statiques ou des systèmes de type [CW00, Hof00] pour borner statiquement la consommation d'une ressource comme la mémoire. Les JAVA CARD utilisent de la preréservation à l'installation pour garantir à une application qu'elle aura suffisamment de mémoire pour fonctionner. Une application va donc réserver la totalité de la mémoire dont elle a besoin lors de son installation, ainsi elle pourra fonctionner à n'importe quel moment quelque soit le nombre d'applications présentes sur la carte. Cette solution entraîne une utilisation non optimale de la mémoire. GALLAND et al [GB03a, GB03b] proposent une architecture exploitant un ordonnanceur permettant de minimiser l'usage d'une ressource. Ils exploitent une phase d'analyse statique hors carte pour calculer la consommation d'un traitement. Un ordonnanceur basé sur l'algorithme du banquier de DIJKSTRA exploite ces informations sur la carte pour ordonnancer les différents traitements en minimisant l'usage de la ressource et en évitant les interblocages.

Une autre ressource souvent abordée dans les systèmes embarqués est la ressource énergetique. Toutefois, très peu de travaux ont abordé ce problème dans le contexte des cartes à microprocesseur. Ceci s'explique par le fait que la carte ne contrôle pas sa source d'énergie qui lui est fournie par le terminal dans lequel elle est installée.

Garantir la disponibilité de la mémoire ou du microprocesseur est un problème clef pour les systèmes embarqués supportant le chargement dynamique d'applications. En effet, le système peut facilement être amené à charger un code qui monopolise une de ces deux ressources et qui réduit ainsi fortement ou totalement la qualité de service du système.

1.2 Le temps réel

Si l'on regarde les motivations de l'évolution des systèmes d'exploitation (standards ou embarqués), on s'appercoit qu'ils ont évolué pour supporter de plus en plus de services et de fonctionnalités et ainsi être plus proches des besoins de leurs utilisateurs. L'évolution majeure concerne le support de l'exécution concurrente de plusieurs traitements et a donné naissance aux systèmes dits multitâches. Ces systèmes permettent de supporter les différents traitements que souhaitent effectuer les utilisateurs mais aussi tous les traitements nécessaires au bon fonctionnement du système. Toutefois, les systèmes multitâches même s'ils permettent de partager l'accès au microprocesseur n'offrent que peu de garanties concernant la disponibilité de cette ressource. Un traitement peut monopoliser le microprocesseur et ainsi diminuer les performances globales de tout le système et de ses applications.

Un système temps réel garantit à ses traitements qu'ils auront à coup sûr le temps d'arriver à leur terme. Les systèmes temps réels sont donc une réponse au problème de disponibilité de la ressource microprocesseur. Ils sont couramment utilisés dans des milieux ou les garanties de temps de réponse sont vitales. Ils sont présents dans les systèmes de contrôle de processus industriels mais aussi dans les systèmes de gestion du trafic aérien qui doivent réagir rapidement à des informations diverses comme par exemple celles provenant d'un radar.

Le temps réel est un domaine à part entière de recherche qui est le sujet de nombreuses études multi-domaine (logique temporelle, théorie des files d'attente, systèmes d'exploitation ...). Nous nous intéresserons dans cette section à présenter les spécificités de ce domaine. Tout d'abord nous donnerons les motivations inhérentes à la maîtrise du temps et quelques solutions pour y réussir. Nous listerons ensuite les fondamentaux nécessaires à la bonne compréhension du temps réel en nous attardant sur les stratégies d'ordonnancement. Enfin, nous nous intéresserons à l'utilisation du temps réel dans l'informatique embarquée.

1.2.1 La maîtrise du temps : motivations

La maîtrise du temps est un problème important dans l'informatique. Il souléve deux questions auxquelles nous allons tenter de répondre :

- « *Pourquoi maîtriser les temps d'exécution des traitements ?* » ;
- « *Comment maîtriser les temps d'exécution des traitements ?* ».

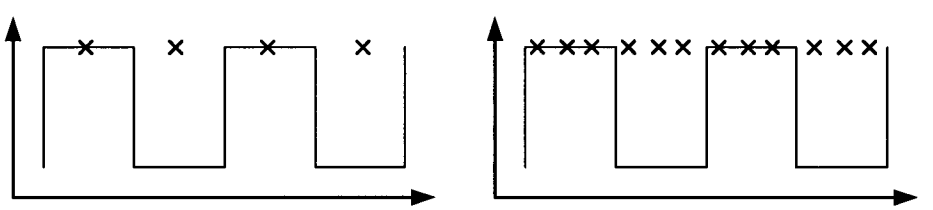


FIG. 1.4: Échantillonage de la ligne série.

La motivation première derrière la maîtrise des temps d'exécution des traitements concerne le contrôle de son comportement. Borner l'exécution d'un traitement permet d'obtenir des garanties quant aux temps de réponse aux divers ensembles de données auxquels il pourra être soumis. Plus précisément cela permet de garantir la terminaison d'un traitement et aussi de mesurer la quantité de processeur nécessaire à sa terminaison. Ce contrôle est en particulier nécessaire dans beaucoup de traitements proches du matériel dans les systèmes d'exploitation standards et embarqués. Ces traitements ont souvent des temps de réponse imposés par le matériel qu'il est impossible de remettre en cause.

Prenons l'exemple du traitement responsable de la réception d'un bit de communication dans une carte à microprocesseur. Comme décrit précédemment section 1.1.2, la majeure partie des cartes à puce sont dépourvues de circuit de type UART[7] et doivent donc gérer manuellement l'échantillonnage de la ligne série permettant la réception des bits de données. La carte à puce se doit de tenir au minimum des taux de transfert de l'ordre de 9600 bauds dans le cas du protocole de transport ISO7816-3. Une carte à puce étant cadencée à 4.73 Mhz en fréquence externe, cela laisse 492 cycles[8] pour le traitement d'un bit de communication

[7] Composant électronique universel de réception et transmission asynchrone.

[8] Ce nombre tombe à une quarantaine de cycles dans le cas d'un taux de transfert de 115000 bauds.

```
{
        t0 <- COM0.GetState
        t1 <- SerialLine.GetField &current_byte
        t1 <- t1 <<B #1
        t1 <- t1 |B t0
        SerialLine.PutField &current_byte t1

        t1 <- SerialLine.GetField & nb_bits
        t1 <- t1 +B #1
        t0 <- t1 ==B #8
        jumpif t0! L1
        /* l'octet n'est pas encore disponible */
        SerialLine.PutField &nb_bits t1
        jump L2
L1:
        SerialLine.PutField &nb_bits #0
        SerialLine.PutField &byte_ready #1
L2:
        return
}
```

FIG. 1.5: Code FAÇADE du traitement d'un bit de communication.

dans le cas où l'on effectue une mesure par créneau sur la ligne série comme illustré figure 1.4. Le code FAÇADE correspondant au traitement d'un bit de communication donné figure 1.5 doit donc s'exécuter en moins de 492 cycles sous peine de perdre des bits. Ce code est attaché sous interruption à un compteur matériel pour être appelé de manière périodique. Si la ligne série comporte des défauts, on effectue alors plusieurs mesures par créneau et on prend la valeur moyenne mesurée comme illustré à droite de la figure 1.4. Pour trois mesures par créneau à 9600 bauds il reste donc un peu plus d'une centaine de cycles pour recevoir et traiter le bit. Le code correspondant pour traiter les bits de données doit donc s'exécuter dans ce court laps de temps.

Dans le cas où l'UART est embarquée, l'arrivée d'un octet de donnée entraîne la génération d'une interruption sur le microprocesseur ce qui permet de minimiser le traitement nécessaire à la réception et ainsi de limiter les pertes de données. Souvent les UART possèdent une file en réception de quelques octets, le problème devient donc de ne pas prendre plus de n octets de retard si cette file est de taille n.

Dans un système muni d'une carte réseau il est nécessaire de traiter les données d'un paquet avant réception du suivant ou de fournir à la carte une nouvelle adresse mémoire d'une zone libre où stocker le prochain paquet. Les temps de réponse sur ce type de traitements proches du matériel sont garantis par bonne construction du code par un programmeur expert. Toutefois, plus le code est complexe plus il devient difficile de borner son exécution par bonne construction. Les règles de bonne écriture de code deviennent inutilisables dans le cas de code applicatif. Le code applicatif est souvent complexe et il n'exploite pas directement le matériel mais utilise des abstractions de celui-ci.

Un autre point important concernant la maîtrise du temps est propre au domaine des cartes à puce. Dans un contexte sécuritaire, maîtriser les temps d'exécution d'un traitement donné

peut aider à augmenter la sécurité du système. Prenons l'exemple d'une carte à puce utilisée dans le bancaire ou dans la téléphonie. Celle-ci réalise certains traitements de cryptographie comme par exemple des phases de génération de clef, de chiffrement et de déchiffrement de données. Si un attaquant arrive à isoler une portion de code de cryptographie et arrive a soumettre plusieurs jeux de données judicieux à la carte, il peut ainsi en observant les temps de réponse ou la consommation énergétique de la carte déterminer des informations sur une partie des données secrètes utilisées par l'algorithme de cryptographie [HKQ99, DKL+00]. Ce type d'attaque est d'usage courant dans le milieu de la carte à puce et les systèmes embarqués proposent des contre-mesures afin de lutter contre les attaques temporelles ou énergétiques. La contre-mesure la plus simple et la plus efficace consiste en une maîtrise totale des temps d'exécution de ces traitements sensibles. L'idée est de les rendre *temps-constant*, c'est-à-dire que les temps de réponse soient les mêmes pour tous les chemins possibles du traitement quelles que soient les données d'entrée du traitement. Ce bornage se fait dans l'état actuel des choses manuellement par bonne connaissance et écriture du code par des experts des algorithmes de cryptographie et aussi de la plateforme cible [JV02]. Une problématique similaire [IIT04, Joy04] permet de rendre ces traitements *courant-constant* afin de lutter contre les attaques basées sur la scrutation énergétique [MDS99].

1.2.2 La maîtrise du temps : mise en œuvre

La maîtrise des temps d'exécution restant un pré-requis des systèmes temps réel, différentes techniques permettant leur calcul sont utilisées. Le but n'est pas de calculer précisément les temps d'exécution d'un traitement, mais d'obtenir une borne majorante de ce temps d'exécution appelé «Temps d'exécution au pire cas» [9]. Rappelons qu'il est impossible de calculer les WCET dans le cas général car cela revient à calculer la terminaison d'un programme.

Colin et al dressent un état de l'art complet des différentes techniques de calcul de WCET dans [CPRS03]. Ces techniques peuvent être classées en deux grandes familles. On trouve d'abord les méthodes par estimations et mesures expérimentales, puis la famille des méthodes exploitant des phases d'analyse sur le code source et le code généré.

Les méthodes expérimentales consistent à tester le traitement face à divers jeux de données bien choisis afin de mesurer les temps de réponse correspondants. Le problème principal vient du choix des données servant aux tests expérimentaux : pour que les temps de réponse mesurés soit exploitables, il faut que ceux-ci correspondent soit au pire cas *réel* que pourra rencontrer le système, soit au pire cas théorique. De plus, il est difficile voire quasi impossible de trouver le pire cas possible dans l'ensemble des données d'entrée d'un traitement quand celui-ci devient d'une taille conséquente. De même, prouver que le temps d'exécution mesuré est proche du majorant est complexe dans le cas général. C'est pourquoi ce type de techniques se limite à des utilisations ponctuelles pour des petits traitements bien spécifiques caractérisés par des

[9] De l'anglais WCET : Worst Case Execution Time.

domaines de données d'entrée restreints.

La deuxième famille de technique utilise des phases d'analyses statiques de code et de son flot de contrôle [Col01] afin d'obtenir une valeur pour le WCET ou une formule permettant de le calculer. Ces analyses statiques de code peuvent se ranger en deux sous-familles :

- les méthodes travaillant sur le graphe de flot de contrôle du programme ;
- les méthodes travaillant sur l'arbre syntaxique du programme.

Un graphe de flot de contrôle d'un programme est un graphe dont les sommets sont les différents blocs de base du programme (*i.e.* une section linéaire sans point de branchement) et les arcs les sauts qui relient les blocs entre eux. Notons B_i les différents blocs de base, w_i le coût de ces blocs et n_i le nombre de fois où l'on passe par chaque bloc de base. Ainsi, pour obtenir le WCET d'un programme on utilise un algorithme de recherche du plus long chemin dans son graphe de flot de contrôle (*e.g.* algorithme de DIJKSTRA). Ces méthodes se limitent donc à des programmes dont l'exécution est bornable c'est-à-dire des programmes dépourvus de boucles non-bornées ou alors des programmes possédant des boucles non-bornées qui ont été annotées par le programmeur. Une variante de ces méthodes par analyse du graphe de flot de contrôle est la méthode d'énumération implicite des chemins [LM95, EES00][10] permettant d'obtenir l'ensemble des contraintes d'un programme. Chaque fois qu'un lien existe entre deux blocs on peut en déduire une relation au niveau de leurs n_i. On obtient ainsi un ensemble d'équations auquel on ajoute les informations de bornage des boucles (obtenues par annotations du programmeur par exemple) pour former un ensemble d'inéquations. Ainsi, le WCET d'un programme se calcule en maximisant l'égalité 1.1 à l'aide d'un algorithme de programmation linéaire comme par exemple la méthode du Simplex.

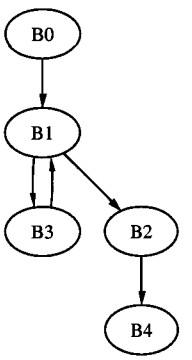
$$WCET = Max(\sum_i n_i \times w_i). \tag{1.1}$$

Considérons l'exemple simple donné figure 1.6 comportant une boucle bornée par annotation du programmeur à 10 itérations. On obtient un système d'inéquation donné figure du milieu qui se résout trivialement, dans notre cas, avec les solutions données à droite. En exploitant les coûts des blocs de base, on obtient ainsi le temps d'exécution au pire cas du traitement.

Les méthodes de la deuxième sous-famille de calcul de WCET utilisent des représentations plus haut niveau des programmes à analyser. Elles exploitent en particulier une représentation structurée obtenue par transformation à partir du code source exprimé dans un langage de haut niveau (C, C++, Ada ...). Cette représentation offre plus d'informations que le simple graphe de flot de contrôle. Les nœuds de cet arbre sont des séquences, des conditions et des boucles bornées et les feuilles sont les blocs de base formant le programme à analyser. Les blocs de base sont composés d'instructions élémentaires ou d'appels de méthode.

Le WCET s'obtient en parcourant l'arbre depuis la racine jusqu'aux feuilles et en appli-

[10] De l'anglais IPET : Implicit Path Enumeration Technique.



$$
\begin{array}{rcl@{\qquad}rcl}
n_0 & = & 1 & n_0 & = & 1 \\
n_1 & = & n_0 + n_3 & n_1 & = & 11 \\
n_4 & = & n_2 & n_2 & = & 1 \\
n_2 + n_3 & = & n_1 & n_3 & = & 10 \\
n_3 & \leq & 10 & n_4 & = & 1
\end{array}
$$

FIG. 1.6: Application de l'IPET sur un exemple simple.

quant les assertions suivantes pour chaque nœud :

$$WCET(Seq(A,B)) = WCET(A) + WCET(B). \quad (1.2)$$

$$WCET(If) = WCET(Test) + WCET(Then). \quad (1.3)$$

$$WCET(If - Else) = WCET(Test) + Max(WCET(Then),WCET(Else)). \quad (1.4)$$

$$WCET(Loop(n)) = n \times (WCET(Test) + WCET(Body)) + WCET(Test). \quad (1.5)$$

Si l'on applique ces formules sur l'exemple donné figure 1.7, on obtient le temps d'exécution au pire cas suivant :

$$WCET = w_1 + 10 \times (w_2 + (w_3 + max(w_4,w_5)) + w_2 + w_6.$$

Les bornages de boucles sont obtenus soit par annotation du programmeur soit par inférence sur le code (propagation des constantes par exemple). Les coûts des blocs de base sont calculés par analyse sur le code machine généré. On doit en particulier prendre en compte les appels de méthode ce qui peut devenir complexe dans le cas d'un langage objet ou d'un pointeur de fonction pour lequel on ne sait pas précisément quelle méthode sera appelée. Ainsi la phase d'analyse de haut niveau effectuée sur le code source est indépendante de la plateforme matérielle cible. Toutefois, cette technique est très sensible aux optimisations des compilateurs qui changent la structure du programme (factorisation de code, injection de code, élimination de code mort ...).

De nombreux outils de calcul du WCET exploitant ces deux familles d'analyses existent, le plus connu étant certainement HEPTANE. HEPTANE [Col01] est un outil de calcul de WCET composé de plusieurs modules. Les deux modules principaux sont le module d'analyse de haut niveau exploitant l'arbre syntaxique du programme pour obtenir une formule symbolique du WCET et le module qui calcule le coût des blocs de base au niveau du langage machine. Le découpage retenu dans HEPTANE lui permet de supporter plusieurs processeurs [CP01a]. En effet, pour bénéficier d'HEPTANE pour un nouveau microprocesseur il suffit d'écrire le

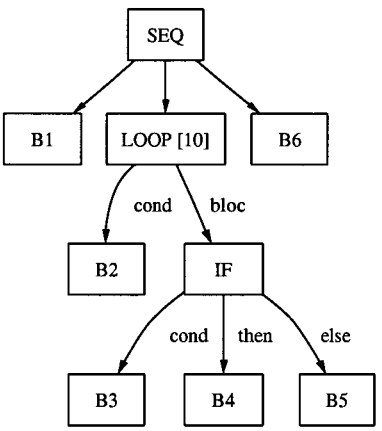


FIG. 1.7: Exemple d'arbre syntaxique d'un programme.

module d'analyse bas niveau correspondant. HEPTANE a notamment été utilisé pour analyser le code complet [CP01b] du système d'exploitation temps réel *open-source* RTEMS [RTE]. Nous pouvons aussi citer les travaux de ENGBLOM et al visant à définir un outil de calcul de WCET adapté à l'informatique embarquée [EES01]. Leur but est de proposer un outil capable d'analyser du code C quelconque pour une utilisation dans un contexte industriel de développement de logiciel embarqué.

De nombreux problèmes existent concernant le calcul de WCET. Le premier problème concerne les excès de pessimisme. En effet les différentes méthodes présentées précédemment visent à obtenir un majorant pour les temps d'exécution au pire cas et ce faisant, elles considèrent les cas les plus défavorables au niveau du matériel (défaut de cache à chaque instruction, mauvaise prédiction de branchement ...). HEPTANE utilise par exemple des techniques de prédiction fine du comportement du matériel afin d'améliorer l'exactitude du WCET calculé. ENGBLOM et al utilisent des phases de simulation du comportement des caches d'instructions et de données. On peut citer les travaux autour de la prédiction de branchement [CP00] et ceux autour du comportement des caches [Pua02, AP03].

L'autre problème est issu des analyses mixtes effectuées en parallèle sur les représentations de haut niveau extraites du code source et sur le code machine compilé. Il est en effet complexe de garder une cohérence entre ces deux représentations. Les compilateurs effectuent souvent des optimisations qui détruisent la structure du code (injection du code des fonctions dans le code de l'appelant, compression de code ...). Les outils de calcul de WCET sont souvent amenés à devoir simuler le comportement du compilateur pour éviter les divergences entre les analyses symboliques et les analyses de bas niveau.

1.2.3 Principes des technologies temps réel

Une fois que l'on dispose d'outils et de méthodes permettant de calculer les temps d'exécution au pire cas des traitements, il convient de proposer des solutions au niveau du système d'exploitation afin d'exploiter ces informations pour permettre l'exécution concurrente de ces traitements tout en leur garantissant la disponibilité de la ressource microprocesseur.

Les traitements que l'on trouve dans un système d'exploitation peuvent être classés en différentes familles. On trouve tout d'abord les tâches standards ou sans contraintes de temps. Ces tâches forment la majeure partie des traitements orientés utilisateur. Ensuite viennent les tâches ayant des besoins quant à leurs temps de réponse :

- les tâches *périodiques* sont des tâches qui sont l'image de traitements se répétant de manière régulière comme par exemple le contrôle régulier de l'état d'un capteur physique ou l'échantillonnage de la ligne série de communication ;
- les traitements dit *apériodiques* ou à instant fixe sont des traitements pouvant avoir des garanties concernant leurs temps de réponse. Ce sont des traitements ponctuels qui se déclenchent à des temps particuliers ou en réponse à un changement de l'environnement (traitement d'alerte en cas de dépassement d'une variable mesurée comme la température, la pression, la radioactivité, dans un système de contrôle de processus industriel par exemple) ;
- les traitements dits *sporadiques* qui sont des traitements *apériodique* possédant une durée minimale qui va s'écouler entre deux activations potentielles du traitement.

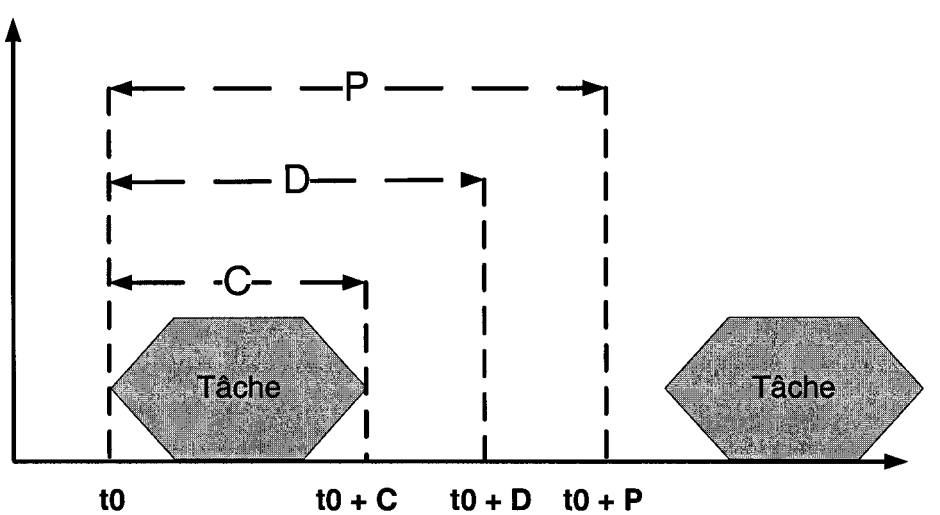


FIG. 1.8: Exemple de tâche périodique.

Le formalisme le plus utilisé et accepté dans la communauté temps réel pour les tâches périodiques a été proposé en 1973 par LIU et LAYLAND [LL73]. Une tâche est caractérisée par le t-uple (t_0,C,D,P) comme illustré figure 1.8. t_0 est le temps d'arrivée de la tâche, c'est-à-dire

le temps à partir duquel la vie de la tâche démarre. C est la quantité de ressource d'exécution nécessaire à la terminaison de la tâche, c'est-à-dire son temps d'exécution au pire cas (aussi appelé coût d'exécution en référence aux algorithmes d'allocation de ressources). Ainsi une tâche débutant au temps t_0 doit avoir reçu C ressources d'exécution avant le temps $t_0 + D$ pour respecter son échéance temporelle noté D. Finalement, P est la période de la tâche, c'est-à-dire le temps qu'il faut attendre avant de pouvoir lancer la prochaine exécution de la tâche. Beaucoup d'algorithmes d'ordonnancement définissent et exploitent la progression d'une tâche : quantité de ressources d'exécution nécessaire à un instant donné pour terminer le traitement. On appelle $L(t)$ le compte à rebours avant exécution d'une tâche[11]. C'est un nombre dépendant de la progression courante de la tâche (quantité de ressource qu'elle a déjà reçue) quantifiant la marge temporelle qu'il reste à l'instant t avant d'être dans l'obligation d'exécuter la tâche pour que sa terminaison soit synchronisée avec son échéance. Cette marge se déduit du t-uple de la tâche et de sa progression à l'instant t à l'aide de la formule suivante :

$$L(t) = D - (C - Progression(t)) \tag{1.6}$$

La *laxité* est donc bornée pour tout traitement par $D - C$. C'est-à-dire que si un traitement n'a pas encore été exécuté il devra au pire être exécuté pendant C ressources, au plus tard C ressources avant son échéance.

Enfin, Il existe deux grandes familles de systèmes temps réel : les systèmes dits *mous* et les systèmes dits *durs*[12]. La famille des systèmes *mous* permet le non-respect des échéances temporelles d'un traitement. La priorité de ces systèmes est d'exécuter au mieux les différents traitements (stratégie de type *best-effort*). C'est à dire de chercher à respecter les échéances temporelles du plus grand nombre de tâches. Ces systèmes sont beaucoup représentés dans les domaines liés au multimédia. En effet un lecteur de musique ou de vidéo portable peut se permettre de perdre quelques images (dans des limites acceptables suivant le type de compression des données utilisé) sans pour autant nuire fortement à la qualité du flux audio ou vidéo. Les systèmes *durs* quant à eux, interdisent le non-respect des échéances d'une tâche. Ils utilisent donc des tests pendant la phase d'admission des tâches permettant de garantir qu'une fois la tâche acceptée ses échéances seront respectées. Ces systèmes sont utilisés dans des domaines où il est vital de garantir les temps de réponse à certains évènements comme par exemple dans un téléphone portable GSM qui doit crypter le flux de communication. L'appartenance à une de ces familles joue fortement sur la manière dont le système va fonctionner et en particulier sur le type d'algorithme pouvant être utilisé pour ordonnancer les tâches du système.

[11] De l'anglais *laxity*.

[12] On parle aussi de système critique et non critique dans la littérature.

1.2.4 Stratégies d'ordonnancement

Une fois que l'on dispose de la connaissance des tâches à exécuter, il convient de trouver un ordonnancement de ces tâches. Un algorithme d'ordonnancement peut être vu comme un ensemble de règles permettant d'élire la tâche à exécuter à tout moment de la vie d'un système [LL73]. On peut donc considérer l'ordonnancement comme un algorithme d'allocation d'unités élémentaires de temps appelées *quantum de temps*. Dans le cas d'un système mono-processeur, l'ordonnanceur doit élire la tâche à exécuter. Dans le cas d'un système multi-processeur, l'ordonnanceur doit élire plusieurs tâches (autant que de processeurs) et aussi gérer le placement de celles-ci sur les différents processeurs en respectant certains critères propres à l'algorithme utilisé comme par exemple équilibrer la charge entre les différents processeurs ou encore minimiser la migration des tâches. Dans notre cas nous nous intéresserons uniquement aux algorithmes mono-processeur. En effet l'informatique enfouie, de par ses contraintes matérielles présentées section 1.1.1, utilise rarement des architectures multi-processeurs qui sont réservées à une informatique orientée haute performance utilisant du matériel plus conventionnel.

Les algorithmes d'ordonnancement peuvent se ranger dans deux grandes familles. On trouve tout d'abord les algorithmes *hors-ligne* qui nécessitent une connaissance à l'avance de l'ensemble des tâches qui vont s'exécuter sur un système ainsi que de leurs caractéristiques temps réel. Cette famille utilise des algorithmes lourds en temps d'exécution et en consommation mémoire afin de trouver le meilleur ordonnancement pour cet ensemble de tâches et uniquement pour celui-ci. L'ordonnancement *hors-ligne* est d'utilisation courante dans les systèmes de contrôle de processus industriels et aussi dans certains systèmes pour lequel il est essentiel d'avoir testé et maitrisé le comportement complet du système en réponse à tout évènement auquel il se doit de faire face comme par exemple dans le domaine de l'avionique. Les algorithmes *hors-ligne* sont peu dynamiques. Il est en effet impossible d'accepter une nouvelle tâche une fois le système en fonctionnement si cela n'a pas été prévu à l'avance pendant le calcul du plan d'ordonnancement. C'est pourquoi ils restent cantonnés à des domaines où les systèmes sont fermés et orientés haute fiabilité (automatique, processus industriels, avionique ...). L'autre famille regroupe la majeure partie des ordonnanceurs utilisés dans les systèmes et se compose des algorithmes *en-ligne* qui déterminent l'ordonnancement dynamiquement en fonction des critères des tâches. Ces algorithmes utilisent des critères ou tests d'admission afin de déterminer si l'ajout d'une tâche ne va pas rendre l'ensemble des tâches non-ordonnançables (*i.e.* au moins une tâche ne va plus respecter ses échéances).

Une autre caractéristique importante des ordonnanceurs s'énonce par la question suivante : «Que faire lorsqu'une nouvelle tâche plus prioritaire que celle qui est en train de s'exécuter demande à s'exécuter ?». Les ordonnanceurs non-préemtifs suggèrent de ne pas arrêter l'exécution de la tâche courante et de démarrer la nouvelle tâche quand la tâche courante aura fini de s'exécuter. Ce sont les tâches elles-mêmes qui rendent la main au système en appelant une

primitive du système nommée *yield()*[13]. Ainsi tout processus qui prend la main s'exécutera à terme ou rendra la main à l'ordonnanceur afin que celui-ci élise une nouvelle tâche. Dans le cas où un processus décide de manière volontaire de monopoliser le microprocesseur ou alors s'il rentre dans une boucle infinie il est quasi-impossible de rétablir la qualité de service du système. Les ordonnanceurs non-préemtifs garantissent donc peu de qualité de service, ils s'intéressent seulement au partage de la ressource d'exécution. Ces techniques sont souvent couplées à des ordonnanceurs *hors-ligne* : on calcule hors-ligne l'ordonnancement optimal pour le jeu de tâches à réaliser puis on modifie le code des applications en y insérant les appels à *yield()*. On obtient ainsi le support d'exécution optimal pour le jeu de tâches en question. Les algorithmes préemptifs quant à eux suspendent l'exécution de la tâche moins prioritaire afin de démarrer la nouvelle tâche de priorité supérieure. Ces algorithmes sont les plus répandus dans les systèmes d'exploitation temps réel.

Parmi les ordonnanceurs préemptifs on trouve la famille des ordonnanceurs à priorité statique et celle à priorité dynamique. Pour les ordonnanceurs à priorité dynamique, la priorité d'une tâche est recalculée dynamiquement en fonction de ses caractéristiques temporelles mais aussi en fonction de la charge du système et de l'état des autres tâches présentes sur le système. Pour les ordonnanceurs à priorité statique, la priorité de la tâche est fixée par un expert (programmeur ou concepteur du système) pendant la phase de conception du système ou alors en utilisant un critère pendant la phase d'admission des tâches.

Nous allons maintenant lister et décrire quelques uns des algorithmes d'ordonnancement les plus utilisés et cités dans la littérature. On trouve tout d'abord l'ordonnanceur équitable *Round-Robin*. Les tâches ont accès les unes à la suite des autres au processeur. *Round-Robin* peut être implémenté sous forme non-préemtive en utilisant une fonction *yield()* pour améliorer le partage ou alors sous forme préemtive en exploitant un compteur matériel pour changer de tâche à chaque *quantum* de temps. Ce n'est pas à proprement un ordonnanceur temps réel, toutefois il est utilisé sous des formes évoluées dans beaucoup de systèmes d'exploitation comme par exemple dans LINUX qui utilise des classes de processus ayant des priorités différentes et ordonnance les processus d'une même classe à l'aide d'un *Round-Robin*. Les deux algorithmes qui sont les pères de beaucoup d'ordonnanceurs sont *Rate-Monotonic* et *Earliest-Deadline-First* proposés par LIU et LAYLAND en 1973 [LL73] pour ordonnancer des traitements périodiques. DERTOUZOS a proposé en 1974 une variante de leurs travaux pour ordonnancer des tâches dans un cadre plus général ne se limitant pas uniquement aux tâches périodiques [Der74]. *RM* attribue une priorité statique à chaque tâche inversement proportionelle à sa périodicité. *EDF* attribue une priorité dynamique à chaque tâche en fonction de leurs échéances. La tâche qui a l'échéance la plus proche obtiendra donc la priorité la plus forte. La politique *EDF* définit un critère d'admission ou d'ordonnançabilité exploitant le

[13]Que l'on peut traduire par rendre la main.

taux d'occupation U du processeur :

$$U = \sum_{i=0}^{n} \frac{C_i}{P_i} \tag{1.7}$$

n est le nombre total de tâches périodiques présentes sur le système, P_i est la période de la $i^{ème}$ tâche et C_i le nombre de quanta de temps qu'elle a besoin. Si ce taux d'occupation est inférieur à un, le jeu de tâche correspondant est ordonnançable par la politique *EDF*. *EDF* sert donc de référence pour vérifier qu'il est possible d'ordonnancer un ensemble de tâches periodiques. L'algorithme *Least Laxity First* est une variante de *EDF* qui attribue le processeur à la tâche qui a la plus faible marge d'exécution (*cf* équation 1.6). Dans le cas où deux tâches ont les mêmes caractéristiques, *LLF* va alterner l'exécution de ces tâches là où une politique de type *EDF* aurait fini totalement une tâche avant de lancer l'autre.

Beaucoup d'algorithmes d'ordonnancement existent [RS94], chacun visant à répondre à un problème donné ou à optimiser un critère précis [CM94] (minimisation de la longueur d'ordonnancement, réduction du nombre de quanta de temps libres, favoriser l'entrelacement des tâches). Comme l'illustre STANKOVIC dans [SSNB95], chaque famille d'ordonnanceur a ses limites (dynamicité *vs* prévisibilité, résistance à la surcharge ...) et de ce fait il n'existe aucun ordonnanceur optimal pour une large famille de tâches appartenant à différents domaines d'application, mais plutôt une multitude de politiques différentes.

1.2.5 Temps réel pour systèmes enfouis

L'informatique embarquée a donné lieu à de nombreuses études autour des politiques d'ordonnancement permettant d'économiser la ressource énergétique. En effet, l'informatique embarquée est souvent nomade et utilise des batteries comme source d'énergie. Différentes solutions sont possibles pour minimiser la consommation énergétique de l'objet. L'idée la plus simple est d'éteindre le processeur, de le mettre en veille plus précisément, lorsqu'aucune tâche n'en a besoin. Si l'ordonnanceur n'est pas adapté, on risque de passer son temps à éteindre et rallumer le processeur. Or, les coûts de commutation du processeur sont souvent élevés. Éteindre puis rallumer le processeur a un coût énergétique qui doit être amorti par l'économie réalisée pendant le temps où le processeur ne travaille pas. Il convient donc d'utiliser un ordonnancement qui regroupe les exécutions des tâches quand cela est possible et qui en contrepartie dispose de longues périodes d'inactivité. Lorsque ces périodes atteignent un certain seuil, il devient rentable d'éteindre et de rallumer le processeur.

Une autre approche a été proposé dans le système VERTIGO [FM02b] utilisé principalement dans les baladeurs MP3. VERTIGO exploite un processeur ARM à fréquence variable. Partant de l'idée que les changements de fréquence ont un coût qui doit être amorti par l'économie d'énergie qui doit en résulter, VERTIGO exploite les informations temporelles des tâches afin de déterminer la plus basse fréquence qui respecte encore les échéances des tâches mais qui

permet d'économiser de l'énergie. La fréquence est choisie de telle sorte que chaque tâche se finisse au plus tard tout en respectant son échéance. Ainsi en diminuant la fréquence le système consomme moins d'énergie. De plus la charge du processeur est ainsi mieux équilibrée car cette approche a tendance à gommer les plages de temps où le processeur était non utilisé. Des approches similaires ont été présentées dans [QH01, DMZ02, YK03] dont le but est de trouver un ordonnancement de tension sur des processeurs supportant plusieurs niveaux de tension d'alimentation.

Beaucoup de noyaux temps réels existent, qu'ils soient des projets commerciaux (pSOS, QNX [Hil92], VxWorks) ou des projets de recherche (Spring, RT-Mach). Toutefois, ces architectures sont peu adaptées au domaine de la carte à microprocesseur et de ses contraintes. En effet elles visent et nécessitent toutes des architectures possédant des puissances de calcul et des capacités mémoire plus standard. À l'opposé de ces *gros* noyaux, on trouve beaucoup de petits noyaux de systèmes dont le seul but est de gérer l'exécution concurrente de plusieurs traitements. Picos [EA03] est un modèle de programmation permettant d'exécuter plusieurs traitements écrits sous forme de *coroutines*. Un traitement se résume donc à une fonction ayant plusieurs points d'entrée. Une *coroutine* possède sa propre pile d'exécution. Les commutations sont explicites, ainsi le système se résume à gérer le redémarrage d'une nouvelle tâche lorsqu'une *coroutine* libère le microprocesseur. Le plan d'ordonnancement peut être soit dynamique soit calculé *hors-ligne*.

Une approche similaire a été présentée dans [HKM03]. Cet article décrit une adaptation du PCC[14] [NL97] à l'ordonnancement de processus. Un compilateur exploite des algorithmes d'ordonnancement *hors-ligne* afin de tisser la politique d'ordonnancement dans le code des applications[15]. Le système embarqué se réduit à fournir un support d'ordonnancement simplifié permettant le gel et l'activation de tâche ainsi qu'un algorithme permettant de vérifier que le plan d'ordonnancement embarqué dans le code est effectivement faisable. Cette approche part du principe qu'il est plus facile de vérifier un plan d'ordonnancement que de le calculer.

Emeralds [ZPS99] est un micro-noyau de système temps réel conçu pour servir de système d'exploitation pour des objets embarqués dotés de capacités de communication. Emeralds a une empreinte mémoire de 13Ko sur un processeur Motorola 68040 et exploite un ordonnanceur hybride basé sur les algorithmes *Rate Monotonic* et *EDF*. Emeralds gère deux files de tâches. Les tâches de priorité dynamique sont ordonnancées selon une politique *EDF*, les tâches de priorité fixe sont ordonnancées à l'aide d'un *Rate Monotonic*. Si l'on considère un ensemble de n tâches triées suivant leur priorité statique et que la tâche d'indice r rend le jeu non-ordonnançable par la politique *RM*, les tâches de 0 à r sont insérées dans la file dynamique, le reste dans la file statique. Tant que des tâches sont activables dans la file dynamique, la tâche de plus faible échéance prend la main. Dans le cas contraire, la tâche de priorité la plus forte de la file statique prend la main. Cette politique hybride permet d'ordonnancer

[14] De l'anglais Proof Carrying Code.
[15] De l'anglais Schedule Carrying Code.

plus de jeux de tâches que les politiques *EDF* ou *RM* prise séparément. EMERALDS fourni aussi un mécanisme léger de communication inter-processus et aussi des sémaphores adaptés à l'embarqué [ZS97].

La carte à puce possède un noyau de système industriel temps réel appelé HIPERSIM basé sur un architecture MACH comme décrit précédemment section 1.1.4. HIPERSIM utilise les algorithmes d'ordonnancement classiques de la littérature. Les travaux de Sébastien JEAN autour l'architecture AWARE [Jea01, DGGJ03b] sont le point de jonction entre les cartes multi-applicatives et les cartes multitâches. AWARE est une machine virtuelle JAVA CARD modifiée supportant l'exécution de plusieurs traitements simultanés. AWARE propose un ordonnancement non-préemtif distribué entre la carte et le terminal. Les applications libèrent explicitement le microprocesseur et le système embarqué interroge le terminal pour savoir quel traitement doit être réactivé.

1.3 Architectures extensibles pour le temps réel

Comme nous avons pu le voir au cours des sections précédentes, le temps réel est un domaine fécond. Beaucoup de travaux traitent des ordonnanceurs ou proposent des architectures de système temps réel. Chacun de ces travaux s'attaquent à des problématiques nouvelles ou proposent des solutions à des problèmes déjà connus. En revanche l'ordonnanceur optimal n'existe que pour une famille d'applications donnée [SSNB95] et comme le souligne STANKOVIC dans les articles [Sta96a, Sta96b] nombreux sont les problèmes non-traités en particulier concernant l'extensibilité des ordonnanceurs ou des architectures proposés. Il convient de trouver le juste équilibre entre flexibilité et prévisibilité. Le système doit être suffisamment flexible pour s'adapter à un environnement dynamique et changeant tout en garantissant le respect des échéances temporelles et en détectant les conflits liés aux différentes ressources. Les architectures temps réel classiques sont réputées pour leur fiabilité mais pas pour leur dynamicité.

Dans un premier temps, nous nous intéresserons aux motivations et problèmes soulevés par le temps réel dans les systèmes ouverts en donnant quelques clefs permettant d'y parvenir. Nous nous intéresserons ensuite aux ordonnanceurs hiérarchiques permettant de créer un environnement d'exécution dynamique par combinaison et assemblage de politiques d'ordonnancement classiques. Enfin, nous discuterons des différentes méthodes permettant de construire et d'écrire des politiques d'ordonnanceur. Ce dernier point étant une des étapes nécessaires afin de permettre le chargement dynamique d'ordonnanceurs permettant de rendre extensible les systèmes temps réel.

1.3.1 Ordonnancement temps réel pour systèmes ouverts

Un système d'exploitation ouvert se caractérise par le fait qu'il supporte le chargement dynamique de nouvelles applications. Que se passe-t-il si ces applications ont des besoins en termes d'échéance temporelle ? Comment peut-on prendre en charge l'aspect fortement dynamique de la charge du processeur qui va évoluer suivant le chargement de nouvelles applications ? Les premières solutions proposées pour prendre en compte cette dynamicité forte des systèmes ouverts a été d'utiliser des politiques d'ordonnancement dynamiques ou reconfigurables, ou encore d'utiliser des assemblages de politiques d'ordonnancement (*c.f.* l'ordonnancement mixte *RM* et *EDF* du micro-noyau EMERALDS).

Ces solutions, même si elles améliorent l'adaptabilité du système, ne le rendent pas pour autant ouvert d'un point de vue temps réel. En effet, le temps réel ouvert vise à autoriser le chargement dynamique de politique d'ordonnancement, au même titre que les systèmes ouverts autorisent le chargement dynamique d'applications. STANKOVIC, dans une étude [Sta96b] de 1996, liste les différents problèmes de recherche soulevés par le temps réel en milieu ouvert. Tout d'abord, les caractéristiques du matériel varient d'une plateforme à l'autre et ne sont donc pas connues à l'avance par le développeur de l'application ce qui rend complexe les calculs de temps d'exécution au pire cas en particulier. Ensuite, la charge du processeur et la manière dont les applications vont se partager dynamiquement le processeur ne sont pas connues à l'avance. Enfin, les techniques classiques d'ordonnancement ne sont pas ou peu applicables en l'état.

On retrouve donc des problématiques dans le temps réel pour systèmes ouverts similaires à celles rencontrées au début des systèmes ouverts. Dans un premier temps concernant les calculs de temps d'exécution au pire cas qui sont nécessaires au bon fonctionnement du système temps réel. Un système ouvert utilise souvent des langages intermédiaires ou bytecode pour charger le code ouvert et de ce fait il est plus complexe de calculer ou de vérifier un temps d'exécution au pire cas sur du code qui n'est pas dans sa forme finale ou native. Certaines machines virtuelles JAVA temps réel utilisent par exemple du *green-threading* pour partager le microprocesseur entre les différentes applications. La machine virtuelle dispose de son propre ordonnanceur et suspend les différentes tâches au bout de l'exécution d'un certain nombre de bytecodes. D'autres machines virtuelles utilisent des compilateurs «en avance de phase» [16] permettant de transformer le bytecode en code natif et ainsi d'utiliser des outils classiques de calcul de temps d'exécution au pire cas avec les mêmes restrictions concernant la detection des boucles.

Le point clef du temps réel dans un système ouvert concerne le chargement dynamique de politique d'ordonnancement. En effet une politique d'ordonnancement est souvent adaptée à une famille d'applications, un système ouvert doit donc proposer une architecture permettant l'installation de nouvelles politiques en fonctions des applications présentes à un moment donné de la vie du système. Ce chargement dynamique soulève nombre de problèmes en par-

[16] De l'anglais AOT : Ahead Of Time.

ticulier concernant la sécurité du système. Une politique d'ordonnancement est un composant proche du système et du matériel, de ce fait il convient de proposer des solutions permettant de valider qu'elle ne mettra pas le fonctionnement du système entier en péril ou ne monopolisera pas le processeur.

1.3.2 Les ordonnanceurs hiérarchiques

Les ordonnanceurs hiérarchiques partent du principe que dans un système toutes les applications n'ont pas exactement les mêmes besoins. En revanche, on peut quand même les différencier grâce à certaines caractéristiques communes et ainsi obtenir des familles d'applications. En effet, un traitement proche du matériel ou vital au fonctionnement du système comme par exemple une routine de réception des données de communication n'a pas les mêmes besoins qu'un traitement applicatif. Une fois ces différentes familles identifiées on se rend compte qu'il est plus facile de choisir la politique d'ordonnancement la plus adaptée pour une famille de traitements. Ainsi on obtient des familles de traitements ayant chacune leur ordonnanceur. En exploitant les relations qui existent entre les différentes familles on peut obtenir un classement de celles-ci par ordre de priorité. Une fois ce classement obtenu on peut le projeter sur une structure hiérarchique utilisant des politiques d'ordonnancement classiques, équitables ou proportionnelles [WW94, WW95] pour obtenir un environnement d'exécution dédié pour nos traitements. Les nœuds de l'arbre obtenus sont des politiques d'ordonnancement et les feuilles les différents traitements du système. Les ordonnanceurs hiérarchiques supportent aussi d'ordonnancer des ordonnanceurs dans le cas où cela est nécessaire par exemple lorsqu'il existe des différences de priorité au sein d'une famille. Un nœud qui reçoit un quantum de temps doit donc élire un de ces fils et lui déléguer celui-ci jusqu'à arriver à un ordonnanceur de feuille qui peut activer une tâche.

Prenons l'exemple d'un ordonnanceur hiérarchique correspondant à la politique d'ordonnancement d'un système UNIX donné figure 1.9. L'ordonnanceur racine partage le temps entre les processus utilisateurs (priorité faible) et les routines de traitement des interruptions (priorité forte). Ces routines sont séparées en deux familles. La première comporte les traitements courts qui doivent être effectués dès réception d'une interruption matérielle comme par exemple accuser reception de l'interruption et le traitement complet ou partiel de celle-ci. Dans le cas du contrôleur de disque dur ou de la carte réseau, la donnée reçue est simplement insérée dans une file d'attente qui sera traitée par la seconde sous-famille de cette branche de la hiérarchie. Dans l'autre branche de la hiérarchie principale on trouve l'ordonnanceur des processus utilisateurs.

L'approche hiérarchique peut être utilisée de manière statique ou dynamique. Dans le cas statique [DL97, ZDS97], la hiérarchie d'ordonnanceurs est construite par un expert. Elle est ensuite testée puis instanciée pour obtenir un environnement d'exécution acceptant de charger des tâches au sein des différentes familles. En revanche une fois le système déployé, il est impos-

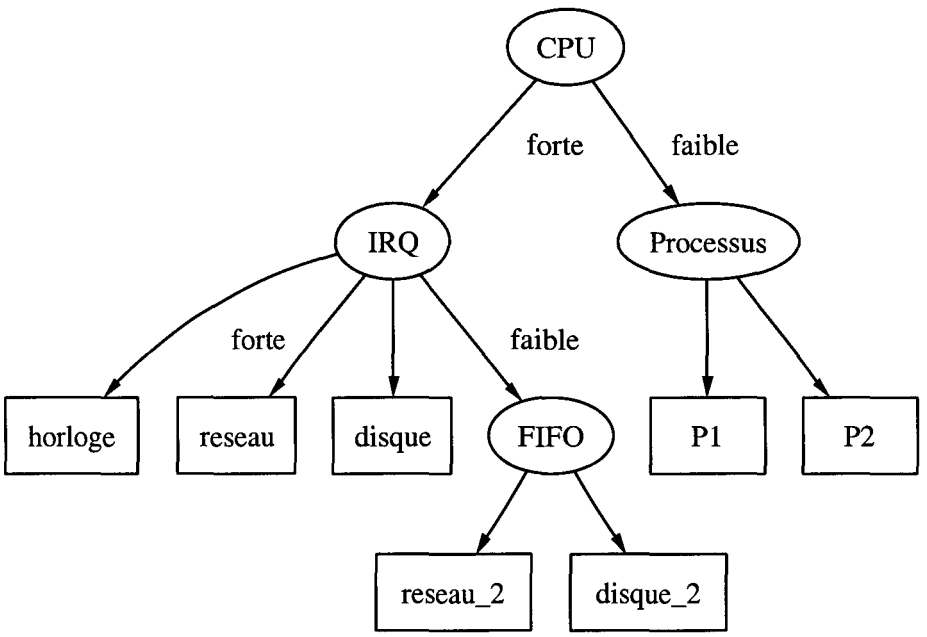


FIG. 1.9: Ordonnanceur hiérachique à la UNIX.

sible d'intervenir sur la hiérarchie et de la modifier. Cette approche est souvent utilisée dans le contexte des systèmes ouverts car elle permet d'améliorer la capacité du système obtenu à accepter de nouvelle tâche. De plus, on peut ainsi bâtir un système qui supporte l'exécution de traitements standards et temps réel de manière simultanée. Un tel système pourrait être fondé sur un ordonnanceur racine partageant le processeur avec un pourcentage fixe entre un ordonnanceur temps réel et un ordonnanceur round-robin par exemple. L'approche dynamique permet de construire la hiérarchie en ajoutant de nouvelles classes d'application et en chargeant dynamiquement de nouveaux ordonnanceurs. Le premier problème concerne les critères d'admission d'une nouvelle tâche ou d'un nouvel ordonnanceur car on doit prendre en compte les caractéristiques des ordonnanceurs des étages supérieurs ce qui est plus complexe que les tests d'admission des politiques simples.

Les ordonnanceurs hiérarchiques ont donné lieux à de nombreux travaux théoriques qui ont amené à la définition d'architecture de système. FENG et MOK attaquent ce problème suivant un axe orienté allocation de ressource en introduisant la notion de ressource virtuelle temps réel correspondant à une partition du processeur. Ils proposent un modèle hiérarchique autour de ces ressources permettant de définir des critères d'admission dans le contexte d'un système ouvert [FM02a]. Les travaux de John REGHER [Reg01] autour des ordonnanceurs hiérarchiques ont porté sur la définition d'une architecture et d'outils permettant l'assemblage dynamique d'ordonnanceurs en hiérarchie. Ces travaux ont donné lieu à la définition d'un canevas de composition de politiques d'ordonnancement *molles* nommé HLS [RS01] qui a été

évalué expérimentalement en modifiant l'ordonnanceur du système d'exploitation grand public WINDOWS 2000. Des travaux plus récents ont appliqué les concepts de HLS à l'informatique embarquée en particulier dans le cadre de la plateforme TINYOS pour les réseaux de capteurs. Ils ont amené à la définition d'un formalisme permettant de détecter les erreurs dans les environnements d'exécution ainsi créés [RRW+03].

1.3.3 Expression des politiques d'ordonnancement

L'approche la plus souvent retenue pour écrire une politique d'ordonnancement est d'utiliser un langage de programmation standard[17] en particulier celui correspondant au système qui va utiliser la politique afin de faciliter les interactions avec le système. Ce faisant le développeur de la politique doit donc être un expert du système en question ou au minimum de ses bibliothèques système. Il doit aussi connaître les spécificités du matériel et du système ainsi que de ses bibliothèques de programmation afin d'écrire au mieux sa politique d'ordonnancement. Avec ce genre d'approche, il n'y a donc aucune séparation entre l'aspect conception de politique d'ordonnancement et l'aspect programmation de système d'exploitation. Ceci contribue fortement à la grande complexité liée à la programmation de politiques d'ordonnancement dans un système temps réel.

```
...
On unblock.preemptive {
  if (e.target in blocked)
  {
      if ((!empty(running)) && (e.target > running))
      {
        running => ready;
      }
      e.target => ready;
  }
}
...
```

FIG. 1.10: Exemple de traitement de l'évènement unblock d'une politique RM.

Les travaux menés autour de BOSSA proposent de séparer la phase conception de la politique de sa phase de déploiement dans le système. Ils proposent de concevoir [CM98] un langage dédié[18] à la programmation de la politique et de fournir à son développeur des outils permettant de vérifier et valider ses propriétés et finalement de l'installer dans un système d'exploitation qui a été conçu ou modifié pour la recevoir. BOSSA peut donc être vu comme un canevas logiciel de programmation de politique d'ordonnancement. Une politique BOSSA se décompose de la sorte. Le développeur commence tout d'abord par décrire les attributs temps réels (échéance, WCET, période ...) de ses tâches. Il définit ensuite les différents états (prêt, bloqué, terminé ...) dans lesquels ses tâches peuvent se retrouver et associe à ceux-ci

[17] De l'anglais GPL : General Purpose Language.
[18] De l'anglais DSL : Domain Specific Language.

des variables représentant des singletons ou des files de tâches et permettant leur manipulation. Le programmeur doit ensuite fournir un critère permettant d'ordonner les tâches prêtes à fonctionner. Le coeur de la politique est ensuite de fournir un ensemble de réponses possibles à différents évènements remontés par le système d'exploitation. La figure 1.10 donne un exemple pour une politique de type RM. Lorsque l'on doit débloquer une tâche bloquée préemtive on regarde si une tâche est active et si elle est de priorité plus faible, dans ce cas on la suspend et dans les deux cas on active la tâche à débloquer. L'expert en système fournit au compilateur BOSSA une liste des transitions d'état autorisés et il peut ainsi vérifier que la politique respecte celles-ci [LMM03]. Le compilateur de politique vérifie aussi différentes propriétés comme par exemple que la politique ne perd pas de tâche ou ne se bloque pas. Le langage BOSSA est depuis peu modulaire [LMM04]. On peut ainsi construire une politique par composition de sous-modules de politiques. Pour ce faire on doit fournir des informations sur l'ordre d'appel des différents sous-modules dans le cas où ils sont plusieurs à demander la notification d'un évènement. BOSSA nécessite donc un système cible pour son compilateur qui pourra charger et exécuter les politiques compilées. Dans sa version actuelle, BOSSA est intégré au sein d'un noyau LINUX afin de permettre le chargement dynamique de politiques d'ordonnancement par les applications et les utilisateurs. Les modifications du noyau LINUX consistent à enlever toute trace de l'ordonnanceur initial et à les remplacer par un mécanisme de notification d'évènements à la politique BOSSA [MCM+00]. Ainsi la politique est notifiée des évènements du système et peut donc réagir à l'aide des différentes routines de traitement de ces évènement et ainsi appeler les interfaces du noyau liées à l'ordonnancement.

Finalement, Le canevas logiciel BOSSA permet donc d'exprimer [BM02] et de charger dynamiquement des politiques d'ordonnancement au sein d'un noyau modifié LINUX [LMB02]. Il fournit différentes implantations des politiques d'ordonnancement standards ainsi que des moyens de les assembler en hiérarchie. Il fournit aussi des outils permettant d'automatiser la modification du code initial du noyau LINUX. Des travaux sont en cours pour utiliser cette approche sur d'autres noyaux de système.

L'autre point important est que la politique d'ordonnancement est un des codes les plus sensibles du système d'exploitation. Une mauvaise politique va nuire au fonctionnement du système complet. Dans un système classique l'expert en système d'exploitation est souvent celui qui écrit la politique. Il peut ainsi la tester et la valider. Dans le contexte des systèmes extensibles autorisant le chargement dynamique d'une politique d'ordonnancement, on se trouve face à un problème complexe. Comment peut-on garantir le bon fonctionnement de ce composant sensible du système ? Le système doit au minimum disposer des moyens permettant de s'assurer que la politique d'ordonnancement est inoffensive. Au mieux, il devrait pouvoir tester et valider son fonctionnement avant de l'installer dans le système. BOSSA permet de valider les bonnes propriétés (pas de perte de processus, cohérence de la politique ...) de fonctionnement d'une politique au moment où elle est compilée ou conçue. Toutefois, cette seule

garantie n'est pas suffisante pour un système extensible chargeant du code mobile. Il devrait pouvoir valider la politique au moment de son déploiement dans le système embarqué. Cette validation pose un problème concernant le langage utilisé pour exprimer la politique d'ordonnancement. Peut-on prouver partie du fonctionnement d'une politique d'ordonnancement si elle est écrite en code machine ? Cette question reste ouverte et est importante pour parvenir à maitriser le comportement d'un système extensible. Elle s'applique aussi à tout composant système que l'on pourrait souhaiter charger dans un système ouvert (gestionnaire mémoire, politique d'ordonnancement, politique de protection ...).

Chapitre 2

Spécification d'un système extensible pour les applications en temps réel

«Ci-gît Celui, Dont le nom, Était écrit dans l'eau. Épitaphe de la tombe de John Keats» –
Dan Simmons, Hyperion 2, La chute d'Hyperion, 1990.

L'objectif de ce chapitre est d'identifier les besoins des applications temps réel qui ne sont pas satisfaits par les systèmes d'exploitation extensibles. À cette fin, dans la première section, nous mettons en lumière les biais introduits par les architectures extensibles dans le support des propriétés du temps réel. La deuxième section de ce chapitre liste les différents aspects que nous devons prendre en compte afin de supporter le chargement dynamique d'extensions temps réel au sein d'un exo-noyau embarqué. Finalement, nous concluons ce chapitre en donnant les stratégies que nous proposons afin de résoudre ces différents points.

2.1 Besoins temps réel pour systèmes embarqués ouverts

Les systèmes embarqués ouverts sont la solution permettant d'augmenter la capacité d'adaptation du système face aux divers changements de son milieu ou face aux interactions avec d'autres objets embarqués. Les motivations pour proposer un système embarqué ouvert supportant le chargement d'applications temps réel peuvent être abordées suivant deux axes différents. On peut tout d'abord s'interroger sur les motivations visant à introduire du temps réel dans un système embarqué ouvert. On peut aussi, à l'opposé, se demander pourquoi les concepts des systèmes temps réel classiques ne sont pas directement adaptables au domaine des objets mobiles.

Nous tenterons dans un premier temps de lister les défauts et reproches qui sont faits aux

systèmes temps réel classiques dans le cadre d'une utilisation dans un contexte de code mobile. Nous donnerons ensuite différents exemples tirés du domaine de la carte à puce qui tireraient partie de l'introduction du temps réel dans les systèmes ouverts pour carte à microprocesseur. Enfin, nous donnerons les avantages que peut apporter le support du temps réel dans un système embarqué ouvert.

2.1.1 Réactivité du système

Les systèmes temps réel ont comme préoccupation principale de garantir la fiabilité du fonctionnement des tâches qu'ils supportent. Ils proposent des solutions permettant de garantir l'accès et le partage de la ressource microprocesseur. Comme nous avons pu le voir dans la section 1.3 du premier chapitre, le principal reproche fait au temps réel concerne justement le côté fermé des environnements d'exécution obtenus [Sta96b]. Les systèmes temps réel sont en effet caractérisés par un coté statique qui permet de garantir un haut degré de prévisibilité. Les applications supportées par le système sont établies une fois pour toutes et sont embarquées en ROM avec le code du système. Pour pouvoir dimensionner correctement le système et choisir une politique d'ordonnancement, il est préférable de connaître à l'avance le nombre de tâches que le système va supporter et il est nécessaire d'avoir des informations précises concernant les caractéristiques des tâches du système.

Ces systèmes temps réel durs ont un comportement hautement prévisible et garantissent des temps de réponse quasi constants face à un évènement quelconque quelle que soit la charge et l'état du système. C'est pour ces différentes raisons que les systèmes temps réel durs sont utilisés principalement dans des domaines liés au contrôle de processus industriels automatiques ou dans des systèmes informatiques pour lesquels l'erreur n'est pas permise (avionique, automobile ...).

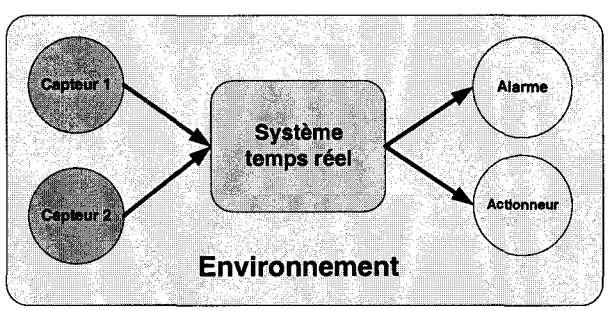


FIG. 2.1: Système temps réel industriel.

Prenons l'exemple typique de l'utilisation d'un système temps réel dans un contexte industriel illustré à l'aide de la figure 2.1. Le système scrute son environnement à l'aide de capteurs. Une ou plusieurs tâches échantillonnent l'état de ces capteurs avec une période qui correspond au nombre de mesures requises. En interne, un certain nombre de tâches sont

chargées du traitement de ces mesures et du déclenchement d'une alarme, dans le cas où un seuil critique est dépassé, ou des interactions avec le monde physique à l'aide d'actionneurs. Ainsi ce système est dimensionné pour cet environnement physique bien précis et au mieux il est reconfigurable dans le cas d'un changement de l'environnement. En revanche, pour la majeure partie de ces systèmes, lorsque le programmeur doit intervenir sur l'architecture du système pour prendre en compte une modification de l'environnement (ajout d'un capteur, modification des paramètres temporels des tâches ...), la phase de sélection d'une politique d'ordonnancement adéquate et de test d'ordonnançabilité doivent être relancées.

Concernant les politiques d'ordonnancements, il existe beaucoup de politiques temps réel dures à l'efficacité éprouvée comme, par exemple, les politiques *Rate Monotonic*, *Earliest Deadline First*, ou *Least Laxity First*. Toutefois, chacune possède ses caractéristiques qui la cantonne à un domaine d'utilisation bien précis. En revanche, peu sont optimales dans un contexte d'utilisation quelconque et en particulier dans un environnement fortement dynamique.

Ainsi, des architectures hybrides utilisant des assemblages de plusieurs politiques d'ordonnancement ont été proposées comme par exemple les ordonnanceurs hiérarchiques de REGHER [RS01] ou les travaux de DENG sur l'ordonnancement en milieu ouvert [DL97, ZDS97]. Ces politiques, ou architectures hybrides, permettent une meilleure adaptabilité du système en particulier lorsque celui-ci est amené à charger de nouvelles tâches. Les systèmes temps réel mous sont plus adaptés à une utilisation dans un environnement dynamique car ils ne cherchent qu'à faire au mieux sans véritables garanties temporelles vis à vis des tâches qu'ils hébergent.

Ainsi, même si ces architectures et politiques hybrides améliorent la capacité d'adaptation des systèmes temps réel, leur problème majeur reste leur relative fermeture dans le cas d'une utilisation en milieu dynamique. Les systèmes temps réel sont souvent conçus pour faire fonctionner une application unique ou au mieux une famille d'applications dans un environnement d'utilisation qui est maîtrisé et connu à l'avance par le concepteur du système. Ils sont peu adaptés pour faire face aux problèmes liés à l'hétérogénéité ou à l'évolution des applications, des environnements d'utilisation et des média permettant l'interaction entre le système et son environnement.

Le domaine des petits objets portables et sécurisés (POPS), dont la carte à microprocesseur est l'un des membres le plus répandu, est un de ces domaines dynamiques qui pourrait bénéficier du temps réel, en particulier pour les garanties de fiabilité. Ces POPS sont amenés à évoluer dans un environnement dynamique et changeant. Ils doivent faire face, par exemple, aux problèmes liés à la gestion d'une ressource énergétique présente en quantité limitée, mais aussi à des problématiques liées au code mobile, à la découverte et à l'interaction avec leur environnement. Le système embarqué doit donc posséder des capacités d'adaptation et de réactivité fortes aux changements de son environnement.

2.1.2 Exemple de la carte à puce

Comme nous l'avons vu précédemment (cf 1.1.3), le milieu de la carte à microprocesseur a vite accepté les architectures dites ouvertes supportant le chargement dynamique d'applications. Les systèmes d'exploitation pour les cartes à microprocesseur comptent plusieurs exemples de traitements qui bénéficieraient des concepts du temps réel. Les traitements présents dans les systèmes pour carte à microprocesseur comptent trois familles d'échéances temporelles :

- les échéances issues des protocoles ;
- les échéances applicatives ;
- les échéances imposées par le matériel.

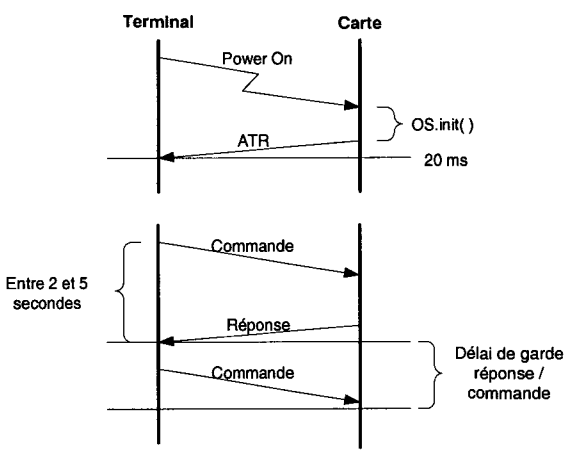


FIG. 2.2: Modéle d'exécution et de communication terminal / carte.

Ces échéances sont principalement issues du modèle d'exécution de la carte à microprocesseur qui est fortement corrélé au modèle de communication défini au sein des normes ISO7816-3&4 [ISO99]. La carte peut être vue comme un serveur répondant aux besoins du terminal dans lequel elle est insérée. La carte répond à une série de commandes normalisées du terminal et ne peut parler qu'à la suite d'une de ces commandes.

Le modèle d'interaction entre le terminal et la carte repose sur un certain nombre d'échéances temporelles permettant au terminal de contrôler l'activité de la carte.

Le premier exemple d'échéance imposée par le protocole de communication que l'on retrouve dans toutes les cartes à microprocesseur quelque soit leur domaine d'utilisation ou leur système d'exploitation est l'ATR[1]. L'ATR est la suite d'octets que doit émettre la carte après sa mise sous tension par le terminal comme illustré sur la figure 2.2. Il contient des octets d'identification comme, par exemple, la famille de la carte ou son fabriquant et aussi des octets permettant de négocier les paramètres du protocole de communication (vitesse,

[1]De l'anglais Answer To Reset.

parité, ordre des bits ...). Un temps maximum de 20ms doit séparer la mise sous tension de la carte de la réception du premier octet de l'ATR. Ce temps normalisé par l'ISO est le même pour toutes les cartes quels que soient les processeurs utilisés ou le type de carte[2]. Après émission du dernier octet de l'ATR, la norme spécifie que la carte doit être prête à fonctionner et à répondre aux besoins du terminal. Ainsi, toute carte doit initialiser son matériel (ligne série, pompe d'écriture en mémoire persistante ...), démarrer son système et être prête à fonctionner dans un court laps de temps. De plus, le redémarrage d'un système carte peut nécessiter des phases de vérification de la cohérence de la mémoire persistante car une carte peut être arrachée du terminal à tout moment par son utilisateur et se retrouver ainsi privée de courant en une fraction de seconde. Ainsi la procédure de démarrage d'un système carte est un traitement borné qui doit donc être programmé par des experts du matériel et du système d'exploitation.

Les temps de réponse à une commande du terminal doivent être compris entre 2 et 5 secondes suivant les options négociées grâce à l'ATR. Si la carte a besoin de plus de temps, elle peut envoyer au terminal une réponse de continuation spécifiant qu'elle n'a pas encore fini le traitement courant. De plus, la carte doit être prête à recevoir et traiter une nouvelle commande après un délai de garde très court, suite à l'émission de la réponse à la précédente commande. Ainsi, le protocole de communication de la carte impose à celle-ci un respect strict des échéances temporelles.

L'introduction d'architectures à base de machine virtuelle comme la JAVA CARD a permis de simplifier la prise en compte des échéances imposées par les protocoles. En effet, la machine virtuelle peut facilement suspendre l'exécution d'une application pour émettre une réponse de continuation par exemple.

L'exemple type d'échéance imposée au niveau des applications se trouve dans le module d'identification et de gestion de la communication des cartes SIM des téléphones portables. Son fonctionnement est défini par les normes ISO GSM [tGPPGb, tGPPGa]. Le module SIM est une application particulière d'une machine virtuelle JAVA CARD. Le traitement le plus important qu'il doit prendre en charge est la génération des clefs cryptographiques de session. Le module SIM doit produire une clef de session pour chaque unité de communication consommée. Ces clefs doivent être délivrées dans un temps borné et sont vitales au maintien de la communication. Si elles ne sont pas produite en temps et en heure par la carte, le téléphone portable coupe la communication. La production des clefs est donc un traitement périodique que les systèmes embarqués des carte SIM doivent prendre en charge.

La dernière famille d'échéances temporelles regroupe celles qui sont directement imposées par le matériel. Ces échéances sont présentes dans tous les systèmes d'exploitation, qu'ils soient classiques ou embarqués. Toutefois le matériel spécifique de la carte doit être pris en compte. Les normes carte [ISO99] imposent par exemple un temps maximum pendant lequel

[2]Pour un processeur carte cadencé a 3.3Mhz cela représente 66000 cycles que l'on peut comparer aux 10000 cycles nécessaires pour écrire un octet dans une page d'EEPROM.

un traitement a le droit de masquer les interruptions. En effet, pendant une phase où les interruptions sont masquées, le système carte pourrait perdre des octets de communication et serait donc dans l'incapacité de répondre à une commande du terminal. L'autre exemple type d'échéance issue du matériel est présent dans les circuits électroniques appelés pompes servant aux écritures en mémoire persistante de type EEPROM. Si les pompes sont activées pendant un laps de temps trop court, l'octet ne sera pas écrit en mémoire. À l'inverse, une trop longue durée d'activation des pompes contribuera à griller la cellule ou la page mémoire correspondant à l'octet.

Les différents exemples que nous avons décrits montrent qu'un système d'exploitation pour carte à puce respecte déjà des échéances temporelles imposées par le matériel ou par les protocoles de communication. Ce respect serait plus facile à mettre en œuvre à l'aide d'un modèle d'exécution temps réel, en particulier en présence d'outils de calcul de temps d'exécution au pire cas.

2.1.3 Sûreté de fonctionnement des systèmes ouverts

Les systèmes ouverts autorisent le chargement dynamique d'applications. Ce faisant, ils sont sujets aux problèmes classiques liés au code mobile. Le problème principal concerne la difficulté à différencier les applications véritables des applications dites malignes, qui visent à compromettre l'intégrité et la sûreté de fonctionnement du système d'exploitation et des autres applications.

Nous avons vu précédemment que beaucoup de systèmes embarqués utilisent des solutions basées sur le typage fort et sur des phases d'inférence de type, faites au niveau du langage intermédiaire dans lequel les applications sont écrites. Ces analyses de code permettent de garantir de manière statique une isolation des différentes applications. L'isolation permet de garantir l'intégrité et la confidentialité des données et des traitements des applications. Cette isolation n'est pas suffisante et elle est souvent augmentée à l'aide de contrôles dynamiques de type pare-feu ou matrice d'accès. En revanche, il est d'un autre ordre de complexité d'obtenir des garanties concernant l'utilisation les ressources matérielles par un traitement (mémoire, communication, microprocesseur).

Le multitâche est un moyen permettant d'augmenter le partage de l'accès au microprocesseur. Le temps réel permet de fiabiliser ce partage en garantissant aux tâches la disponibilité de la ressource d'exécution au moment précis où elles en ont besoin. Le temps réel permet ainsi d'améliorer la gestion de la ressource processeur en exploitant les informations sur les temps d'exécutions des différents traitements. Cela permet de valider le comportement global du système en augmentant sa «prédictabilité». Le comportement du système devient prédictible, il n'est fonction que de la politique d'ordonnancement utilisée et des attributs temporels des tâches présentes. Des tests réalisés à l'admission des tâches permettent de garantir que le système, au travers de sa politique d'ordonnancement, pourra subvenir aux besoins de ses

tâches. Ces points de contrôles sont les garants de la prédictabilité du système et du respect des garanties d'accès à la ressource d'exécution.

Le temps réel permet aussi au système de se prémunir des attaques en déni de service. En effet, même si il se fait noyer par des requêtes, ses traitements vitaux ont toujours accès au processeur grâce aux garanties fournies par la politique d'ordonnancement. De plus, au bout d'un certain temps, les requêtes ne seront même plus prises en compte car le processeur aura atteint sa charge maximale. Ainsi, le système se protège face à l'attaque en refusant d'accepter de nouveaux traitements car il ne pourra pas leur garantir un accès au processeur, mais il continue quand même à servir ses propres traitements.

Le temps réel permet ensuite de séparer les préoccupations propres au bon fonctionnement du système d'exploitation de celles propres au fonctionnement des applications. On peut ainsi isoler les éventuels défauts des traitements. Une application qui cherche à monopoliser le processeur, soit parce qu'elle cherche à attaquer le système, soit parce qu'elle a un défaut, n'interférera que sur sa propre exécution et non sur celles des autres applications. Ainsi les traitements nécessaires au bon fonctionnement du système ont des garanties immuables d'accès au microprocesseur en fonction de leurs besoins, qui sont respectés quelques soient les défauts des autres applications.

Ainsi le temps réel permet d'améliorer le partage de l'accès à la ressource d'exécution tout en conservant des garanties fortes sur cet accès. Pour ces différentes raisons, un système embarqué ouvert bénéficierait grandement du temps réel, en particulier concernant l'augmentation de sa fiabilité. Il pourrait ainsi se protéger des attaques en déni de service et garantir la disponibilité de la ressource d'exécution à ses traitements.

2.2 Les points clefs du support temps réel dans un exo-noyau

L'architecture standard des exo-noyaux, telle qu'elle a été définie par ENGLER et les chercheurs du MIT, reste fidèle à ses principes de bases concernant le démultiplexage de la ressource d'exécution. L'exo-noyau est partagé en extensions, aussi appelées systèmes, qui sont des environnements d'exécution regroupant un ensemble de traitements. L'exo-noyau est conçu de manière à permettre l'extensibilité en autorisant le chargement de nouvelles extensions tout en leurs garantissant une sécurité maximale. Concernant l'accès au microprocesseur, il ne fait que leur offrir un accès équitable [Eng98, Chapitre 3, section 3]. Nous reviendrons plus en détail dans la suite de ce chapitre sur la solution proposée initialement dans les exo-noyaux. La motivation principale de CAMILLERT est de pouvoir offrir aux extensions système et aux applications utilisateur la possibilité d'implanter des primitives temps réel mais aussi de faire cohabiter des applications standards avec des applications temps réel.

Un système temps réel dur a besoin d'un certain nombre de prérequis pour garantir l'accès au microprocesseur. Le système doit :

1. connaître les performances du matériel sur lequel il repose ;
2. maîtriser les temps d'exécution au pire cas des applications ;
3. connaître les impératifs temporels des applications ;
4. trouver un ordonnancement qui soit capable de satisfaire l'ensemble des traitements temp réel du système et de ses applications.

Ces différents prérequis au temps réel dur induisent un certain nombre de problèmes dans le cadre d'un exo-noyau ouvert comme CAMILLE.

1. le matériel est démultiplexé par l'exo-noyau, il est donc difficile de connaître précisement ses performances ;
2. le calcul des WCET des traitements des applications est rendu complexe par le fait qu'ils sont écrits à l'aide du langage intermédiaire FAÇADE et sont donc sous une forme qui ne correspond pas à celle qui sera exécutée ;
3. les tâches sont présentes au niveau des extensions, il faut donc fournir des primitives leur permettant de déclarer leurs besoins temporels au noyau ;
4. les politiques d'ordonnancement sont du domaine des extensions, il faut pouvoir valider une politique d'ordonnancement qui a été dynamiquement chargée.

Le deuxième point est spécifique à l'architecture ouverte de CAMILLE qui utilise un langage intermédiaire pour améliorer la portabilité des applications et du système. Le calcul des temps d'exécution au pire cas d'un code mobile FAÇADE représente un thème de recherche à part entière et n'est donc pas le cadre de nos travaux. Il soulève beaucoup de problèmes concernant la distribution efficace et sécurisé du calcul entre le producteur et le consommateur de code.

Les autres points concernent la problématique temps réel dans son ensemble dans un noyau ou exo-noyau quelconque. Nous aborderons ces trois problèmes clefs dans l'ordre où nous les avons listés.

2.2.1 Quantifier les temps d'exécution de l'exo-noyau

Le premier problème auquel nous sommes confrontés concerne l'ensemble des interfaces exportées par l'exo-noyau. Afin d'étendre CAMILLE vers un système d'exploitation temps réel, nous devons pouvoir quantifier les temps d'exécution des différentes applications que nous allons charger dynamiquement. Les principes clefs [Eng98, Chapitre 2] des exo-noyaux sont :

- exposer le matériel ;
- exposer l'allocation et la révocation de l'accès aux ressources ;
- protéger les ressources à faible grain ;
- exposer les noms et les informations sur le système.

Les exo-noyaux sont souvent vus comme des systèmes d'exploitation à base de bibliothèques (LibOS) [EK95]. L'exo-noyau donne une illusion d'un matériel directement accessible, les

libOS utilisent ce matériel pour en donner des abstractions et ainsi fournir aux applications des primitives permettant de l'utiliser. L'exposition des ressources matérielles par l'exo-noyau introduit deux problèmes importants concernant les temps d'exécution :

- il faut pouvoir maîtriser les performances d'une ressource qui a été virtualisée ;
- il faut pouvoir gérer les verrous matériel implicites.

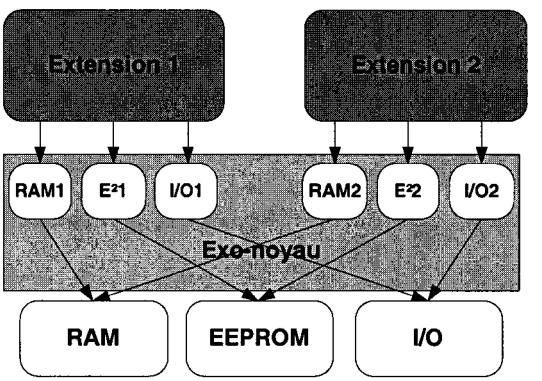


FIG. 2.3: Demultiplexage du matériel.

L'exo-noyau virtualise les ressources comme le processeur, la mémoire, l'interface réseau pour les partager entre les différentes extensions. Cette virtualisation introduit en particulier des différences entre les coûts réels des opérations supportées par le matériel et les coûts des mêmes opérations sur les ressources virtuelles. L'exo-noyau donne l'impression aux extensions qu'elles manipulent le matériel, alors qu'elles ne manipulent qu'une virtualisation de ces ressources comme illustré sur la figure 2.3. L'exo-noyau est en charge de faire les conversions nécessaires entre les ressources virtuelles et les ressources physiques mais aussi les vérifications permettant de garantir l'isolation des données et traitements des extensions. Ceci contribue donc à ajouter des surcoûts aux coûts réels d'accès et d'utilisation du matériel.

L'exemple typique de cette différence concerne le démultiplexage des interruptions matérielles. Les extensions ne manipulent pas directement le matériel mais les ressources virtuelles proposées par l'exo-noyau. Ainsi, elles ne reçoivent pas directement les interruptions matérielles. C'est l'exo-noyau qui les reçoit et qui est en charge de les remonter aux extensions en utilisant un mécanisme de notification. Le coût de réception et de traitement de l'interruption après démultiplexage vers les extensions est donc supérieur au coût physique de réception et de traitement de l'interruption par l'exo-noyau. En effet, en raison du partage du microprocesseur entre les extensions, la notification de réception d'une interruption arrive au mieux immédiatement après que l'extension qui a l'accès au microprocesseur rende la main. Les interruptions sont souvent problématiques dans les systèmes temps réel classiques. La manière dont elles sont démultiplexées par l'exo-noyau introduit des problèmes supplémentaires concernant le retard des notifications. Les retards des notifications des interruptions sont donc difficiles à prédire car ils sont dépendants de critères dynamiques (nombre d'extensions,

ordre d'exécution des différentes extensions, temps d'exécution des fonctions de réception des interruptions démultiplexées, ...).

L'autre problème qui rend complexe les calculs des temps d'exécution au pire cas pour l'exo-noyau concerne la prise en compte des verrous matériels implicites. L'exo-noyau exposant le matériel tel qu'il est, les extensions, ou libOS, sont donc sujets aux désagréments qu'il engendre. Prenons l'exemple d'une mémoire persistante virtualisée sous la forme d'un ensemble de pages qui supporte la lecture et l'écriture de données de différentes tailles. Comme nous l'avons vu précédemment 1.1.2, les écritures en mémoire persistante entraînent un gel du microprocesseur (ou au mieux du banc mémoire correspondant à l'octet écrit) tant que l'écriture n'est pas entièrement finie. Un autre exemple de verrou matériel peut être présent dans un circuit de communication de type UART. Suivant l'UART, il est possible que l'émission d'un octet soit bloquante. Il est aussi relativement courant que les interfaces de communication regroupent les émissions de bits de données. Ainsi, le coût d'émission n'est pas le même pour tous les bits de données. Un coût supplémentaire est ajouté tous les n bits lors de l'émission d'un paquet.

Ces verrous matériels doivent être pris en compte dans les calculs des temps d'exécution au pire cas au sein de l'exo-noyau. Dans le cas contraire, une application pourrait les exploiter pour monopoliser la ressource d'exécution et ralentir les autres traitements, comme par exemple, en lançant un nombre important d'écritures en mémoire persistante.

2.2.2 Trouver un ordonnancement satisfaisant l'ensemble des tâches

Connaissant le temps d'exécution requis pour chaque opération du matériel exposée par l'exo-noyau et aussi pour chaque application chargée, le problème est maintenant de trouver un ordonnancement faisable pour un ensemble de tâches standards et ayant des besoins temps réel. AEGIS & XOK, les deux prototypes d'exo-noyau développés par le MIT gèrent le partage de l'accès au microprocesseur de la même façon. Le microprocesseur est découpé en unités élémentaires appelées quantum de temps. Le noyau fournit des primitives permettant la réservation et la révocation des quanta de temps. Les extensions peuvent ainsi réserver un quantum, un groupe de quanta consécutifs, ou une série de quanta avec une périodicité donnée en fonction des besoins de leurs applications. Ainsi, le microprocesseur est vu comme un vecteur de quanta réservés par les différentes extensions. Pour chaque quantum de temps du microprocesseur, le noyau utilise une politique équitable de type «*round-robin*» pour élire une extension [Eng98, Chapitre 3, Section 3]. Cette extension peut alors utiliser le quantum pour exécuter un de ses traitements. Ce traitement peut, s'il n'a pas utilisé la totalité du quantum, offrir le temps restant à une autre application de la même extension. Pour ce faire, le noyau fournit aux extensions une primitive `yield(int pid)` permettant de designer explicitement le traitement qui va recevoir le reste du quantum. Un appel à `yield(int pid)` entraîne donc un changement de contexte pour (ré)activer le nouveau traitement. L'exo-noyau gère ainsi

l'accès à la ressource d'exécution (allocation, révocation) et fournit des primitives permettant aux extensions de manipuler les contextes d'exécution des traitements. Les extensions peuvent exploiter ces différentes primitives afin de supporter à leur niveau des politiques permettant d'ordonnancer leurs traitements.

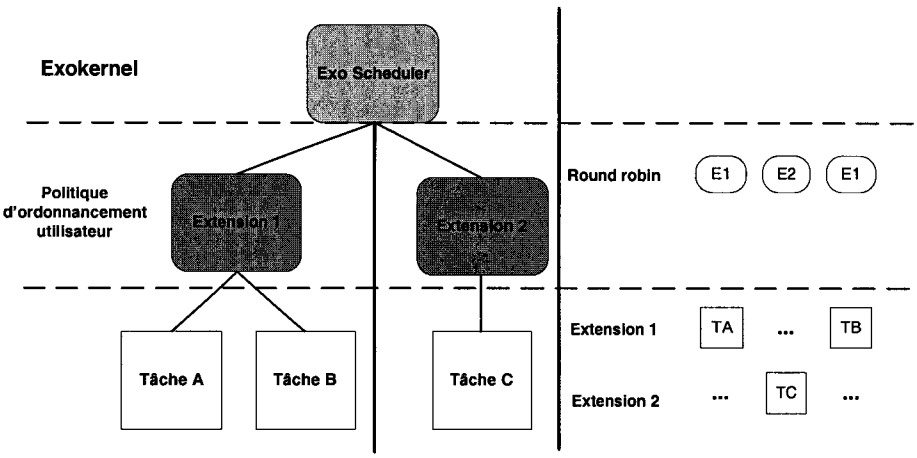


FIG. 2.4: Démultiplexage du microprocesseur.

L'exo-noyau virtualise la ressource d'exécution en utilisant une politique simple et impartiale de type *round-robin* et chaque extension a ainsi l'impression d'être un système à part entière, isolé des autres extensions. Les quanta de temps sont partagés entre les extensions du système qui choisissent alors une application à qui les donner en fonction de leur propre politique d'ordonnancement. Illustrons le fonctionnement du démultiplexage du microprocesseur l'exemple donné figure 2.4. Considérons un exo-noyau partageant équitablement le processeur entre deux extensions. La première extension supporte deux tâches nommées A et B et la deuxième supporte une tâche unique C. L'ordonnanceur de l'exo-noyau donne alternativement la main à chacune des extensions qui peuvent ainsi choisir parmi leurs propres tâches celle qui va utiliser le quantum de temps. Ce processus de partage de CPU à deux niveaux est proche des ordonnanceurs hiérarchiques [Reg01] ou des politiques d'ordonnancement en milieu ouvert de DENG et LIU [DL97, ZDS97] mais introduit des solutions sous optimales.

Reconsidérons l'exemple précédent mais cette fois avec des tâches ayant des besoins temps réels. La tâche A nécessite un quantum tous les deux quanta ce qui correspond à un taux de $\frac{1}{2}$; B a un taux de $\frac{1}{5}$ et C un taux de $\frac{1}{4}$. Le système obtenu a donc une hyperpériode de vingt quanta (plus petit commun multiple des périodes des tâches). Supposons que la première extension utilise une politique d'ordonnancement de type *EDF*.

L'ordonnanceur de l'exo-noyau donne alternativement la main aux deux extensions. On obtient donc la trace d'exécution suivante concernant les activations des deux extensions : `E1-E2-E1-E2-....`

L'ordonnanceur de la première extension doit choisir une des deux tâches A et B lorsqu'il

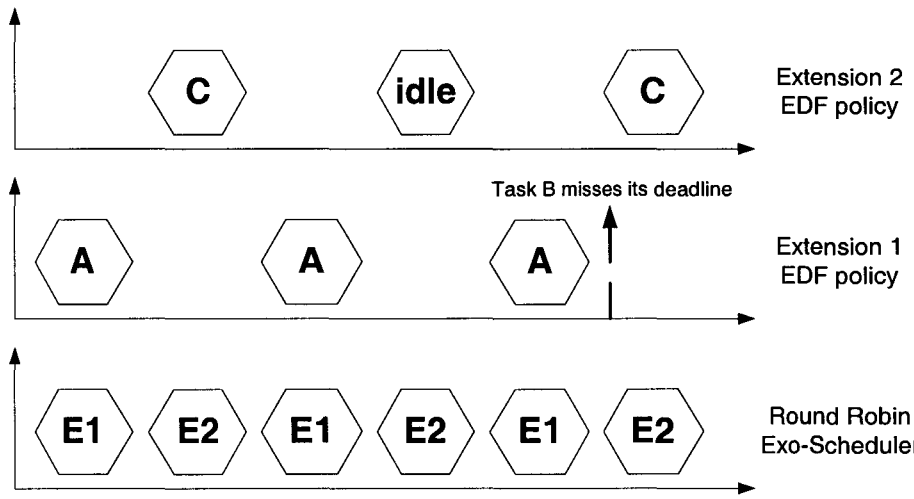


FIG. 2.5: Ordonnancement aveugle.

reçoit un quantum du microprocesseur. C'est une politique *EDF* qui choisit donc la tâche qui possède la plus petite échéance temporelle pour chaque quantum de temps. A ayant la plus petite échéance, elle répartit les quanta de temps de la manière suivante : A-A-A-.... Cette extension est donc dans l'incapacité de respecter les échéances temporelles de la tâche B.

L'ordonnanceur de la deuxième extension n'ayant qu'une seule tâche donne un quantum sur deux à la tâche C pour respecter ses besoins. Les autres quanta sont attribués à la pseudo tâche *idle* car non utilisés.

Les traces complètes de l'exécution des différents ordonnanceurs sont résumées sur la figure 2.5. Cet exemple illustre le problème de l'ordonnancement aveugle. La première extension est surchargée, avec un quantum tous les deux, elle ne peut pas satisfaire ses deux tâches. En revanche, le quatrième quantum n'a pas été consommé par la deuxième extension. Si l'on considère les mêmes tâches regroupées sous une même extension, une simple politique de type *Rate Monotonic* est capable de les ordonnancer et produit comme plan d'ordonnancement ACAB-ACAB-ACAB-ACAB-ACA-LIBRE.

Ce modèle n'est pas optimal pour des taches temps réel car chaque extension essaie d'ordonnancer ses tâches dans ses propres quanta sans tenir compte des autres extensions. L'isolation entre les extensions est nuisible car elle peut amener le système à rejeter des extensions dont les tâches pourraient être ordonnancées avec une collaboration des extensions. Nous présentons par la suite notre architecture d'ordonnancement collaboratif qui résout le problème de l'ordonnancement aveugle.

2.2.3 Fiabiliser l'exo-ordonnancement

Nous avons illustré précédemment le problème majeur lié à la manière dont l'exo-noyau partage le microprocesseur entre les extensions. Les extensions ont la capacité d'ordonnancer

leurs propres tâches grâce à l'exo-noyau. En revanche, l'isolation des extensions entraîne à refuser des tâches qu'un système non isolé pourrait ordonnancer. Ce problème est appelé problème de l'ordonnancement aveugle. L'isolation permet de fournir aux extensions des garanties concernant l'accès au microprocesseur, mais aussi concernant la non-interférence entre extensions. En revanche, elle limite les interactions possibles entre les extensions. Dans l'exemple précédent, si la deuxième extension avait eut connaissance des tâches de la première, elle aurait pu céder son droit d'accès au quatrième quantum au profit de la tâche B.

En introduisant de la collaboration entre extensions sous la forme d'une mise en commun et d'un partage des quanta du microprocesseur, on pourrait satisfaire plus d'ensembles de tâches. Toutefois, il faut pouvoir garantir la fiabilité de cette collaboration. Pour revenir à l'exemple, s'il n'est pas possible d'assurer à la deuxième extension que la première va lui céder un de ses quanta au profit de la tâche B à chaque hyperpériode, il devient impossible de garantir le respect des échéances temporelles de B dans l'absolu. Or, ce respect des échéances temporelles des tâches est à la base du temps réel dur. Une solution est de se passer de la politique d'ordonnancement d'une des deux extensions et de rattacher les trois tâches sous la politique restante. Sous réserve que la politique soit capable de satisfaire les tâches, on règle le problème de l'ordonnancement aveugle mais on court-circuite le schéma de confiance verticale. En effet, une extension a confiance envers l'exo-noyau et les tâches rattachées à cette extension lui font confiance et par transitivité font confiance à l'exo-noyau.

Cette proposition introduit donc trois nouveaux problèmes importants. Comme les extensions n'ont connaissance que de leur propres tâches, il faut fournir un moyen de rendre publiques les caractéristiques temps réel d'une tâche à une autre extension. Pour pouvoir partager l'accès au microprocesseur et donc collaborer, il faut alors dépasser l'isolation entre extensions. Les tâches doivent ainsi notifier à l'exo-noyau leurs caractéristiques temps réel. Elles doivent, en particulier, publier les informations concernant les besoins temps réel (période, échéance, temps d'exécution au pire cas). Connaissant les caractéristiques des tâches, l'ordonnanceur choisi pourra donc gérer les tâches de sa propre extension, mais aussi celles des autres extensions qui se sont rattachées à lui.

Le deuxième problème se rattache à la confiance qu'a une extension envers les autres extensions. Si une extension décide de déléguer l'ordonnancement de ses tâches à une autre extension, elle doit avoir la garantie que l'extension en question dispose d'une politique d'ordonnancement en laquelle elle peut avoir confiance. Cette politique doit être capable d'ordonnancer les tâches de sa propre extension et les tâches rattachées. Ainsi, les politiques d'ordonnancement, au même titre que n'importe quelle partie d'une application, sont du code mobile qui même s'il est proche du système n'est pas de confiance. On doit proposer des solutions permettant de garantir certaines propriétés concernant le bon fonctionnement d'une politique d'ordonnancement. Le fait que les extensions notifient les caractéristiques de leurs tâches à l'exo-noyau permet à celui-ci de vérifier que la politique choisie satisfait bien toutes

les échéances des tâches.

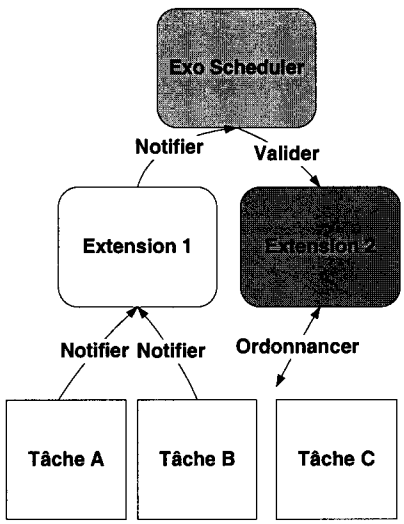


FIG. 2.6: Collaboration et confiance.

La figure 2.6 illustre ces deux problèmes. Nous avons vu précédemment que dans certain cas, les échéances de toutes les tâches n'étaient pas respectées. Si l'on décide de se passer de l'ordonnanceur de la première extension, celle-ci doit dans un premier temps notifier les caractéristiques de ses deux tâches A et B à l'exo-noyau. L'exo-noyau doit s'assurer que la politique d'ordonnancement de la deuxième extension est capable de satisfaire les échéances des tâches A, B et C. Si tel est le cas, la première extension aura la garantie de la bonne exécution de ses tâches et la deuxième extension sera en charge d'ordonnancer les trois tâches.

Le dernier problème concerne la gestion des contextes d'exécution des tâches. Au minimum, l'exo-noyau doit fournir un moyen de sauvegarder et de restaurer l'état du microprocesseur. Toutefois, les extensions doivent pouvoir intervenir dans ce processus car elles peuvent enrichir les contextes d'exécutions à l'aide d'autres informations comme par exemple l'état de la tâche (prête, bloquée, terminée ...). Le mécanisme de partage doit prendre en compte les notions de contextes enrichis. En effet, dans le cas d'un partage d'une politique d'ordonnancement et de la mutualisation des quanta de temps, ce n'est pas nécessairement l'extension à laquelle appartient la tâche qui va gérer les activations et les gels de celle-ci.

2.3 Stratégies pour support temps réel dans un exo-noyau

Nous avons listé dans la section précédente les différents problèmes auxquels nous sommes confrontés afin de supporter du temps réel au sein de l'exo-noyau CAMILLE. Nous allons donner les stratégies que nous comptons mettre en œuvre pour résoudre ces problèmes.

Nous nous intéresserons, dans un premier temps, à la maîtrise des temps d'exécution des primitives de l'exo-noyau. Nous expliquerons, par la suite, comment nous comptons supporter

du temps réel au niveau des extensions de l'exo-noyau. Enfin, nous conclurons ce chapitre en discutant des problèmes liés à l'isolation entre les extensions en particulier dans le cas où elles désirent collaborer en partageant un service.

2.3.1 Maîtrise des temps d'exécution du noyau

Nous avons vu précédemment que pour calculer les temps d'exécution des codes mobiles FAÇADE, il est nécessaire de pouvoir borner les temps d'exécution des primitives du noyau. Le but premier de l'exo-noyau étant de fournir des abstractions du matériel et des primitives permettant aux extensions de le manipuler et de l'utiliser.

Nous avons vu que le problème majeur concernant le bornage des temps d'exécution est introduit par la virtualisation du matériel proposée par l'exo-noyau. CAMILLE étant un exo-noyau pour petits objets portables et sécurisés, il reste proche du matériel au niveau du noyau. Cette proximité du matériel permet d'avoir une empreinte mémoire la plus faible possible pour le code du noyau et de garantir des accès efficaces au matériel. L'utilisateur peut bâtir à sa guise ses propres abstractions au sein de ses extensions. Ainsi, les abstractions de plus haut niveau seront bâties par les extensions et seront donc vues comme du code mobile utilisant les abstractions du matériel fournies par l'exo-noyau.

Considérons l'exemple des abstractions de la mémoire. CAMILLE fournit une vue de la mémoire sous la forme d'un ensemble de pages de taille fixe. Les opérations de manipulation de ces pages comme la lecture ou l'écriture sont regroupées au sein de différents composants. Par soucis d'efficacité, ces primitives sont directement projetées par le compilateur embarqué sur les accès élémentaires à la mémoire du microprocesseur. CAMILLE fait la séparation entre la mémoire persistante et volatile. La persistance est une propriété statique : un objet transiant dans CAMILLE le restera pendant toute la durée de son cycle de vie. Il n'aura jamais l'occasion de se rendre persistant comme cela est possible en JAVA par exemple[3]. Cette séparation franche des différents types de mémoire au sein de composants permet de faciliter les garanties de cohérence de la mémoire. Ainsi, les abstractions du matériel proposées dans CAMILLE, de part leur proximité au matériel, possèdent les bonnes propriétés permettant de borner les temps d'exécution des traitements qu'elles supportent.

Toutefois, le bornage des temps d'exécution reste un calcul impossible à réaliser sur du code quelconque (cela revient à prouver la terminaison d'une fonction quelconque).

Les traitements couramment réalisés par CAMILLE n'ont pas tous des besoins temps réel ou plus simplement ne sont pas bornables. Parmi ces traitements, on peut en particulier isoler le processus de compilation embarqué permettant de produire du code natif à partir du code intermédiaire FAÇADE. La compilation d'une méthode FAÇADE est un algorithme complexe qui est difficilement bornable. En effet, le processus de compilation de CAMILLE traite le code FAÇADE instruction par instruction pour limiter la quantité de mémoire de

[3]La persistance en JAVA est une propriété dynamique.

travail nécessaire pendant la phase de compilation. Ainsi, on ne connait pas la totalité du code FAÇADE à l'avance, mais seulement quelques informations pertinentes pour initialiser la phase de compilation (nombre d'instructions, nombre de variables, preuve de typage ...). Il est donc difficile d'estimer le temps nécessaire pour finaliser la compilation d'un code FAÇADE en code natif.

Nous devons donc catégoriser les primitives du noyau en deux familles :

- la première famille regroupe les abstractions et primitives que des extensions temps réel peuvent appeler ;
- la deuxième famille regroupe les traitements non bornés ou non bornables du système.

Nous devons donc respecter quelques règles de bonne écriture de code pour les traitements de la première famille afin de produire du code dont les temps d'exécution sont bornables (pas de boucle infinie, maîtrise des appels récursifs, des cycles d'appel ...).

Le deuxième point important concerne les moyens et méthodes à utiliser pour borner les temps d'exécution des primitives du noyau pouvant être appelées par des extensions ayant des besoins temps réel. Le code de notre noyau est écrit dans un sous-ensemble fortement typé du C (pas de `void *`, restriction sur les conversions de type, pas d'utilisation des fonctions de la librairie standard C). Le code de CAMILLE peut donc être soit compilé à l'aide d'un compilateur C standard ou alors avec notre GCC modifié, qui produit du code FAÇADE à partir du C fortement typé. Ainsi nous pouvons utiliser deux méthodes différentes pour obtenir les temps d'exécution au pire cas des primitives du noyau.

La première solution est d'analyser le code natif produit par un compilateur C classique. Nous pouvons utiliser des outils de calcul des WCET comme par exemple HEPTANE, comme cela a été fait sur le code du noyau de système temps réel RTEMS [CP01b]. La deuxième solution est d'utiliser notre compilateur GCC modifié pour compiler le code du noyau en FAÇADE, puis d'effectuer l'analyse des temps d'exécution au pire cas sur ce code comme si c'était du code mobile classique comme expliqué précédemment (cf ??).

2.3.2 Supporter des extensions «temps réel»

L'architecture de CAMILLE à été initialement pensée pour répondre aux problèmes d'extensibilité système. L'objectif principal derrière CAMILLERT est de proposer une révision de cette architecture permettant à CAMILLE de charger dynamiquement des extensions temps réel.

Une première approche est de proposer et de supporter une politique temps réel extensible unique au niveau de l'exo-noyau. Chaque extension peut alors déclarer et enregistrer ses tâches auprès du système pendant la phase d'installation. La politique d'ordonnancement de l'exo-noyau est alors en charge d'ordonnancer les tâches des extensions ainsi que les différents traitements nécessaires au bon fonctionnement du système. En utilisant une politique d'ordonnancement dynamique résistante au chargement de tâche, on peut ainsi fournir une

certaine forme d'extensibilité au niveau du système.

Les extensions de l'exo-noyau servant à virtualiser et démultiplexer l'accès au matériel, elles forment des systèmes virtuels regroupant un ensemble de traitements partageant des caractéristiques communes. Comme l'a illustré STANKOVIC dans [SSNB95], chaque famille d'ordonnanceur à ses limites et il n'existe aucun ordonnanceur optimal pour une large famille de tâches appartenant à différents domaines d'application. L'extensibilité que fournirait une telle architecture serait, de ce fait, forcément limitée à la capacité d'extensibilité de la politique d'ordonnancement choisie. Cela reviendrait à considérer l'exo-noyau comme étant un système monolithique temps réel supportant une unique extension virtuelle qui regrouperait l'ensemble des tâches.

Nous voulons proposer une architecture extensible supportant des applications en temps réel. Plus précisément, nous voulons placer l'intelligence des politiques d'ordonnancement au niveau des extensions et non pas au niveau de l'exo-noyau. Nous voulons que chaque extension puisse charger sa propre politique d'ordonnancement. En effet, une extension est la seule à connaître précisément les caractéristiques de ses tâches et est donc la plus à même à choisir la politique d'ordonnancement la plus adaptée à celles-ci. Nous ne cherchons donc pas à proposer une nouvelle politique d'ordonnancement extensible, mais plutôt à fournir des solutions au niveau de l'exo-noyau permettant à une extension d'amener et d'utiliser la politique d'ordonnancement la plus adaptée aux traitements qu'elle réalise et à gérer ceux-ci de manière adéquate.

L'exo-noyau doit donc fournir différents services aux extensions :

- virtualiser la ressource d'exécution ;
- fournir un moyen de négocier l'accès à cette ressource (allocation, révocation) ;
- garantir l'accès à la ressource d'exécution et gérer des quotas ;
- gérer un premier niveau de contextes d'exécution ;
- fournir des primitives de gestions de ces contextes (initialisation, (ré)activation, gel).

Son premier rôle est donc la virtualisation et le démultiplexage du microprocesseur afin que chaque extension puisse avoir accès à la ressource d'exécution en fonction de ses besoins. À cette fin, nous nous proposons de :

1. partager le processeur en quanta de temps élémentaires ;
2. fournir des mécanismes permettant à l'exo-noyau de notifier les débuts et les fins de quantum à une extension ;
3. hiérarchiser les extensions suivant leurs besoins et le type de tâche qu'elles supportent ;
4. garantir les accès et la taille des quanta de temps à l'aide d'un mécanisme à base de vote.

Une extension doit pouvoir négocier l'accès au microprocesseur à l'exo-noyau en ayant la possibilité de réserver ou de libérer des quanta de temps. Nous allons exploiter un algorithme à base de vote permettant à une extension de demander un quantum de temps ou de changer

la taille des quanta sans nuire au bon fonctionnement des autres extensions. L'exo-noyau doit impérativement fournir aux extensions des garanties concernant l'accès à ces quanta de temps. En effet, pour qu'une extension puisse respecter les échéances temporelles de ses tâches, elle doit au minimum avoir la garantie d'avoir accès au microprocesseur pendant les quanta qu'elle a négocié. Notre mécanisme de vote va permettre de garantir ces quanta aux extensions.

De plus, comme nous l'avons dit précédemment, tous les traitements supportés par CAMILLE n'ont pas des besoins temps réel. Beaucoup de traitement sont des traitements standard comme par exemple le processus de compilation embarqué. Ainsi, l'architecture de démultiplexage du microprocesseur que nous proposons doit donc permettre de supporter la coexistence de traitements standards et de tâches temps réel. Le système doit garantir qu'un traitement standard ne va pas mettre en péril les échéances temporelles d'une tâche temps réel.

2.3.3 Au delà de l'isolation des extensions

Nous avons vu précédemment que l'architecture des exo-noyaux, telle qu'elle a été initialement proposée par le MIT, a tendance à isoler les extensions en leur donnant l'impression de fonctionner sur un système à part entière. Cette isolation entre extensions, même si elle a des avantages, possède aussi des inconvénients.

L'avantage majeur de l'isolation est qu'elle favorise la sécurité des extensions en les cloisonnant dans des espaces logiciel distincts. Nous avons illustré précédemment, à l'aide d'un exemple lié à l'ordonnancement de trois tâches temps réel, que l'isolation entre extensions contribue parfois à refuser des traitements qu'un système non isolé pourrait supporter. Dans cet exemple, l'isolation couplée à une répartition non équilibrée des tâches entre les deux extensions illustre le fait que l'isolation ne favorise pas l'utilisation optimale de la ressource d'exécution. L'isolation devient nuisible dans de tels cas et il faut mettre en balance ce problème avec le bénéfice qu'elle apporte concernant la sécurité.

L'isolation introduit aussi une certaine forme de redondance des traitements et des structures de données. Deux extensions utilisant exactement le même service vont devoir l'implémenter dans leur propre espace. Cette duplication du code et des structures de données n'est pas un problème dans l'informatique traditionnelle. En revanche, les systèmes embarqués ne disposent souvent que de peu de mémoire de code ou de travail. La redondance est donc une chose que l'on ne peut pas accepter dans le domaine du logiciel enfoui.

À l'opposé de l'isolation, le partage d'une ressource favorise une meilleure utilisation des différentes ressources du système. Le partage de service entre extensions permet en particulier d'éviter la duplication de code ou de données. En revanche, dans la mesure où une extension décide de partager un traitement ou une donnée, elle s'expose à d'éventuelles utilisations malignes de la part d'une autre extension. De même, si une extension utilise un service d'une autre extension, comment peut-elle avoir la garantie du bon fonctionnement de celui-ci. De

manière générale, le partage d'un service sous-entend souvent une certaine forme de confiance mutuelle.

Nous nous retrouvons face à un problème nécessitant un compromis : l'isolation favorise la sécurité, le partage optimise l'utilisation des ressources. Nous souhaitons proposer une architecture non-isolée exploitant un partage entre extensions mais qui continue de garantir la sécurité des extensions.

Nous supportons deux formes de partage dans notre architecture :

- une extension peut tout d'abord utiliser l'implantation d'un service d'un autre extension car elle ne posséde pas sa propre implantation de ce service ;
- plusieurs extensions peuvent mutualiser leurs accès à une ressource dans le but de mieux l'utiliser ensemble que séparement.

La première forme permet de pallier les problèmes de duplication des données et des traitements entre les extensions. La deuxième permet de lutter contre les problèmes de mauvaise utilisation d'une ressource issue de l'isolation comme, par exemple, le problème de l'ordonnancement aveugle qui a été décrit précédement.

Notre architecture de partage de service va rétablir la confiance entre les extensions. Elle va exploiter un algorithme en plusieurs phases permettant à une extension de tester le bon fonctionnement d'un service d'une autre extension en présence de ses données puis, par la suite, de l'utiliser en ayant des garanties concernant la gestion des données.

Le prochain chapitre sera consacré à la description de notre canevas pour partage sûr de service entre extensions. Il donnera des exemples d'utilisation possibles pour valider des politiques d'ordonnancement ou des politiques de protection de données type matrice d'accès.

Nous décrirons ensuite l'architecture que nous avons mise en place dans CAMILLERT permettant de supporter du temps réel dans un exo-noyau au niveau des extensions. Cette architecture exploite notre canevas de partage sûr de service dans le cadre d'extensions désirant partager une politique d'ordonnancement ou mutualiser leurs accès au microprocesseur. Cette architecture permet notamment de résoudre le problème de l'ordonnancement aveugle que nous avons énoncé précédement.

Chapitre 3

Canevas pour le partage sûr de services dans un noyau

«De chacun selon ses capacités à chacun selon ses besoins» – **Karl Marx.**

Nous allons, dans ce chapitre, présenter notre algorithme permettant de valider le respect d'un service système vis-à-vis de ses spécifications. Cet algorithme s'utilise dans un contexte où une extension «producteur» délivre un service exploité par une extension «consommateur». Cette dernière va tester le service dans un contexte correspondant à son futur cadre d'utilisation , puis s'il remplit les besoins attendus, va l'utiliser. Pour que ce test de conformité soit possible, nous devons pouvoir garantir que le service est déterministe, c'est-à-dire qu'il fournit le même résultat d'une exécution à l'autre en présence des mêmes entrées. Notre algorithme est utilisable pour tous les services systèmes possédant un paramètre critique appartenant à un domaine borné. Il repose sur l'idée qu'il est plus facile de valider le fonctionnement d'un service chargé dynamiquement vis à vis de l'état courant du système au moment de son déploiement que de le valider dans un cadre général.

Dans un premier temps, nous discuterons des limites que l'isolation des ressources faites par l'exo-noyau impose au partage des ressources. Nous discuterons de la nécessité, dans un système supportant le chargement dynamique de services systèmes, de valider leurs comportements afin de sécuriser leur partage entre extensions. Enfin, nous présenterons notre algorithme distribué permettant de valider le bon fonctionnement d'un service système.

3.1 Extensibilité et factorisation

L'architecture des exo-noyaux à été proposée pour augmenter l'extensibilité des systèmes d'exploitation. Nous discuterons ici des caractéristiques de l'architecture des exo-noyaux. Dans un premier temps, nous nous intéresserons à l'isolation entre extensions qui permet de garantir

la sécurité. Nous verrons ensuite que cette isolation introduit une utilisation non-optimale des ressources et peut conduire les extensions à dupliquer un même service plutôt que de prendre le risque de réutiliser un composant étranger en lequel elles n'ont aucune confiance. Nous conclurons cette section en traitant du problème de confiance entre extensions qui est un véritable verrou au partage de services dans notre contexte.

3.1.1 Isolation des extensions

L'architecture des exo-noyaux a été proposée pour palier aux problèmes de fermeture des systèmes monolithiques. Pour ce faire, ENGLER a proposé de redéfinir les services que doit proposer le système. Un système d'exploitation propose des abstractions des différentes ressources matérielles. Il est aussi en charge de protéger l'accès et l'utilisation de ces ressources. Pour ce faire, les architectures de systèmes exploitent de plus en plus des abstractions de haut niveau du matériel. Ces abstractions de haut niveau ont pour motivation première de faciliter son utilisation et aussi sa protection. Toutefois, ce faisant, les applications sont contraintes à utiliser le matériel d'une manière prédéfinie, imposée par la façon dont le système l'a abstrait.

Les exo-noyaux exploitent une approche radicalement opposée permettant d'avoir un système extensible. Ils proposent une architecture permettant à plusieurs abstractions systèmes, appelées extensions, d'utiliser le même matériel de la manière la plus adaptée à leurs propres objectifs. Ainsi, les politiques de cache d'une base de données et d'un système de fichier exploitant le même disque matériel peuvent coexister avec un minimum d'interférences. Un exo-noyau est donc un système proposant la vue la plus proche possible du matériel et supportant la coexistence (pacifique) de plusieurs extensions. Un système à base d'exo-noyau est un système que l'utilisateur doit construire [RDG04]. Il limite à exposer et protéger l'accès au matériel, le concepteur du système devant alors écrire les extensions qui correspondent à ses besoins. Les fonctionnalités du système sont donc grandement déportées vers les extensions. La figure 3.1 donne un exemple d'un exo-noyau pour système embarqué supportant deux extensions. Ces deux extensions exploitent les ressources matérielles qui sont virtualisées par l'exo-noyau.

L'extensibilité de l'exo-noyau est donc obtenue en déportant les abstractions et la gestion du matériel dans les extensions. Le noyau expose et protège le matériel et les extensions l'utilisent à leur guise. L'utilisateur peut, en chargeant dynamiquement une nouvelle extension, construire l'environnement d'exécution le plus adapté à ses applications.

Cette manière d'exposer le matériel est de le partager entre les différentes extensions permet à l'exo-noyau de donner l'impression aux extensions qu'elles fonctionnent seules sur un système à part entière. Les extensions peuvent être vues comme des systèmes virtuels. L'isolation est donc une propriété de base de l'exo-noyau qui permet la coexistance des extensions.

Une relation de confiance implicite existe entre une extension et l'exo-noyau. De la même façon, les applications supportées par une extension font confiance à l'extension qui les sup-

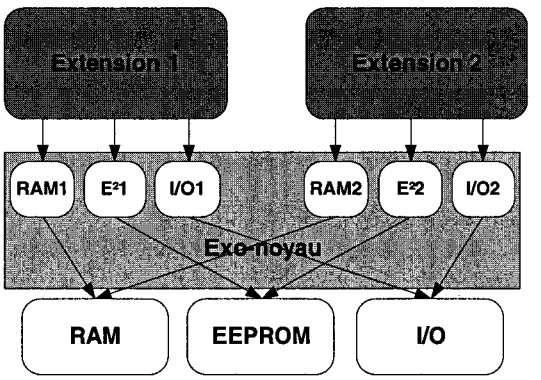


FIG. 3.1: Demultiplexage du matériel.

porte. Les extensions font confiance à l'exo-noyau, l'exo-noyau les surveille pour isoler leurs éventuels défauts et ainsi garantir sa propre stabilité autant que celles des autres extensions. En revanche, une extension ne fait pas directement confiance à une autre extension. La confiance dans les exo-noyaux respecte donc un schéma vertical.

Les deux prototypes d'exo-noyaux (AEGIS et XOK) proposés par ENGLER [Eng98] respectent ce schéma de confiance en particulier concernant le démultiplexage de l'accès au disque dur. Les extensions partagent toutes le même disque dur qui est abstrait sous la forme de secteurs par l'exo-noyau. L'exo-noyau fournit des primitives permettant l'allocation des secteurs. Pour garantir l'isolation des secteurs du disque, chaque extension proposant un système de fichier doit fournir une fonction particulière appelée `own()`. Cette fonction permet à l'exo-noyau d'interroger une extension pour savoir quels secteurs du disque dur lui appartiennent. L'exo-noyau impose que ces fonctions aient des propriétés particulières sur lesquelles nous reviendrons par la suite. En contrôlant ainsi les extensions, l'exo-noyau garanti qu'une extension n'ira pas lire ou écrire dans un secteur du disque qui ne lui appartient pas. Ainsi, les extensions font confiances au contrôle effectué par l'exo-noyau permettant de garantir l'intégrité de leur données et de ce fait font confiance aux fonctions `own()` des autres extensions.

L'isolation entre les extensions est une propriété de base d'un exo-noyau, directement issue du démultiplexage du matériel. Elle permet de garantir, à faible coût, la confidentialité et l'intégrité des données des extensions. Cependant, pour permettre une totale liberté d'implantation des abstractions par les extensions, elle impose à l'exo-noyau de valider le bon fonctionnement de quelques opérations (ici de contrôle d'accès) de chaque extension.

3.1.2 Duplication au sein des extensions

Les exo-noyaux sont des systèmes extensibles par ajout ou chargement dynamique de nouveaux services au niveau des extensions. Chaque extension ayant, grâce au démultiplexage du matériel, l'impression de fonctionner seule sur un matériel nu. Nous avons constaté précédemment que l'isolation contribue grandement à la sécurité du système. Toutefois, elle n'offre pas

que des avantages en particulier dans le contexte d'un exo-noyau pour système embarqué.

Considérons l'exemple donné figure 3.2 décrivant un exo-noyau composé de deux extensions. Chaque extension possède des objets et souhaite gérer des droits d'accès vers ceux-ci à l'aide d'une politique de protection, comme par exemple une matrice d'accès. L'exo-noyau ne fournissant pas de représentation ou d'abstraction de haut niveau du matériel, les extensions doivent tout d'abord bâtir une abstraction objet de la mémoire à partir de la vue de la mémoire que fournit l'exo-noyau (souvent sous la forme de pages de mémoire). Supposons que les politiques de protection des extensions sont des matrices d'accès fournissant les deux opérations suivantes:

- `SetUserRight(uid,oid,right)` pour attribuer des droits d'accès `right` à un utilisateur d'identifiant `uid` sur l'objet `oid` ;
- `CheckUserRight(uid,odi)` pour vérifier les droits d'accès de l'utilisateur `uid` sur l'objet `oid`[1].

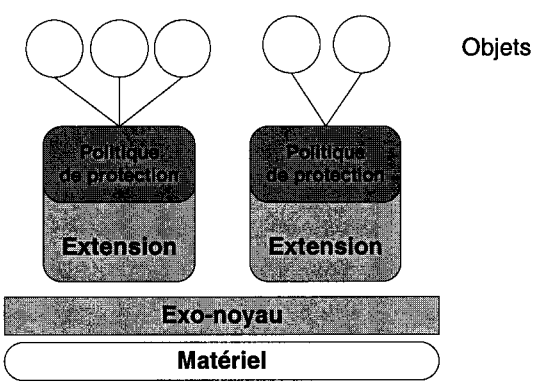


FIG. 3.2: Duplication des services.

Les deux extensions peuvent gérer dynamiquement les droits des différents traitements utilisateur sur leurs objets à l'aide de leur matrice d'accès. Chaque implantation de la politique de protection utilise ses propres traitements et structures de données. On a donc une duplication de la fonctionnalité «politique de protection» dans les deux extensions. Les deux extensions ne peuvent pas utiliser la même matrice d'accès car cela nécessiterait qu'une des deux fasse confiance à l'autre. De plus, elles ne peuvent pas utiliser les mécanismes de protection proposés par l'exo-noyau. En effet, les mécanismes de protection de l'exo-noyau ne visent que l'isolation des données des extensions. Ils ne fournissent aucun raffinement susceptible de contrôler les droits de manière plus fine.

Ainsi, l'isolation entre les extensions favorise une utilisation non optimale des ressources du système en obligeant les extensions à réécrire tous les traitements dont elles ont besoin, même si une autre extension fournit déjà ces services. Cet exemple n'est pas un exemple isolé.

[1]Les `uid` et `oid` sont générés par la matrice d'accès. Le système de type de CAMILLE assure qu'on ne puisse les forger par d'autres moyens.

Nous avons vu dans le chapitre précédent que l'isolation posait aussi problème dans le cas où l'on souhaite supporter des politiques d'ordonnancement temps réel et des tâches temps réel au niveau des extensions.

Initialement, les exo-noyaux ont été conçus pour une utilisation en tant que serveur applicatif supportant plusieurs extensions, comme par exemple des serveurs WEB, des bases de données, ou des système UNIX, sur la même machine physique. Comme les ressources (disque, mémoire, réseau) sont présentes en grande quantité sur ce type de configuration, le gaspillage n'est pas la préoccupation majeure dans ce contexte.

Dans le contexte d'un exo-noyau pour système embarqué comme CAMILLE, ce gaspillage n'est pas acceptable car les ressources comme la mémoire de code ou la mémoire de travail ne sont disponibles qu'en quantité (très) limitée. On perd ainsi l'avantage principal d'un exo-noyau pour cartes à microprocesseur. L'architecture des exo-noyaux permet de construire un système minimaliste exposant le matériel présent sur la carte. Puis de l'étendre par écriture et chargement d'extensions permettant de construire le système final (extension JAVA CARD, extension SIM ...). Le système obtenu doit cependant posséder une faible empreinte mémoire [GD01]. En plus, ce type d'extensions doivent souvent collaborer.

3.1.3 Sécurité et partage des extensions

L'isolation entre les extensions favorise la sécurité du système. Elle permet tout d'abord à l'exo-noyau d'isoler les défauts d'une extension et ainsi de ne pas compromettre le fonctionnement du système et des autres extensions. L'isolation permet aussi de garantir la confidentialité et l'intégrité des traitements et des données des extensions. Cette isolation prend encore plus d'importance dans le cas d'un système embarqué autorisant le chargement de code mobile. En effet, il est souvent difficile de différencier un code mobile sain d'un code mobile malveillant cherchant à attaquer les données du système ou à diminuer sa qualité de service (attaque en déni de service). À l'opposé de l'isolation, le partage de données ou de services favorise une meilleure utilisation des ressources du système.

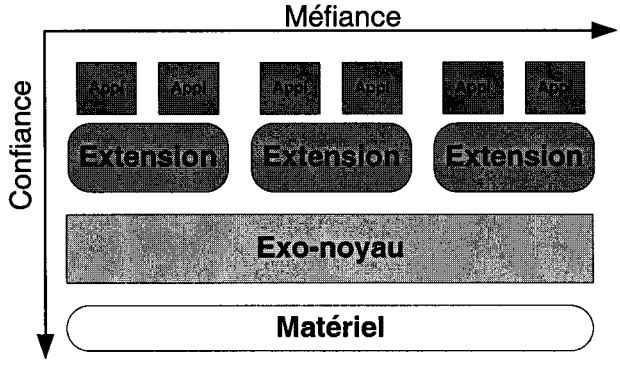


FIG. 3.3: Schéma de confiance.

La notion de confiance dans un exo-noyau est une notion verticale comme illustrée sur la figure 3.3. Une application fait confiance à l'extension sur laquelle elle repose qui, elle même, fait confiance à l'exo-noyau. Les extensions sont naturellement méfiantes envers les autres extensions. L'exo-noyau est l'unique membre de la base de confiance. Pour pouvoir partager un service de manière sûre, il doit appartenir à la base de confiance. Cette base de confiance ne peut donc se limiter à l'exo-noyau puisqu'elle doit pouvoir inclure les services de plus haut niveau. Ce qui ne saurait être fait au sein de l'exo-noyau («Exterminate all operating system abstractions» [EK95]).

Autoriser le partage de service revient à rétablir la confiance entre extensions. Le schéma de confiance vertical doit être étendu sur le plan horizontal. Ce qui revient à valider un service inconnu qui a été dynamiquement chargé par une extension comme illustré sur la figure 3.4. L'application `App3` utilise et fait confiance à un composant de l'extension 1 alors qu'elle repose sur l'extension 2.

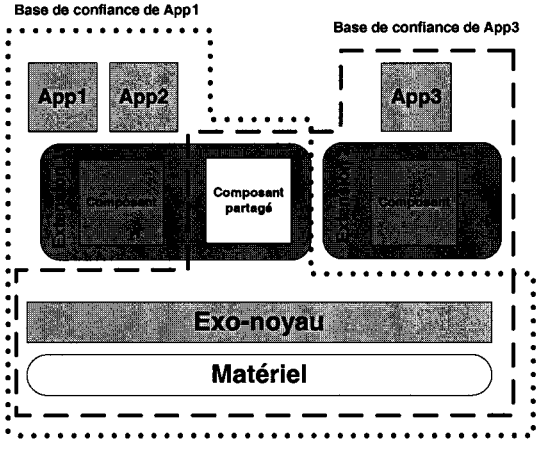


FIG. 3.4: Partage de service entre extensions.

Dans le cas d'une politique de protection, les deux extensions pourraient partager la même base logicielle (composants systèmes) permettant la gestion des droits de leurs utilisateurs sur leurs objets. L'exo-noyau doit être capable de rétablir la confiance mutuelle entre ces deux extensions.

De même, dans le cas d'extensions supportant des traitements temps réel, elles pourraient partager leurs quanta d'accès au microprocesseur et/ou leurs politiques d'ordonnancement pour résoudre le problème de l'ordonnancement aveugle que nous avons décrit sur un exemple simple dans le chapitre précédent.

3.2 Validation de services étendus

Nous avons vu que l'isolation des extensions était le mode de fonctionnement de base de l'exo-noyau. Elle permet de garantir la sécurité des extensions mais introduit des limitations concernant l'utilisation des ressources (duplication des traitements et des données). Pour optimiser l'utilisation des ressources et éviter les duplications, les extensions doivent partager des services. Pour valider le partage de services entre extensions qui ne se font pas confiance mutuellement, l'exo-noyau doit pouvoir valider le fonctionnement des services partagés. Ce faisant, la base de confiance d'une application, qui se limite initialement à l'exo-noyau et l'extension sur laquelle elle repose, pourra être étendue à d'autres extensions comme illustré sur la figure 3.4.

Nous nous intéresserons dans un premier temps à lister les garanties non fonctionnelles que l'on peut extraire du code d'un service. Nous verrons ensuite que même si les garanties non fonctionnelles permettent de connaître mieux un service, elles restent insuffisantes pour en valider le bon fonctionnement. Nous présenterons notre architecture de partage de services consistant en une phase de vérification pendant le déploiement du service associée à un contrôle dynamique réalisé à des instants caractéristiques de l'exécution du service.

3.2.1 Vérifier des garanties non fonctionnelles

Les progrès fait autour de l'analyse de programmes font que l'on sait prouver un grand nombre de propriétés non fonctionnelles sur ceux-ci.

Depuis peu, ces analyses ont pu être appliquées sur des codes mobiles, soit telles quelles, soit en utilisant le principe de distribution des codes auto-certifiés [NL97]. Un code mobile est, par définition, un traitement étranger au système qui doit donc être validé, car son producteur n'est pas forcément une entité de confiance. Un ensemble de lemmes (obligations de preuve) est généré par analyse de code par le prouveur de code. Le code et ses obligations de preuve transitent par le réseau puis sont chargés sur le consommateur de code. Le consommateur de code possède un composant particulier appelé vérifieur de code qui est en charge de reproduire la preuve établie par le producteur. C'est pour faciliter et accélérer cette opération que le producteur a ajouté des éléments de preuves. Il faut bien comprendre que de faux éléments de preuves ne peuvent pas tromper le vérifieur mais seulement faire échouer sa vérification. Si le consommateur est capable de reproduire la preuve à l'aide des éléments de preuve, le code a validé la propriété attendue, et rejoint donc l'espace de confiance. La figure 3.5 reprend l'ensemble de ces étapes.

Les codes auto-certifiés sont principalement utilisés pour valider des propriétés non fonctionnelles sur du code classique ou mobile. Voici quelques exemples de propriétés non fonctionnelles :

Programmes correctement typés [RR98, GLV99, DG02, Ler03] L'inférence de type per-

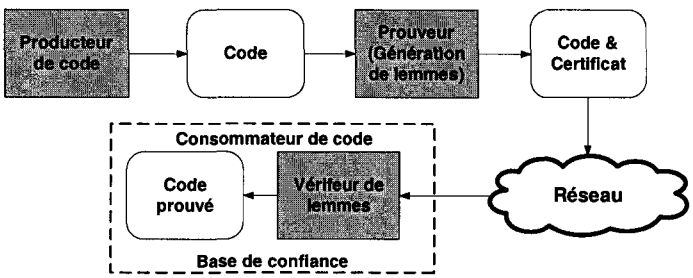


FIG. 3.5: Principe du code auto-certifié.

met de prouver qu'un code respecte les interfaces d'utilisation des traitements qu'il sollicite. Ainsi, un code correctement typé ne viole pas les règles d'usage des composants logiciels qu'il utilise, mais on ne montre pas que ces composants sont utilisés à bon escient.

Garanties transactionnelles [DGL98, LGD99] Un gestionnaire mémoire transactionnel permet de garantir que les écritures en mémoire seront correctement et entièrement réalisées ou non réalisées. Ainsi, il garantit l'atomicité des écritures en mémoire, mais n'apporte aucune information sur la nature et le rôle des données qui ont été écrites.

Non échappement [Bla99, BSF04] L'analyse d'échappement permet de s'assurer qu'il n'existe plus de référence sur un objet à la fin de l'exécution d'un traitement, mais elle ne donne aucune information sur la manière dont a été utilisé un objet.

Terminaison en temps borné [ARDG04, ARG04] Le calcul du temps d'exécution au pire cas d'un traitement permet de garantir qu'il sera réalisé dans un temps borné et connu, en revanche on ne sait rien sur ce que le microprocesseur va réaliser durant cet intervalle de temps.

Disponibilité d'une ressource [GB03a, GB03b] Les algorithmes de gestion de la disponibilité d'une ressource type algorithme du banquier permettent de garantir qu'il n'y aura pas d'interblocage d'une ressource critique, mais ils ne donnent aucune information sur l'usage qui est fait de la ressource.

Cette liste est ouverte en grande partie grâce aux principes des codes auto-certifiés qui sont génériques et applicables à beaucoup de propriétés non fonctionnelles. Toutefois, les différentes propriétés non fonctionnelles listées précédemment sont insuffisantes si l'on désire valider le partage d'un service exprimé sous forme de code mobile entre plusieurs extensions.

Les propriétés non fonctionnelles permettent, au mieux, au système de se prémunir contre les traitements malveillants, de lutter contre les attaques en déni de service, ou d'optimiser l'usage des ressources. Une garantie non fonctionnelle permet de s'assurer qu'un traitement ne fait pas quelque chose considéré comme dangereux, mais elle n'apporte aucune information sur le résultat qu'il produit. C'est pourquoi, pour rendre le partage de composants sûr, nous devons pouvoir valider des garanties fonctionnelles.

3.2.2 Garanties fonctionnelles

Un traitement peut respecter les interfaces des autres classes, ou être correctement typé, ou borné en temps d'exécution et pour autant ne pas répondre au service qu'il dit fournir. Ce problème devient préoccupant si une application souhaite utiliser les services d'une autre application sans pour autant lui faire «a priori» confiance.

Considérons l'exemple de la fonction d'ordonnancement donnée figure 3.6. Cette fonction respecte le typage de la fonction `TaskActivate( )` en lui fournissant une référence sur une tâche en paramètre. Son comportement est déterministe. Elle est de plus bornée en temps d'exécution sous réserve que la fonction `TaskActivate( )` le soit. Pourtant, elle favorise toujours la tâche `tl[0]` quelque soient les caractéristiques des tâches présentes dans la liste `tl`. Son comportement est inoffensif vis à vis des données du système ou des autres extensions. Pourtant, si cette fonction d'ordonnancement est partagée entre deux extensions elle risque de ne pas satisfaire les exigences des tâches de l'extension qui l'utilisera au profit de la seule tâche privilégiée.

```
void schedule(TaskList tl[])
{
  TaskActivate(tl[0]);
}
```

FIG. 3.6: Importance des garanties fonctionnelles.

Nous avons vu que pour améliorer l'utilisation des ressources du système, les extensions doivent collaborer en partageant, soit des structures de données, soit des traitements. L'exonoyau doit pouvoir détecter les fonctions «malveillantes» comme celle donnée en exemple sur la figure 3.6. En effet, même si ces fonctions respectent en partie ou en totalité les garanties non fonctionnelles que nous avons énoncées précédemment, elles ne sont pas pour autant fiables et ne devraient donc pas être partagées entre extensions.

Le système doit pouvoir vérifier le comportement des services partagés. Valider le comportement d'une application est un procédé complexe qui est souvent réalisé à l'aide d'une ou de plusieurs phases de test. Pour réaliser le test d'une application, il faut tout d'abord déterminer l'ensemble des paramètres possibles auxquels elle peut être soumise. Cet ensemble représente le domaine des entrées possibles de l'application. Si le testeur arrive à construire cet ensemble, il peut tester les sorties produites par l'application pour chacune des entrées possibles, et ainsi explorer en totalité l'espace de fonctionnement de l'application. Un tel test permet de prouver totalement le comportement d'une application en explorant de manière exhaustive son domaine d'entrée. Il permet d'obtenir une preuve de bon fonctionnement dans un cadre d'utilisation générale. Cependant, un tel test exhaustif est souvent impossible à appliquer sur des applications quelconques. En effet, la cardinalité du domaine d'entrée explose rapidement en fonction de la complexité du code à tester. Il devient rapidement trop grand

pour que le testeur puisse le construire et le parcourir dans sa totalité.

Ainsi, de tels tests sont souvent restreints à un sous-ensemble fini du domaine d'entrée. Les tests par sous-ensemble permettent au mieux de détecter les applications qui ne respectent pas leur spécifications. En revanche, on ne peut absolument rien dire sur une application qui respecte correctement ses spécifications en produisant des résultats correctes sur un sous-ensemble de son domaine d'entrée.

3.2.3 Principe du «test au déploiement, contrôle à l'exécution»

Une solution pour tester le bon fonctionnement d'une application est de la tester *in situ*. Au moment du déploiement de l'application, si le système possède la possibilité et la capacité de connaître précisément son domaine d'utilisation, il peut la tester sur ce domaine (souvent plus restreint) et ainsi valider son fonctionnement dans ce cadre précis d'utilisation. Toutefois, il est nécessaire de relancer cette phase de test lorsqu'une application ou un événement modifie l'état du système de telle sorte que le domaine d'entrée de l'application devient différent de celui sur lequel elle a été testée et validée.

Ce type de test en situation est particulièrement bien adapté pour valider le comportement d'un service d'un système d'exploitation pour petits objets portables et sécurisés. En effet, beaucoup de services de systèmes d'exploitation embarqués possèdent des domaines d'entrée réduits de par les faibles ressources présentes. Une matrice d'accès, par exemple, ne sera pas amenée à gérer les droits de centaines d'utilisateurs sur des centaines d'objets. Une politique d'ordonnancement embarquée ne gérera pas des centaines de tâches comme cela est le cas dans un gros système temps réel. Ainsi, beaucoup de services des systèmes d'exploitation embarqués possèdent un ou plusieurs paramètres dont le domaine est borné.

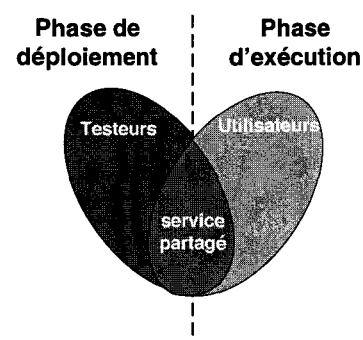


FIG. 3.7: Les deux phases d'utilisation d'un service partagé.

En exploitant ceci on pourrait donc valider le fonctionnement de services partagés entre extensions en les testant exhaustivement en situation au moment de leur déploiement dans le système. Une fois ce test validé, le service peut être partagé entre plusieurs extensions. Ainsi, les services partagés sont utilisés pendant deux phases distinctes comme illustré sur la figure 3.7 :

- la phase de test au déploiement où le composant utilisateur valide le comportement du service partagé ;
- la phase d'exécution où le composant utilisateur exploite «réellement» le service partagé.

```
static int state;

void service( )
{
  if( state < 1000 )
  {
    // fonctionnement normal
    ...
  }
  else
  {
    // fonctionnement malveillant
    ...
  }

  state++;
}
```

FIG. 3.8: Service possédant un état interne.

Que se passe t'il si un service malveillant arrive à déterminer dans quelle phase il se trouve ? Si cela se produit, il peut changer de mode de fonctionnement entre les deux phases et ainsi avoir la possibilité de tricher en mentant à son testeur, c'est-à-dire en changeant de comportement entre les phases de test et d'exécution. Pour ce faire, il lui est nécessaire de conserver un état interne entre la phase de test et la phase d'exécution ou obtenir un état à l'aide d'une interface du noyau ou d'un autre composant. La figure 3.8 donne l'exemple d'un service qui possède un état interne à travers la variable `state` et qui après mille exécutions change son comportement. Nous nous proposons d'assurer qu'un programme ne puisse changer son comportement entre plusieurs utilisations consécutives comme l'a proposé ENGLER dans sa thèse [Eng98]. Pour cela, il faut pouvoir determiner les états propres à un service au moment de son chargement et imposer leur réinitialisation avant chaque nouvelle exploitation du service.

Il existe beaucoup de moyens direct ou indirect permettant à un traitement d'obtenir un état interne. Il peut tout d'abord utiliser les fonctionnalités du système d'exploitation permettant de lire l'heure, ou encore utiliser un générateur aléatoire. Il peut aussi cacher une valeur dans un membre publique d'une classe. Il existe aussi beaucoup de moyens indirects qui sont moins triviaux à détecter mais qui permettent d'obtenir un état. Un traitement peut, par exemple, mesurer l'état de la mémoire en appelant l'allocateur du système et en comptant le nombre d'appels nécessaire pour remplir totalement la mémoire. Ce nombre est variable en fonction de l'état du système et peut être utilisé pour paramétrer un changement de comportement. Dans un système objet, un traitement peut aussi utiliser les champs statiques des classes pour faire perdurer un état entre deux appels successifs. Ainsi, pour pouvoir garantir le déterminisme d'un traitement et rendre son partage entre extensions sûr, il est

nécessaire de détecter et de contrôler tout ce qui peut faire office d'état interne (variable globale, opérations liées aux différents matériels, ...).

3.3 Formalisation de la notion de déterminisme

Nous nous intéressons dans cette section à présenter notre algorithme permettant de prouver et de vérifier le déterminisme d'un service système partagé entre plusieurs extensions. La conception de cet algorithme est issue d'une collaboration avec Yann HODIQUE et Isabelle SIMPLOT-RYL. À partir de ce premier travail, une preuve formelle de cet algorithme a pu être établie [DHSR05]. Tout le mérite de ce résultat revient à Yann HODIQUE et Isabelle SIMPLOT-RYL. Cet algorithme sera appliqué au traitement des programmes exprimés dans le langage intermédiaire FAÇADE et dans le contexte de CAMILLE, mais peut être adapté à d'autres langages et architectures de systèmes.

Le déterminisme est une propriété facile à garantir dans le cas où les traitements à vérifier sont des fonctions pures. Une telle fonction n'a pas d'état interne qui perdure entre plusieurs appels successifs et donc si elle n'appelle aucune fonction à effet de bord lui permettant de récupérer un état (générateur de nombres aléatoires, lecture de l'heure ...), son comportement sera forcément déterministe. Plus exactement, le résultat produit par son exécution dépendra uniquement de l'état de ses arguments (domaine d'entrée). En résumé, si elle est appelée plusieurs fois avec les mêmes arguments elle produira toujours le même résultat.

Toutefois, cette famille de fonctions, même si elle possède les bonnes propriétés concernant le déterminisme, n'est pas adaptée pour la programmation de services systèmes car elle est trop limitée. Prenons l'exemple d'une politique d'ordonnancement à priorité dynamique comme par exemple une politique *Least Laxity First*. Elle doit calculer et conserver la progression des tâches afin de pouvoir évaluer leur laxité qui est le critère de sélection de la tâche à élire. Ainsi, elle maintient à jour la progression de chacune des tâches, ce qui représente donc un état interne. De ce fait, elle n'est pas une fonction pure. Dans le cas d'une politique de protection de données à base de matrice d'accès, la matrice d'accès conserve les droits des utilisateurs sur les objets protégés et peut donc être considérée comme un état interne.

Il n'est donc pas envisageable de se limiter à la famille des fonctions pures car les limitations qu'elles imposent sont trop réductrices pour la programmation de services système. Nous devons donc autoriser un service à posséder des états internes. Il faut contrôler l'usage qu'il fait de ses états internes pour pouvoir distinguer les services malveillants qui les utilisent à mauvais escient pour changer dynamiquement leur comportement et tricher, des services qui les utilisent sans risque de manière légitime.

L'algorithme que nous proposons pour valider le déterminisme d'un service système en vue de le partager entre plusieurs extensions se décompose de la sorte :

- la première phase consiste à analyser le code du service au moment de son chargement

dans le système afin de détecter tous ses états internes. L'analyse distinguera deux familles d'états internes. Ceux qui peuvent être réinitialisés (par exemple les variables globales) et ceux qui ne peuvent pas l'être (générateur matériel de nombre aléatoire, quantité de mémoire disponible, ...). Les programmes possèdant des états internes de la seconde catégorie ne peuvent pas être partagés de manière fiable.

- lorsqu'une extension veut utiliser un service partagé, elle va le tester dans un environnement similaire à celui correspondant à son utilisation future (test en situation). Si le service est fonctionnel, elle va initialiser ses états internes qui appartiennent à la première famille et se mettre à l'utiliser.
- la dernière phase est réalisée avant les différentes sollicitations du service partagé. Elle consiste à réinitialiser les valeurs des états internes du service avant de le réutiliser pour l'empêcher de changer de comportement.

Prouver le déterminisme dans le cas de services possédant des états internes est une propriété complexe à valider qui nécessite une analyse statique associée à un contrôle dynamique à l'exécution du service.

3.3.1 États internes et Façade

En FAÇADE, tous les traitements sont exprimés par invocation d'une méthode d'une interface sur un autre composant en utilisant l'opérateur `invoke`. Un traitement ne peut donc utiliser une interface d'un composant que s'il a obtenu légitimement une référence sur celui-ci, par exemple à l'aide d'un de ses arguments lors d'une invocation de méthode ou bien par lecture d'un champ d'une instance. Le comportement d'un traitement FAÇADE ne dépend donc que des choses auquelles il a accès et de la façon dont il les utilise notamment en les utilisant comme arguments des différentes méthodes qu'il utilise.

Une méthode FAÇADE a tout d'abord accès à ses arguments qui lui sont fournis par le composant appelant. Elle a ensuite accès à tous les membres du composant auquel elle est rattachée au travers de son argument `this`. Enfin, elle a accès à tous les membres publiques de classe des composant formant le système. Elle peut modifier l'état global du système comme par exemple lorsqu'elle utilise une méthode d'une interface du noyau permettant l'allocation d'un objet qui modifie la quantité de mémoire disponible. Aussi, par la suite nous compterons parmi les arguments non seulement ceux qui sont passés en paramètres à la méthode mais aussi la valeur de retour, le `this` et le pseudo argument `world` répresentant l'état global de tout ce qui se trouve en dehors du contexte de la méthode.

Pour caractériser le comportement d'une méthode, Nous devons détecter comment elle utilise ses arguments. Elle peut tout d'abord utiliser ses arguments légitimement en tant qu'arguments et non en tant qu'état interne. Ceci correspond à une utilisation qui n'est pas dangereuse et qui est celle qui est la plus representée dans du code classique. Une méthode peut utiliser un de ses arguments pour stocker une donnée (par exemple dans un membre d'un

composant dont une référence est passée en argument) et la relire plus tard au cours d'une future sollicitation. Nous devons détecter ce type d'utilisation «dangereuse» des arguments.

Une méthode en FAÇADE possède des variables de travail qui ont une durée de vie locale au corps de la fonction. Toutefois, ces variables peuvent être utilisées pour contenir une copie d'un argument pour, par exemple, le retourner à l'appelant en tant que variable de retour, ou encore pour le donner en paramètre à une méthode que le traitement utilise, ou finalement l'utiliser pour stocker une donnée de manière indirecte dans la suite du programme. Notre analyse ne doit donc pas se limiter aux arguments mais aussi suivre les copies des arguments dans les différentes variables locales de la méthode.

Enfin, une méthode peut lier directement ou indirectement deux de ses arguments. Elle peut par exemple écrire un de ses arguments dans un membre d'une instance dont une référence est contenue dans un autre argument. Ainsi, une modification de l'argument contenant pourra entrainer une modification de la référence de l'argument contenu. Le type de lien entre argument le plus courant se fait au travers d'une utilisation d'une autre méthode qui peut renvoyer un de ses arguments comme valeur de retour.

En résumé, un traitement peut utiliser ses arguments de plusieurs façons que nous devons détecter et que nous nommons mode d'accès :

Définition 1 (Mode d'accès). *Soit $E = \{R\} \cup A$, où $A \subset \mathbb{N}$ l'ensemble fini des arguments d'une méthode et R sa valeur de retour. On associe un élément du treillis suivant à chaque argument de la méthode :*

- $\bot$ *: état initial, l'argument n'est pas utilisé de manière dangereuse ;*
- $\top$ *: l'argument a été utilisé de manière dangereuse pour obtenir un état interne (par exemple en écrivant ou en lisant un champ d'un objet) ;*
- $Link_\alpha$ *pour tous les $\alpha \subseteq E$: une référence sur l'argument a été récupérée par appel de méthode (par exemple en appelant une méthode qui a renvoyé une référence vers un de ses arguments à travers sa valeur de retour, noté $R \in \alpha$). Toutes les modifications futures d'un élément de α entraîneront automatiquement une modification de l'argument initial.*

$\subseteq$ est la relation d'ordre partiel. Notons $\mathcal{M}$ l'ensemble des éléments de ce treillis.

3.3.2 Détecter les états internes d'un code mobile

La détection des états internes consiste à effectuer une interprétation abstraite [CC77] du code de la méthode pour calculer sa «signature» concernant les modes d'accès à ses arguments. La signature d'une méthode qui possède n arguments est un élément de $\mathcal{M}^n$. Le but est de calculer le comportement complet d'une méthode à l'égard de ses arguments, en exploitant le fait que son comportement dépend directement du comportement des méthodes qu'elle appelle.

Notre algorithme ne s'applique pas aux méthodes associées à des opérations matérielles (transmission d'un octet sur la liaison série par exemple) qui sont dépendantes de l'architecture cible. Ces méthodes doivent être signée à la main par un expert du système qui doit prendre en compte le service qu'elles réalisent. Il en existe une centaine qui regroupent principalement les opérations arithmétiques et logiques (définies par les composants `CardBool`, `CardByte`, `CardShort` et `CardInt`) ainsi que certaines opérations relatives à la manipulation de la mémoire et des entrées/sorties. Ces méthodes forment une base de confiance initiale que l'on va étoffer en calculant puis en ajoutant les signatures de toutes les méthodes du noyau indépendantes du fonctionnement d'un matériel.

L'algorithme de calcul des signatures de l'ensemble des méthodes FAÇADE du noyau peut rencontrer des invocations vers des méthodes qui n'ont pas encore été signées. Toutes les méthodes non encore signées se voient donc assigner une signature par défaut avec tous leurs arguments à l'état $\perp$. Ceci peut nous amener à remettre en cause la signature d'une méthode pour laquelle une mauvaise hypothèse avait été faite (comme par exemple, suite au calcul de la signature d'une fonction qu'elle utilise).

Pour construire le catalogue de signatures, l'expert système doit avoir, au préalable, signé toutes les méthodes natives. L'algorithme va ensuite construire par itérations successives les signatures des méthodes présentes sous forme de code FAÇADE. Il commence par leur assigner une signature initiale avec tous les arguments à l'état $\perp$. L'algorithme fonctionne par parcours successifs du catalogue de signatures. Toutes les méthodes rencontrées doivent obligatoirement faire partie de la base de confiance ou du catalogue. Lorsque l'algorithme remet en cause la signature d'une méthode il doit propager ce changement sur toutes les méthodes qui en dépendent. Un point fixe est atteint lorsque plus aucune signature ne doit être remise en cause.

Le calcul de la signature d'une méthode est réalisé par interprétation abstraite du code FAÇADE de la méthode à signer et exploite les signatures des fonctions appelées par cette méthode pour calculer les modes d'accès aux arguments.

Pour chaque instruction FAÇADE de la méthode analysée, l'algorithme maintient une liste des dépendances qui existent entre les variables locales au corps de la fonction et les arguments. Une ligne de dépendance est un tableau de taille $|arguments| + |locales| + |temporaires|$ dans lequel chaque entrée est une liste des arguments qui sont contenus dans la variable correspondant à l'entrée. On dit que a *dépend de* b si une modification de b peut entraîner une modification de a (en d'autres termes un chemin du flot d'exécution a mis une référence vers a dans b). La liste des dépendances de la i^{me} instruction est une fonction de ses prédécesseurs. L'algorithme de calcul des modes d'accès d'une méthode atteint donc un point fixe lors de la stabilité du tableau de dépendances. L'algorithme calcule aussi la signature courante de la méthode qui sera la signature finale caractérisant le comportement de la méthode lorsque le point fixe est atteint.

Nous considérons qu'une méthode est découpée en blocs de base correspondant à une suite d'instructions comprise entre deux points de branchements. Nous associons à chaque bloc de base un drapeau permettant de savoir si le bloc doit être réévalué suite à un changement de la liste de dépendances de son point d'entrée engendré par un saut vers celui-ci.

Initialisation:

- marquer le premier bloc comme étant à évaluer ;
- initialiser la liste de dépendances de la première instruction en notant que chaque variable ne dépend que d'elle même.

Boucle principale:

- tant qu'il y a des blocs à évaluer ;
- choisir un bloc parmi ceux à évaluer comme par exemple celui le plus proche du début du programme ;
- évaluer linéairement les instructions du bloc choisi :
  - si l'instruction courante est une instruction de branchement (`Jump`, `JumpIf` et `JumpList`), injecter les dépendances de l'instruction courante dans celles du ou des successeurs de l'instruction de branchement. Si l'injection ajoute des nouvelles dépendances dans la ou les destinations, marquer les blocs correspondant comme étant à réévaluer,
  - dans le cas d'une instruction `return` qui est un point de sortie de la fonction reporter les dépendances courantes sur la signature finale de la méthode :
    - en assignant l'état $Link_R$ à tous les arguments qui contiennent la variable de retour dans leur liste de dépendances.
    - en propageant les dépendances croisées entre arguments dans la signature finale à l'aide des éléments $Link_\alpha$ du treillis $\mathcal{M}$ (si la liste de dépendances de l'argument i contient l'argument j, alors le mode d'accès de l'argument i doit passer à l'état $Link_j$).
  - dans le cas d'un appel de méthode, il faut transformer la liste de dépendances courante et la signature de la méthode appelante en appliquant les modes d'accès (*i.e.* la signature) de la méthode appelée, puis l'injecter dans la liste de dépendances de l'instruction suivante.

Propagation des modes d'accès lors de l'appel d'une méthode sur la liste de dépendances et la signature de la méthode appelante :

- construire une liste de dépendances temporaires en supprimant la variable qui va recevoir le retour de la fonction de la liste de dépendances de l'instruction `invoke` de la méthode appelante ;

- parcourir les modes d'accès de chaque argument de la fonction appelée :
  - si l'argument a le mode d'accès $\bot$, ne rien faire
  - dans le cas d'un argument à l'état $\top$, transformer la signature de la méthode appelante en passant à l'état $\top$ tous ses arguments qui dépendent de l'argument considéré de la méthode appelée
  - dans le cas d'un argument à l'état $Link_R$, injecter une nouvelle dépendance vers la variable qui recoit le retour de la fonction appelée dans la liste de dépendances de toutes les variables de l'appelant qui dépendent de l'argument considéré de l'appelé
  - dans le cas d'un argument à l'état $Link_n$, injecter les dépendances du n^{ieme} argument dans les dépendances de tous les arguments de l'appelant qui dépendent de l'argument considéré de l'appelé

Des versions en pseudo code C++ de ces deux algorithmes sont disponibles en annexe de ce document (cf 5.3 page 115 et 5.4 page 116).

La signature obtenue est donc «l'unification» des dépendances pour l'ensemble des points de sortie de la fonction en tenant compte des marquages à $\top$ des arguments utilisés de manière dangereuse par les méthodes qu'utilise le traitement.

Cet algorithme permet de calculer les modes d'accès d'une fonction à ses arguments et par la même de caractériser l'utilisation qu'une fonction fait de ses arguments au travers de son flôt d'exécution et des méthodes qu'elle utilise. Il permet en particulier de détecter les arguments qui sont utilisés en tant qu'état interne ou qui permettent d'en obtenir un. Ces arguments se retrouvent marqués à l'aide du mode d'accès $\top$. Yann HODIQUE et Isabelle SIMPLOT-RYL ont établi une preuve formelle de cette algorithme publiée dans [DHSR05].

3.3.3 Vers une preuve d'inférence du déterminisme

L'algorithme de détection des états internes d'un code mobile, que nous avons présenté précédemment, a pour vocation d'être utilisé sur des petits objets portables et sécurisés. Nous devons donc prendre en compte les spécificités du matériel embarqué en faisant particulièrement attention à la puissance de calcul, ainsi qu'à la quantité de mémoire nécessaire au calcul des modes d'accès d'un service à ses arguments. L'algorithme que nous avons présenté précédemment n'est pas directement utilisable dans le domaine de l'informatique embarqué fortement contrainte pour les raisons suivantes.

La consommation mémoire nécessaire pour le tableau de dépendances est le facteur limitant le plus important. En effet, une ligne du tableau de dépendance est elle même un tableau de taille $|arguments| + |locales| + |temporaires|$ dans lequel chaque entrée est une liste des arguments contenus dans la variable correspondant à l'entrée. L'algorithme de calcul des modes d'accès nécessite autant de ces lignes qu'il y a d'instructions dans le programme. Ceci nous amène donc à une consommation mémoire théorique en nombre de bits de l'ordre de $(|instructions| * ((|arguments| + |locales| + |temporaires|) * (|arguments|)))$ en utilisant

un bit pour coder la dépendance d'une variable envers un argument. À ceci doit être ajouté la taille de la signature finale de la méthode, c'est-à-dire un élément du treillis $\mathcal{M}$ par argument (classique ou spécial) de la méthode. Sur du code complexe, la quantité de mémoire nécessaire pour la totalité du tableau de dépendance peut facilement dépasser les capacités en mémoire de travail d'une carte à microprocesseur (souvent de l'ordre de 3 à 4 Ko de mémoire RAM). Il serait donc nécessaire de mettre en place des politiques de cache entre la mémoire de travail et la mémoire persistante pour pouvoir calculer la signature d'un code complexe.

Le deuxième point qui rend notre algorithme trop complexe pour être réalisé tel quel sur le système embarqué concerne le fait que le calcul de la signature d'une méthode peut conduire à remettre en cause des signatures déjà calculées. À titre d'information, l'algorithme permettant de calculer le catalogue de signatures des méthodes formant le noyau CAMILLE nécessite quatre itérations successives sur l'ensemble des méthodes. De plus, l'algorithme de signature d'une méthode nécessite aussi plusieurs itérations sur les instructions avant d'obtenir un point fixe qui est la condition d'arret de l'algorithme. Le fait de devoir remettre en cause une signature déjà calculée est un problème important dans CAMILLE. En effet, CAMILLE utilise un compilateur à la volée qui transforme le code FAÇADE en code natif au moment de son chargement dans le système. Le code FAÇADE est vérifié puis compilé et n'est donc pas conservé plus longtemps que nécessaire. Or, l'algorithme de calcul de signature consécutif à une remise en cause travaille sur le code intermédiaire FAÇADE que l'on devrait donc conserver.

Nous devons donc distribuer le calcul des modes d'accès entre le terminal et la carte à puce. Une bonne hypothèse couramment admise dans le monde des cartes à microprocesseur est d'essayer, dans la mesure du possible, de rendre linéaire les traitements lourds que doit réaliser la carte. L'exemple typique concerne la vérification du typage d'un programme qui est distribué entre le système embarqué et son hôte suivant le principe des codes autocertifiés [NL97]. L'hôte calcule une preuve de typage et l'intègre au code. Le tout est envoyé et vérifié par la carte. La vérification du typage des codes mobiles FAÇADE est réalisée de la sorte dans CAMILLE [RCG00b, GLV99].

Nous nous proposons de distribuer le calcul des modes d'accès entre le terminal et le système embarqué à la manière des codes autocertifiants (PCC [NL97]). Le terminal calcule tout d'abord la signature d'une fonction à l'aide de l'algorithme donné précédemment. Il conserve les listes de dépendances obtenues aux points de branchement qu'il fournit comme preuve au système embarqué. La taille de la preuve est donc fonction du nombre de points de branchement du programme. Le système embarqué vérifie linéairement la signature proposée en exploitant les informations aux points de saut. En résumé, le terminal envoit le code FAÇADE de la méthode, une proposition de signature et les éléments de preuve permettant au système embarqué de vérifier la conformité de la signature avec le code de la méthode. Notons qu'il est possible de fabriquer une preuve qui marque certains éléments comme plus dangereux qu'il ne sont vraiment. Ce qui compte, c'est que l'algorithme de vérification détecte

tous les «véritables» éléments dangereux (mode d'accès $\top$).

L'algorithme de vérification embarqué de la signature d'une méthode est obtenu par application direct des principes des codes autocertifiants sur l'algorithme de calcul des modes d'accès d'une méthode à ses arguments. Il consiste en un unique parcours linéaire du code de la méthode en faisant évoluer une unique ligne de dépendances de la manière suivante :

- Lorsque l'instruction courante est la destination d'un saut, on vérifie que la ligne de dépendances courante inclut les dépendances contenues dans la preuve, puis on la remet à jour à l'aide de celle contenue dans la preuve pour poursuivre la vérification.
- Dans le cas où l'instruction courante est une instruction de branchement, on vérifie que la ligne de dépendances est compatible avec celles contenues dans la preuve pour chaque destination possible du branchement.
- Dans le cas d'un appel de méthode, on transforme la ligne de dépendances en appliquant la signature de la méthode appelée et on vérifie que la signature à vérifier est compatible avec cet appel de méthode sur un modèle similaire à l'algorithme de propagation des modes d'accès présenté précédement.
- De même, dans le cas d'une instruction `return`, on vérifie que la signature proposée inclut bien les bons modes d'accès.

Une version en pseudo code C++ de cet algorithme est disponible en annexe de ce document (cf 5.5 page 117).

Si l'on ne rencontre aucune erreur, la signature proposée avec le code est valide. Yann HODIQUE et Isabelle SIMPLOT-RYL ont établi une preuve formelle de cette algorithme publiée dans [DHSR05].

Comme l'algorithme de vérification utilise les signatures des méthodes appelées, nous devons avoir, au préalable, calculé le catalogue de signatures pour toutes les méthodes du noyau et disposer de ce catalogue sur le système embarqué. Les objets CAMILLE représentant les blocs de code (`CardCode`) sont donc étoffés afin de pouvoir retrouver le mode d'accès de la méthode qu'ils décrivent dans le catalogue de signatures.

FAÇADE étant un langage orienté objet, l'algorithme de calcul et de vérification des signatures de mode d'accès doit tenir compte des problèmes liés à l'héritage et à la surcharge de méthode. En effet, lors d'un appel de méthode d'instance, on ne peut connaître avec certitude la méthode qui sera réellement appelée (cela peut être n'importe laquelle des implantations d'une des sous classes). Ainsi, lorsque le système embarqué charge le code d'une méthode qui surcharge une méthode déjà présente, il doit s'assurer que leurs signatures sont compatibles. Dans un premier temps, nous imposons que les signatures des modes d'accès soient identiques. Nous reviendrons plus tard sur ce choix et proposerons d'autres solutions alternatives.

3.3.4 Vérifier le bon fonctionnement d'un service partagé

Les modes d'accès peuvent être exploités pour valider un service partagé entre différentes extensions d'un exo-noyau. Le mode d'accès d'un code à ses arguments n'est pas une propriété fonctionnelle, toutefois nous pouvons l'exploiter pour obtenir des garanties sur son fonctionnement.

Nous allons illustrer les différentes phases de l'algorithme de validation d'un service partagé à l'aide de l'exemple décrit sur la figure 3.9. Une extension (dite fournisseuse) est déjà présente dans l'exo-noyau et possède un service de protection de données exploitant une matrice d'accès qu'une extension en cours de chargement (dite consommatrice) aimerait utiliser pour gérer les droits de ses utilisateurs sur ses ressources.

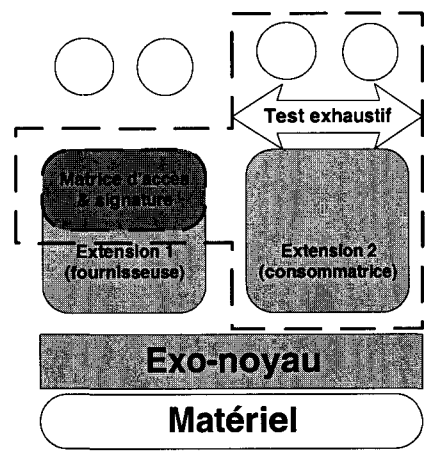


FIG. 3.9: Fiabilisation du partage de service entre extensions.

L'extension consommatrice doit tout d'abord vérifier que la signature du service de protection de donnée qu'elle souhaite utiliser est compatible avec ses prérequis (le service utilise-t'il le pseudo argument `world` ou `this` comme état interne). La signature peut être moins restrictive que celle imposée par l'extension consommatrice du service.

Une fois la signature du service validée, l'extension consommatrice doit vérifier que l'implantation du service est fonctionnelle, c'est-à-dire qu'elle sera capable de gérer les droits de ses utilisateurs sur ses ressources pendant la phase d'exécution.

Pour ce faire, l'extension consommatrice va enregistrer dans la matrice d'accès les droits de tous ses utilisateurs puis elle va tester exhaustivement la matrice d'accès afin de vérifier qu'elle a bien enregistré les bons droits. Dans le cas où le service n'est pas capable de répondre à ses attentes, elle devra chercher une autre implantation de ce service auprès d'une autre extension. Si la matrice d'accès est fonctionnelle pour ses ressources, l'extension consommatrice va réinitialiser les état internes de la matrice d'accès qui sont connus grâce à la signature des modes d'accès aux arguments, puis elle va pouvoir l'utiliser. Un service qui a été testé de manière exhaustive en situation d'exécution et qui possède une signature conforme aux

spécifications requises par le concepteur de l'extension ne pourra pas changer son mode de fonctionnement pendant l'exécution. Il s'exécutera donc de la même façon quelque soit la phase (test ou exécution).

Le choix de la signature autorisée pour l'implantation d'un service est un paramètre critique qui doit être determiné avec soin par l'extension consommatrice. En effet, si elle est trop restrictive en interdisant par exemple tout état interne, très peu d'extensions seront en mesure de proposer une implantation du service respectant la signature. Il est par exemple très difficile d'implémenter une politique d'ordonnancement purement déterministe, c'est-à-dire qui ne possède pas d'argument ayant le mode d'accès $\top$.

Suivant le type de service que l'on désire partager de manière fiable, il peut être nécessaire de réaliser, en plus du test exhaustif en situation d'exécution, un contrôle du ou des états internes à certains moments caractéristiques de l'exécution du service. Ce contrôle n'est pas nécessaire dans le cas de notre matrice d'accès.

En résumé, voici les différentes phases permettant de prouver et de vérifier le bon fonctionnement d'un service système exprimé sous forme de code mobile comme FAÇADE en vue de le partager entre plusieurs extensions pour optimiser l'usage des ressources présentes sur le système :

1. Calculer les mode d'accès du service à partager au moment de son chargement dans le système.
2. Vérifier que cette signature est compatible avec celle imposée par le concepteur de la nouvelle extension qui souhaite utiliser ce service.
3. Tester exhaustivement le service en situation d'exécution.
4. En cas de succès, réinitialiser le service au travers de ses états internes en exploitant les modes d'accès aux arguments.
5. Utiliser le service partagé.
6. Éventuellement, contrôler le ou les états internes du service lorsque l'on atteint des points caractéristiques de l'exécution.

La combinaison de l'algorithme de calcul des modes d'accès, du test exhaustif lors du déploiement et de la réinitialisation des états internes permet de prouver le bon fonctionnement d'un service. Ces différentes phases permettent d'obtenir des garanties concernant le fonctionnement d'un service sur un sous-ensemble de son domaine d'entrée, restreint à celui avec lequel il va être utilisé.

Le chapitre suivant est consacré à la description des modifications apportées à l'architecture de l'exo-noyau CAMILLE permettant de supporter du temps réel extensible. Cette architecture autorise le chargement dynamique de politiques d'ordonnancements. Une extension ne possédant pas de politique d'ordonnancement peut utiliser de manière fiable celle d'une

autre extension grâce à l'algorithme de validation du partage que nous avons présenté dans ce chapitre. Plusieurs extensions peuvent aussi mutualiser leurs accès au microprocesseur, et choisir une de leurs politiques d'ordonnancement, pour gérer l'ensemble de leurs tâches mises en commun. Ce partage à plusieurs extensions repose lui aussi sur une extension de l'algorithme présenté au cours de ce chapitre. La mutualisation et la collaboration permettent en particulier, de résoudre le problème de l'ordonnancement aveugle que nous avons présenté dans le chapitre précédent.

Chapitre 4

Exo-noyau pour applications en temps réel

«Le Centre n'a émis aucun message [] Qui l'a émis alors ? [] Quand mon père parlait de la métasphére [] il disait toujours qu'elle était remplie de lions de tigres et d'ours [] Des lions, des tigres et des ours, répétais-je» – **Dan Simmons, Hyperion 3, Endymion, 1996.**

Nous allons, dans ce chapitre, présenter l'architecture que nous avons mise en place dans CAMILLERT afin de supporter des ordonnanceurs temps réel au même titre que n'importe quelle autre extension de l'exo-noyau. Nous ne cherchons pas à rendre l'exo-noyau CAMILLE temps réel mais plutôt à fournir des services élémentaires permettant aux extensions de garantir le bon usage du microprocesseur par les applications qui le sollicitent. Cette architecture autorise le chargement dynamique d'extensions temps réel qui peuvent amener leur propre politique d'ordonnancement. L'isolation naturelle existant entre les extensions est un facteur qui peut contribuer à refuser le chargement d'un ensemble de tâches qu'un système non isolé pourrait ordonnancer. Pour optimiser l'usage du microprocesseur, nous avons introduit une nouvelle famille d'extensions dites collaboratives qui peuvent mettre en commun leurs quanta de temps et partager une même politique d'ordonnancement. Pour valider le partage des politiques d'ordonnancement, nous utilisons les algorithmes présentés dans le chapitre précédent. L'architecture hiérarchique implantée dans CAMILLERT permet de faire coexister des extensions supportant des traitements classiques et partageant équitablement le temps entre ces traitements, des extensions ayant des besoins temps réel[1] et des extensions collaboratives. Il offre à chaque extension qui le nécessite des garanties concernant l'accès au microprocesseur.

Dans un premier temps, nous présentons les différents composants systèmes permettant d'exposer le microprocesseur. Nous expliquons ensuite comment nous pouvons exploiter les

[1]Par abus de langage nous appelons extension temps réel, une extension supportant des applications ayant des besoins concernant leurs échéances temporelles.

services rendus par ces composants de base pour fournir des garanties temps réel aux extensions qui veulent supporter des traitements temps réel. Finalement, nous proposons une architecture hiérarchique plus évoluée permettant d'améliorer l'accès à la ressource d'exécution en autorisant les extensions qui le désirent à partager leurs quanta de temps et leurs politiques d'ordonnancement.

4.1 Exposition du microprocesseur dans Camille

Si l'on se réfère aux principes architecturaux des exo-noyaux proposés par ENGLER [Eng98], le rôle principal d'un exo-noyau est d'exposer le matériel tout en le fiabilisant (c'est-à-dire en empêchant tout programme d'en faire un usage impropre). Pour pouvoir partager le microprocesseur entre plusieurs extensions, nous devons donc, dans un premier temps, fournir un ensemble de composants systèmes permettant aux extensions de se partager et d'utiliser conjointement le microprocesseur.

Dans un premier temps, nous présenterons les couches basses de CAMILLERT permettant de proposer à chaque extension un composant système virtualisant l'état du microprocesseur. Nous présenterons ensuite les notions de contextes d'exécution permettant aux extensions de supporter l'exécution de plusieurs traitements en partageant leur processeur virtuel. Finalement, nous nous intéresserons aux problèmes liés au démultiplexage des interruptions matérielles vers les extensions.

4.1.1 Notion de processeur virtuel

Pour permettre l'exécution concurrente de plusieurs extensions au sein de CAMILLERT, nous devons exposer le microprocesseur aux extensions au travers d'un composant logiciel. Plus précisément, ce composant logiciel doit virtualiser l'état et les opérations du microprocesseur. CAMILLERT est un système d'exploitation qui a été conçu pour les cartes à puce. Les cartes à puce étant mono processeur, une seule extension est active à un instant donné.

Le composant `CardVCPU` représente une portion du microprocesseur. À chaque extension est associée une instance du composant `CardVCPU` matérialisant la fraction du microprocesseur à laquelle elle a accès. Les processeurs virtuels ne sont pas actifs en continu. Ils sont actifs uniquement lorsque l'extension qu'ils représentent peut s'exécuter. Ils sont endormis lorsqu'une autre extension est en cours d'exécution. L'exo-noyau doit donc prévenir une extension, au travers de son processeur virtuel, qu'elle doit se réveiller où se rendormir. Le composant logiciel `CardVCPU` fournit à cet effet les deux interfaces suivantes :

- `PreSlice( )` permet à l'exo-noyau de notifier l'extension de la (ré)activation du processeur virtuel ;
- `PostSlice( )` pour prévenir l'extension active qu'elle va se rendormir immédiatement après terminaison de la méthode `PostSlice( )`.

Les extensions qui veulent disposer d'un processeur virtuel au fonctionnement particulier peuvent créer une sous-classe du composant `CardVCPU` et surcharger ces deux méthodes afin d'être notifiées de ces évènements en fonction de leurs besoins.

Au plus bas niveau, l'exo-noyau doit partager équitablement le microprocesseur entre ses différentes extensions. Il utilise donc une simple politique de type *round-robin*, donnée figure 4.1, qui a l'avantage d'être impartiale. L'ordonnanceur est associé à un compteur du matériel permettant un découpage équitable du microprocesseur.

Le noyau maintient à jour une liste de tous les processeurs virtuels en fonctionnement et leur donne alternativement la main en allumant puis éteignant leur processeur virtuel. Une extension peut enregistrer son processeur virtuel dans cette liste à l'aide de l'interface `RegisterVCPU( )` fournie par le composant `CardKernel`. La figure 4.2 donne un exemple de ce mode de fonctionnement en présence de deux extensions ayant donc chacune accès à la moitié du microprocesseur. De manière générale, un exo-noyau supportant n extensions leur donne l'illusion de fonctionner en continu sur un processeur n fois moins puissant que le microprocesseur physique. Cette couche minimale de virtualisation du microprocesseur permet son partage équitable en n processeurs virtuels qui sont chacun associés à une extension.

```
void _CardKernel_Schedule( )
{
    CardVCPU *vcpu;

    if( nb_vcpu == 0 ) return;

    vcpu=CardObject_DynamicCast( CardVCPU,
          CardPOMem256_Get( VCPU_List, curr_vcpu )
        );

    curr_vcpu++;

    if( curr_vcpu == nb_vcpu )
        curr_vcpu=0;

    run_VCPU=vcpu;

    CardVCPU_PreSlice( vcpu );

    // unreachable
}

void _CardKernel_ScheduleEnd(void)
{
    CardVCPU_PostSlice( run_VCPU );

    CardKernel_Schedule( );
}
```

FIG. 4.1: Couche basse d'ordonnancement de CAMILLERT.

Les instances de processeurs virtuels, au même titre que les extensions qu'ils représentent, doivent perdurer aux différents redémarrages du système. Ainsi, nous avons choisi de rendre les composants `CardVCPU` persistants. Les différentes structures de données que manipulent les extensions n'étant pas toutes persistantes, l'exo-noyau doit notifier des redémarrages du système à toutes les extensions. Chaque extension peut ainsi réallouer les instances de compo-

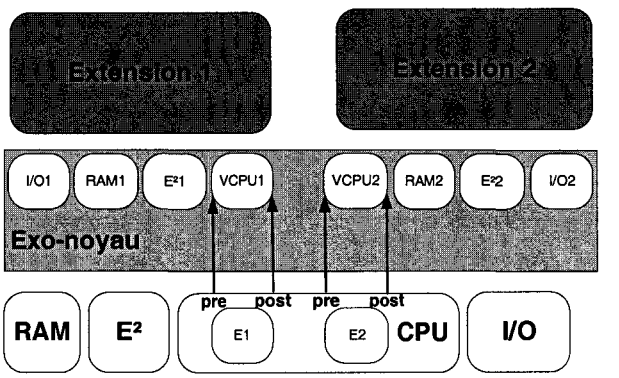


FIG. 4.2: Démultiplexage du microprocesseur.

sants volatiles qui ont disparu suite à l'arrêt du système, mais aussi réinitialiser les structures de données qui le nécessitent. Notons qu'une carte à puce peut être arrachée de son lecteur à tout moment par son porteur et donc être privée instantanément d'énergie. Les systèmes d'exploitation pour cartes à microprocesseur doivent résister à cet évènement de la vie courante appelé «arrachement». À chaque remise sous tension de la carte, l'exo-noyau parcourt la liste de processeur virtuel et notifie à chacun du redémarrage en appelant les méthodes `Restart()` définies par les composants `CardVCPU` et surchargées par les sous-classes qui souhaitent traiter cet évènement de manière spécifique.

4.1.2 Contexte d'exécution

La couche minimale de virtualisation du microprocesseur que nous avons présentée précédemment permet de partager équitablement la ressource d'exécution entre les différentes extensions présentes. Dans un exo-noyau, les extensions sont des entités logicielles chargées de la gestion d'un certain nombre d'applications qui leurs sont rattachées. Ainsi, nous devons autoriser une extension à supporter l'exécution concurrente de plusieurs applications au sein de son processeur virtuel.

Un traitement qui s'exécute est caractérisé à tout instant par sa pile d'exécution et par l'état de tous les registres du microprocesseur. Le composant `CardCTX` virtualise le contexte d'exécution d'un traitement. Une instance du composant `CardCTX` va représenter le contexte d'exécution d'un traitement au cours de toute sa durée de vie, c'est-à-dire jusqu'à sa terminaison ou alors jusqu'au prochain redémarrage de la carte. De plus, le contexte d'exécution d'une application sera mis à jour régulièrement. Ainsi, les instances du composant `CardCTX` sont volatiles. Un contexte d'exécution comprend la pile d'exécution du traitement auquel il est associé ainsi qu'une sauvegarde de l'état des registres pertinents du microprocesseur lorsqu'il est endormi. Une instance de `CardCTX` est initialisée à partir d'une instance de `CardCode` qui contient toutes les informations associées à un code (nombre d'arguments, signature de type, type de code, pointeur de fonction si c'est un code natif, temps d'exécution au pire

cas, ...). Créer un contexte d'exécution consiste donc à allouer une pile d'exécution pour le traitement qui lui est associé et à initialiser cette pile afin que le traitement soit en mesure d'être exécuté par le microprocesseur.

L'exo-noyau doit pouvoir contrôler un contexte d'exécution afin de détecter qu'il ne dépasse pas la capacité de la pile d'exécution qui lui a été allouée. Si le matériel le permet (présence d'un circuit de gestion avancée de la mémoire type `MMU`), il peut utiliser des barrières en écriture pour détecter les dépassements de capacité de la pile d'exécution. Toutefois, peu de cartes à microprocesseur sont dotées de ce type de matériel. Sur une carte «classique», nous modifions le générateur de code embarqué, de telle sorte qu'il injecte un code de vérification dynamique de la hauteur de la pile dans les codes d'entête des fonctions compilées.

Pour autoriser les extensions à gérer plusieurs applications au-dessus de leur processeur virtuel, nous proposons donc le modèle des activités d'ordonnancement suivant :

L'exo-noyau partage le microprocesseur entre les extensions au travers des instances de processeur virtuel. Il sait geler l'état du microprocesseur dans une instance de `CardCTX` en sauvegardant notamment les valeurs de tous les registres pertinents. Il sait recharger le microprocesseur à partir des informations contenues dans un contexte d'exécution et ainsi reprendre l'exécution d'un traitement suspendu.

Un processeur virtuel représente une extension. Cette extension peut partager son processeur virtuel entre plusieurs applications en utilisant par exemple une simple politique *round-robin* au travers d'implantations dédiées de `PreSlice( )` et de `PostSlice( )`.

Un contexte d'exécution est construit et initialisé à partir d'une instance de `CardCode`. Un contexte peut demander à l'exo-noyau de (ré)activer le traitement qu'il représente en rechargeant le microprocesseur à partir de l'état gelé dans son instance. Il peut aussi demander à l'exo-noyau de geler l'état du microprocesseur dans son instance et de recharger le microprocesseur à partir d'un état gelé dans une autre instance de contexte d'exécution. Cette opération et à la base de l'ordonnancement, elle permet de remplacer le flot d'exécution du microprocesseur par celui d'un autre traitement.

La figure 4.3 donne un exemple des interactions entre ces différents composants amenant à la (ré)activation du traitement gelé dans le premier contexte de la première extension. On remarque bien que la virtualisation du microprocesseur en processeur virtuel et en contexte d'exécution introduit plus d'appels que dans le cas d'un simple noyau temps réel gérant ses tâches au plus bas niveau. Cette hiérarchisation à deux niveaux introduit des surcoûts par rapport à une architecture traditionnelle que nous évaluerons dans le chapitre suivant. Nous pouvons quand même noter que les coûts d'exécution relatifs au démultiplexage des évènements correspondant à l'activation et au gel d'un processeur virtuel doivent être les plus faibles possibles car ils imputent sur le temps alloué aux applications. Ils diminuent donc le temps directement exploitable par les applications dans un processeur virtuel.

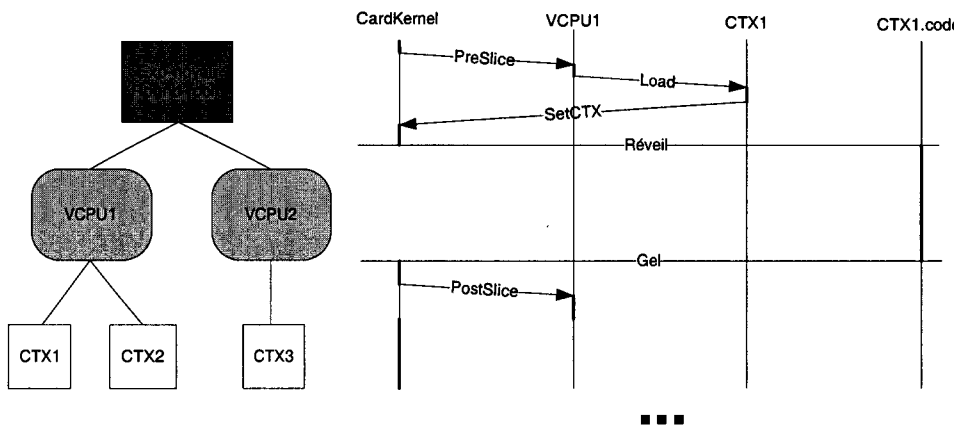


FIG. 4.3: Processeurs virtuels et contextes d'exécutions.

4.1.3 Démultiplexage des interruptions

Les extensions, qui sont des systèmes virtuels, ne peuvent pas, pour des raisons évidentes de sécurité de fonctionnement du système, avoir un accès direct aux interruptions fournies par le matériel sous-jacent. En effet, si tel était le cas, un code mobile pourrait par exemple s'enregistrer comme récepteur des interruptions d'un compteur matériel. En l'initialisant avec une valeur adéquate, il pourrait monopoliser le microprocesseur. En revanche, une extension doit légitimement pouvoir demander à être notifiée de l'arrivée d'une interruption comme par exemple lors de la réception d'un octet sur la ligne de communication ou du dépassement d'un compteur matériel. Nous devons donc hiérarchiser le traitement des interruptions entre l'exo-noyau et les différentes extensions qui se partagent le microprocesseur et le matériel.

Au niveau de l'exo-noyau nous avons introduit un nouveau composant système appelé `CardITManager`. Ce composant est en charge de la gestion des interruptions matérielles. Il comporte une méthode par interruption matérielle qui est chargée de recevoir et d'accuser réception des interruptions auprès du matériel.

Le `CardITManager`, en plus de traiter les interruptions matérielles, est en charge de la notification de la réception d'une interruption à toutes les extensions qui en ont fait la demande (message de type *un vers tous*). Pour ce faire, nous ajoutons des méthodes de réception des interruptions démultipléxées aux composants virtualisant le microprocesseur. Une extension désirant être notifiée des réceptions d'une interruption doit enregistrer son processeur virtuel auprès de l'exo-noyau au travers de l'instance du composant `CardITManager`.

Le démultiplexage des interruptions soulève un certain nombre de problèmes concernant le moment où les notifications doivent être réalisées. Une interruption est un évènement inopiné qui peut arriver à tout moment, en particulier pendant l'exécution d'une application d'une extension. Si une interruption matérielle se produit au moment de l'exécution d'une application, l'exo-noyau ne peut pas notifier de l'arrivée de cet évènement à toutes les ex-

tensions concernées en appelant leurs interfaces correspondantes. En effet, s'il faisait de la sorte, les coûts d'appels et d'exécution des méthodes de notification se feraient au détriment de l'extension en cours de fonctionnement. Les notifications des interruptions doivent donc être mises en attente pour toutes les extensions concernées autre que l'extension active, afin ne pas consommer le temps de l'extension active.

Lorsqu'une interruption se produit, il y a deux cas possibles pour les notifications :

- si l'extension active est concernée, l'exo-noyau lui délivre immédiatement la notification en appelant l'interface correspondant à l'interruption ;
- pour toutes les extensions concernées en dehors de l'extension active, l'exo-noyau met en attente les notifications pour ne pas qu'elles imputent sur le temps de l'application en fonctionnement. L'ordonnanceur, avant de donner la main à une extension en réveillant son processeur virtuel, va appeler la méthode de notification correspondant à l'interruption en attente. L'exo-noyau va ensuite réveiller le processeur virtuel associé à l'extension. Dans le cas où plusieurs échéances de la même interruption sont en attente, l'exo-noyau doit appeler la méthode de notification autant de fois qu'il y a eu d'interruptions depuis la dernière notification.

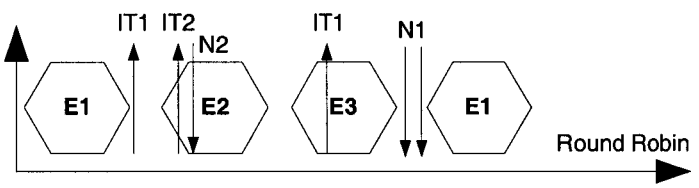


FIG. 4.4: Notification des interruptions.

La figure 4.4 donne un exemple en présence de trois extensions. La première extension a demandé à être notifiée des interruptions de type 1, la deuxième des interruptions de type 2. Une interruption de type 2 se produit pendant que la deuxième extension s'exécute, la notification a donc lieu immédiatement. En fin de cycle, avant d'exécuter la première extension, l'exo-noyau appelle deux fois de suite sa fonction associée aux interruptions de type 1 car deux de ces interruptions sont en attente d'être notifiées.

Il est important de noter que les coûts de démultiplexage des interruptions doivent être pris en compte dans le calcul du temps qui est réellement disponible pour chaque extension. En effet, les appels des méthodes de notification imputent sur le temps alloué à l'extension active. L'exo-noyau doit pouvoir disposer des temps d'exécution des fonctions de notification de réception d'une interruption. Dans l'exemple précédent, l'extension `E1` est notifiée de l'arrivée de deux interruptions du type 1, les appels à la méthode de notification associée à ce type d'interruption sont réalisés dans le quantum de l'extension. Le temps restant pour les applications de `E1` est donc diminué d'autant. L'exo-noyau doit aussi prendre en compte les demandes de modification de paramètre du matériel par les extensions qui peuvent entraîner

une modification de la fréquence des arrivées des interruptions (modification des registres de configuration des taux de transfert de la ligne série).

Nous évaluerons par la suite les performances de ce schéma de démultiplexage des interruptions au sein de l'architecture CAMILLERT. Un exemple type porte sur la mise en place de composants permettant de partager la ligne série de la carte à microprocesseur entre plusieurs extensions afin que toutes puissent recevoir les octets entrants. Un autre exemple possible concerne le démultiplexage d'un compteur matériel afin que les extensions puissent découper leur unité élémentaire de processeur virtuel en sous-unités et réaliser ainsi un sous-ordonnancement de leurs applications.

4.2 Temps réel et processeurs virtuels

Nous avons décris précédemment les différents composants systèmes que nous avons ajoutés à l'architecture initiale de CAMILLE afin d'exposer et de partager l'accès au microprocesseur. Ces composants permettent de partager le microprocesseur entre plusieurs extensions et les autorisent à supporter l'exécution de plusieurs applications au sein de leur processeur virtuel.

Pour pouvoir transformer cette architecture de démultiplexage du microprocesseur en architecture temps réel, nous devons pouvoir garantir l'accès à la ressource d'exécution quelque soit la configuration, la charge et l'état du système. Nous décrirons dans un premier temps les solutions mises en œuvre dans CAMILLERT pour garantir l'accès au microprocesseur aux extensions supportant des tâches temps réel. Nous nous intéresserons ensuite aux problèmes soulevés par les retards des notifications des interruptions matérielles.

4.2.1 Garanties d'accès à la ressource d'exécution

Pour que les extensions puissent offrir des garanties à leurs applications temps réel, l'exo-noyau doit garantir l'accès des extensions au microprocesseur.

La couche minimale de virtualisation du microprocesseur que nous avons présentée au début de ce chapitre permet de partager équitablement le microprocesseur en n processeurs virtuels qui sont chacun associés à une des n extensions. Une tranche du microprocesseur est appelée quantum de temps et sa taille est fixée par l'exo-noyau. L'exo-noyau respecte donc un principe de base appelé principe d'équité en $\frac{1}{n}$: une extension a accès à un quantum de temps tous les n quanta.

Comme nous l'avons illustré précédemment sur la figure 4.3, pour pouvoir garantir l'accès au microprocesseur, il est nécessaire de borner et de rendre le plus faible possible les coûts d'exécution des primitives de notification des réveils et des mises en veille des processeurs virtuels. Il en va de même pour les interfaces de l'exo-noyau qui sont en charge des contextes d'exécution.

Une extension peut exploiter le fait qu'elle a accès au microprocesseur une fois tous les n quanta pour supporter des applications temps réel. À partir de cette information, elle peut valider que sa politique d'ordonnancement pourra ou non respecter les échéances temporelles d'une nouvelle tâche au sein de son $n^{ième}$ du microprocesseur. L'exo-noyau doit notifier à l'extension qu'elle n'a pas accès à la totalité du quantum de temps. La partie du quantum utile à une application est en effet légèrement inférieure à la totalité d'un $n^{ième}$ du microprocesseur. Il faut tenir compte des coûts d'exécution de l'ordonnanceur *round-robin* ainsi que des coûts d'activation et de gel des contextes d'exécution. Ces coûts sont connus et maîtrisés par le concepteur de l'exo-noyau car les méthodes qui y sont associées font partie de la base de confiance. Aux coûts liés à l'ordonnancement, l'exo-noyau doit ajouter les coûts des opérations de démultiplexage des différents types d'interruptions matérielles. Les coûts du démultiplexage des interruptions sont plus difficiles à prendre en compte car ils sont dépendants de la configuration du matériel. Par exemple, la reconfiguration du débit de la ligne série va entraîner une augmentation du nombre potentiel d'octets reçus dans le temps correspondant à un quantum et donc du même coup une augmentation du nombre potentiel d'interruptions du type série pouvant arriver pendant l'exécution d'une application. Ainsi, le temps «réellement» exploitable par une application dans un quantum de temps s'obtient en enlevant les coûts d'ordonnancement et de traitements des interruptions de la taille du quantum de temps élémentaire comme illustré avec la formule 4.1.

$$Temps_{utile} = Quantum_{elementaire} - \epsilon\{ordonnanceur\} - \epsilon\{interruptions\} \quad (4.1)$$

Deux problèmes peuvent survenir lors de l'ajout d'une nouvelle extension à l'aide de l'interface `RegisterVCPU( )` fournie par l'exo-noyau :

- le nombre de processeurs virtuels passe de n à $n + 1$;
- la reconfiguration des paramètres du matériel peut entraîner une augmentation du nombre potentiel d'interruptions.

L'exo-noyau ne dispose pas des informations suffisantes lui permettant de contrôler et de valider si l'admission d'une nouvelle extension ne va pas entraîner une extension à ne plus avoir la capacité de satisfaire les échéances temporelles de ses applications temps réel. En effet, l'exo-noyau n'a pas d'informations sur le type d'algorithme d'ordonnancement qui est utilisé par une extension, il n'a pas non plus accès aux informations concernant les besoins temporels des applications supportées au niveau des extensions. Il y a séparation des préoccupations liées à l'acte d'ordonnancement entre l'exo-noyau et les extensions. Le rôle de l'exo-noyau se limite à virtualiser l'état du microprocesseur, à notifier les extensions de certains évènements et à garantir l'accès au microprocesseur. Les extensions sont quant à elles chargées de gérer et de satisfaire les besoins de leurs applications à l'aide de ce que leur fourni l'exo-noyau.

L'acceptation ou non d'une nouvelle extension est un évènement qui peut remettre en cause la capacité d'une extension à respecter les échéances temporelles de ses applications. Ainsi, la

réponse à cette question ne peut pas venir de l'exo-noyau mais doit venir de l'ensemble des extensions présentes. Pour régler le problème de l'acceptation d'une nouvelle extension, nous utilisons un algorithme à base de vote qui se décompose en deux étapes successives :

La phase de vote consiste à parcourir l'ensemble des extensions en fonctionnement afin de les interroger concernant l'ajout de la nouvelle extension. L'exo-noyau demande à chaque extension, à l'aide de l'interface `Vote( )` de son processeur virtuel, si elle accepte l'installation de la nouvelle extension. Plus précisément, l'exo-noyau demande à chaque extension si elle sera toujours en mesure de satisfaire les échéances temporelles de ses application avec un taux d'accès au microprocesseur de $\frac{1}{n+1}$ au lieu de $\frac{1}{n}$.

La phase de notification consiste à parcourir la liste des processeurs virtuels et à les informer du passage effectif de n à $n+1$ extensions en invoquant l'interface `Commit( )` fournie par les processeur virtuels. Les extensions peuvent ainsi remettre à jour les structures de données utilisées par leurs politiques d'ordonnancement.

Le résultat de la phase de vote doit être unanime pour que l'exo-noyau accepte d'installer la nouvelle extension. Dans le cas où le vote est un échec, l'exo-noyau peut remonter l'identifiant de la ou des extensions qui ont refusé l'installation auprès de l'utilisateur ou de l'expert du système. Ainsi, la personne concernée pourra faire un choix entre les extensions mutuellement incompatibles vis-à-vis de leurs exigences concernant l'accès à la ressource d'exécution.

La phase de notification du passage à $n+1$ extensions ne peut pas se faire immédiatement après la fin de la phase de vote car cela pourrait remettre en cause les échéances des applications restant à être exécutées sur la fin du cycle courant du *round-robin* de l'exo-noyau. Ainsi l'ajout est retardé jusqu'à la fin du cycle courant, c'est-à-dire après activation du processeur virtuel de l'extension en dernière position dans la liste des processeurs virtuels.

Une extension qui désire avoir à sa disposition plusieurs quanta par cycle d'ordonnancement *round-robin* peut enregistrer plusieurs fois son processeur virtuel auprès de l'exo-noyau tant que les votes des autres extensions sont unanimes.

Ce mécanisme à base d'une phase de vote et d'une phase de notification permet à l'exo-noyau de garantir à l'accès à la ressource d'exécution aux différentes extensions qui peuvent ainsi par transitivité garantir l'accès aux processeurs virtuels à leurs applications temps réel.

4.2.2 Maîtrise des retards sur les réceptions des interruptions

Un problème important qui peut apparaître suite au partage du microprocesseur entre plusieurs extensions concerne la réactivité des extensions vis-à-vis des interruptions.

Considérons l'exemple simple d'un exo-noyau partageant le microprocesseur entre trois extensions comme décrit sur la figure 4.5. Supposons que la première extension ait demandé à être notifiée lors de la réception d'une interruption comme par exemple celle associée à l'arrivée d'un octet entrant sur la ligne série. L'ordonnanceur de l'exo-noyau donne alternativement la main aux processeurs virtuels de chaque extension. Si une interruption arrive juste après

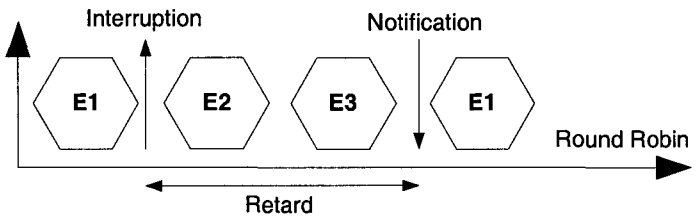


FIG. 4.5: Retard sur les notifications des interruptions.

l'exécution de la première extension, cette extension recevra la notification au prochain réveil de son processeur virtuel, c'est-à-dire après deux quanta de temps. Plus généralement, en présence de n extension, la notification d'une interruption peut se produire au maximum $n-1$ quanta après réception de l'interruption. Dans le cadre de notre exemple, la première extension est incapable de répondre à la réception d'un octet dans un laps de temps inférieur à deux quanta. Si cette extension est en charge de la génération des clefs cryptographiques de session dans une carte SIM, suivant la taille d'un quantum de temps elle ne pourra pas produire la clef en temps et en heure, la communication ne pourra donc pas être maintenue.

Ainsi, il est légitime qu'une extension puisse revendiquer un délai maximum entre la réception d'une interruption et l'invocation de son interface de notification par l'exo-noyau. Le coût du démultiplexage des interruptions doit géner au minimum le fonctionnement des extensions.

Une première solution à ce problème consiste à autoriser les extensions à pouvoir négocier la taille d'un quantum de temps. Le retard maximum entre réception et notification d'une interruption reste toujours égal à $n-1$ quanta, mais en diminuant la taille d'un quantum les extensions peuvent augmenter leur réactivité face aux interruptions. La taille d'un quantum de temps étant la même pour toutes les extensions, les demandes de changements doivent être validées par toutes les extensions avant que ceux-ci ne deviennent effectifs. Nous utilisons un mécanisme de vote similaire à celui effectué avant chaque installation d'une nouvelle extension. Il repose sur les deux étapes suivantes :

La phase de vote consiste à parcourir l'ensemble des extensions en fonctionnement afin de les interroger à l'aide de l'interface `VoteQuantum(CardInt newquantum)` de leur processeur virtuel concernant le changement de la taille du quantum de temps.

La phase de notification consiste à informer tous les processeurs virtuels du changement de la taille du quantum de temps en invoquant leur interface `CommitQuantum(CardInt newquantum)`. Les extensions peuvent ainsi remettre à jour les structures de données utilisées par leurs politiques d'ordonnancement. Cette phase est retardée jusqu'à la fin du cycle d'ordonnancement du *round-robin* pour les mêmes raisons que celles exposées précédemment concernant l'ajout d'une nouvelle extension.

Cette solution ne répond pas à tous les problèmes. En effet, lorsque la taille d'un quantum

de temps atteint un certain seuil, le système se met à payer de plus en plus les coûts de la virtualisation du microprocesseur et des commutations des contextes d'exécution. Diminuer la taille du quantum de temps entraîne une augmentation du nombre de commutation de tâche et donc implique une diminution du temps restant dans un quantum de temps pour l'exécution d'une application. Les coûts de commutation des contextes d'exécution sont gommés si les quanta de temps sont plus longs.

4.3 Ordonnancement collaboratif des extensions

Pour que des extensions qui ne se font pas mutuellement confiance puissent collaborer afin de mieux résoudre un problème ou utiliser une ressource commune, il est nécessaire qu'un tiers soit utilisé afin de rétablir la confiance entre ces extensions. Par exemple, dans l'architecture classique des exo-noyaux, les extensions ne se font pas confiance mutuellement mais font confiance à l'exo-noyau qui, grâce à ses différents mécanismes de protection, partage le matériel de manière sécurisée entre ses extensions. L'exo-noyau joue donc le rôle de tiers de confiance pour l'ensemble des extensions concernant l'accès au matériel.

Dans un premier temps, nous allons décrire un canevas logiciel permettant d'introduire des tiers de confiance génériques chargés de valider qu'une fonction répond aux attentes d'un ensemble d'extensions partageant une ou plusieurs propriétés communes. Nous expliquerons en quoi l'architecture de virtualisation du microprocesseur que nous avons décrit précédemment peut exploiter un tel tiers de confiance et nous montrerons comment nous pouvons sortir l'ordonnanceur *round-robin* de l'exo-noyau et le remplacer par toute autre politique d'ordonnancement amenée par une extension. Enfin, nous décrirons l'architecture d'ordonnanceur hiérarchique que nous avons mis en place dans CAMILLERT et qui permet la coexistence d'extensions standards et d'extensions supportant des tâches temps réel et partageant une même politique d'ordonnancement.

4.3.1 Test d'admission générique

L'installation d'une nouvelle extension au sein d'un groupe d'extensions possédant des caractéristiques communes introduit souvent des problèmes concernant une remise en cause des garanties de bon fonctionnement de ces mêmes extensions. La nouvelle extension peut par exemple consommer beaucoup de mémoire et ainsi limiter les allocations des autres extensions, elle peut aussi utiliser beaucoup de puissance de calcul et remettre en cause la capacité des autres extensions à servir leurs applications. Le problème récurrent, dans le cas où l'exo-noyau fourni des garanties à ses extensions, est de pouvoir déterminer si l'installation d'une nouvelle extension ne va pas remettre en cause les garanties des autres extensions. L'exo-noyau fait donc office de tiers de confiance; il est l'entité qui garantit par exemple l'accès à une ressource. Toutefois, cela nécessite que l'exo-noyau possède des informations concernant

le type des services qui sont réalisés par les extensions ainsi que sur les algorithmes mis en œuvre. Or, comme nous avons pu le voir précédemment avec les critères d'admission d'une extension au sein d'un groupe d'extension cherchant à garantir l'accès à la ressource d'exécution, cela n'est pas toujours possible. L'exo-noyau ne connaît pas forcement le type de tâche qu'une extension supporte au sein de son processeur virtuel car cette extension a pu être chargée dynamiquement. Pour accepter ou non l'installation d'une nouvelle extension, il n'a pas d'autre moyen que d'interroger chaque extension.

Nous souhaitons rendre générique le processus d'admission d'une extension au sein d'un groupe partageant des propriétés communes ainsi qu'une de leurs interfaces comme par exemple une politique d'ordonnancement. Pour ce faire nous proposons d'introduire un composant particulier appelé coordinateur et qui va être en charge de la réalisation des différents tests suite à la demande d'admission d'une nouvelle extension au sein d'un groupe. Il va s'aider des extensions déjà présentes dans ce groupe. Un coordinateur est donc un composant fédérateur d'un ensemble d'extensions, il est aussi le testeur d'une fonction sur un domaine qu'il ne connaît pas mais qu'il peut construire à l'aide des extensions. Les extensions d'un groupe partagent une de leurs implantations d'une fonction qui a été élue par les extensions membres du groupe au travers du coordinateur suite à la dernière admission d'une extension.

Un coordinateur supporte les opérations suivantes :

1. gérer les demandes d'ajouts une extension au groupe d'extensions qu'il fédère à l'aide de l'interface `Register( )` ;
2. interroger les extensions membres sur l'ajout en effectuant une phase de vote qui doit obligatoirement être unanime ;
3. construire le domaine de test de la fonction à partager qui est obtenu par fusion des domaines de chaque extension (*i.e.* $D_{test} = \bigcup_{i=0}^{n} D\{Extension_i\}$);
4. tester les différentes implantations de la fonction à partager fournies par les extensions en :
   (a) parcourant l'intervalle de test et en le transformant par application de la fonction à tester (*i.e.* $D_{sortie} = plan(D_{test})$) ,
   (b) interrogeant chaque extension pour déterminer si le résultat produit par la fonction testée est conforme à leurs attentes ;
5. dans le cas ou une implantation répond aux attentes de toutes les extensions, installer la nouvelle extension ;
6. utiliser la fonction élue, c'est-à-dire appliquer le plan au domaine réel ;
7. utiliser le domaine de test en fonctionnement réel pour réinitialiser les états internes de la fonction partagée, états qui ont été détectés lors du chargement de l'extension à l'aide de l'algorithme de calcul des modes d'accès présenté dans le chapitre précédent ;
8. gérer la suppression d'une extension du groupe à l'aide de l'interface `Remove( )`.

La réinitialisation des états internes du service partagé entre chaque cycle de fonctionnement permet de garantir qu'il se comportera comme pendant la phase de test. Ainsi, si le résultat qu'il produit a été validé par toutes les extensions, il fonctionnera sur un régime identique à chaque cycle.

Nous pouvons remarquer que l'opérateur de fusion des domaines des extensions permettant de construire le domaine sur lequel la fonction partagée va être testée est dépendant des propriétés et critères que partagent les extensions. Par exemple, dans le cas d'extensions gérant des tâches temps réel périodiques, fusionner deux domaines consiste à calculer leur plus petit commun multiple (aussi appelé hyperpériode).

Une autre remarque importante concerne le problème du retrait d'une extension membre du groupe. Il est en effet possible que ce soit l'implantation de la fonction partagée de cette extension qui soit utilisée. Il est possible qu'après le retrait de cette extension, aucune implantation restante ne puisse satisfaire les besoins des extensions restantes.

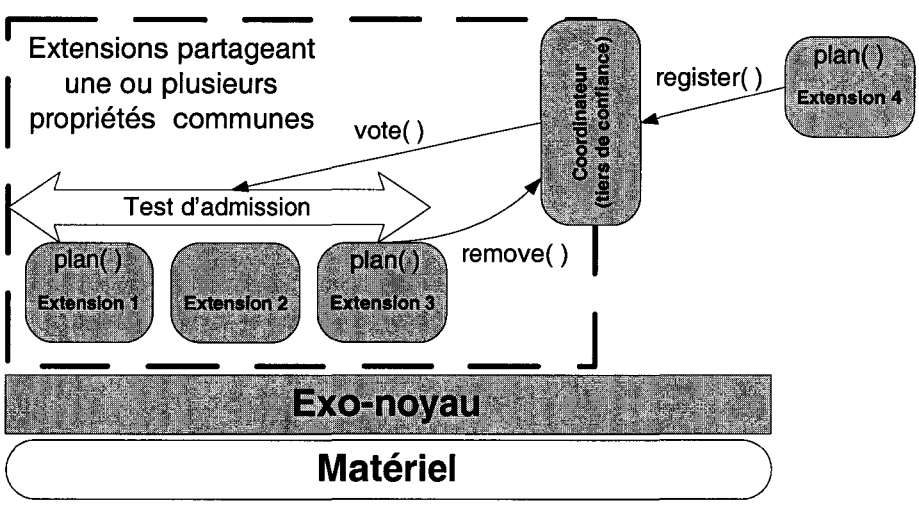


FIG. 4.6: Test d'admission générique.

La figure 4.6 illustre les différentes phases d'interaction entre le coordinateur et les extensions qu'il fédère pouvant amener à l'installation de la nouvelle extension dans le cas d'un consensus des extensions formant le groupe. L'extension quatre désire rentrer dans le groupe. Elle demande donc au coordinateur fédérant le groupe de passer le test d'admission. Le coordinateur va demander un vote pour collecter l'avis des extensions membres en se basant sur le critère qu'elles partagent. Si le vote échoue, il va refuser l'installation de l'extension candidate et notifier à l'utilisateur des votes des extensions. Dans le cas d'un vote unanime, le coordinateur teste de manière exhaustive les différentes implantations de la fonction partagée afin de voir si l'une d'elle répond aux attentes des extensions membres et de l'extension candidate. L'existence d'une telle fonction est le dernier critère permettant l'admission ou non de l'extension candidate.

4.3.2 Support du partage dans les processeurs virtuels

L'architecture de virtualisation du microprocesseur permettant de garantir l'accès à la ressource d'exécution aux extensions qui le désirent que nous avons présentée exploite une phase de vote. L'exo-noyau joue le rôle du coordinateur chargé de contrôler l'admission d'une nouvelle extension. Chaque extension connaît les besoins de ses applications et en déduit donc le taux d'accès au microprocesseur dont elle a besoin pour servir ses applications. De même, les extensions peuvent jouer sur la taille des quanta de temps afin de répondre en temps et en heure aux interruptions matérielles démultiplexées par l'exo-noyau. Ainsi, les critères d'admission d'une nouvelle extension sont donc le nombre et la taille des quanta de temps qui sont alloués à une extension pendant un cycle d'ordonnancement. Ces critères sont inconnus de l'exo-noyau qui doit donc nécessairement interroger les extensions.

Nous avons initialement placé la politique d'ordonnancement *round-robin* dans l'exo-noyau afin de respecter un principe minimal d'équité concernant l'accès au microprocesseur. Cette politique est donc quelque chose que l'exo-noyau impose à l'ensemble des extensions qui peuvent être chargées dynamiquement. Or, il se peut qu'elle ne convienne à aucune extension alors qu'elles pourraient être d'accord pour fonctionner avec une autre politique. Ceci représente une limitation forte à l'extensibilité de notre exo-noyau. Nous nous proposons donc d'utiliser l'exo-noyau comme coordinateur primitif chargé de la sélection d'une politique de partage des quanta de temps entre les différentes extensions. Cette politique va être choisie par l'exo-noyau parmi l'ensemble des politiques qui ont été chargées dynamiquement en même temps que les extensions. Ainsi, l'exo-noyau et ses extensions pourront fonctionner avec n'importe laquelle des politiques de partage du microprocesseur chargées dynamiquement, sous réserve qu'elle satisfasse les besoins de toutes les extensions, c'est-à-dire le nombre et la taille des quanta de temps alloués à une extension pendant un cycle d'ordonnancement.

Chaque extension peut donc amener avec elle sa politique de partage de l'accès au microprocesseur au travers de son processeur virtuel. Cette politique voit le microprocesseur comme un vecteur de quantum de temps et doit associer une extension à chacun des quanta de temps de ce vecteur. Pour ce faire, elle exploite les caractéristiques des extensions concernant l'accès à la ressource d'exécution.

Initialement, l'exo-noyau est dépourvu d'extension; ainsi la première extension qui va demander à s'installer va prendre l'accès exclusif au microprocesseur. Considérons maintenant le régime de fonctionnement illustré sur la figure 4.7 : deux extensions se partagent l'accès au microprocesseur et une troisième est candidate au test d'admission. Elles se sont mises d'accord suite au dernier ajout d'une extension sur un cycle d'ordonnancement de six quanta de temps. La première extension demande la moitié du microprocesseur, la deuxième un sixième et la troisième un tiers. L'exo-noyau va tester l'implantation du plan d'ordonnancement de la première extension qui associe une extension à chaque quantum du vecteur de temps. Le plan résultant est soumis à l'approbation des autres extensions et les extensions deux et trois

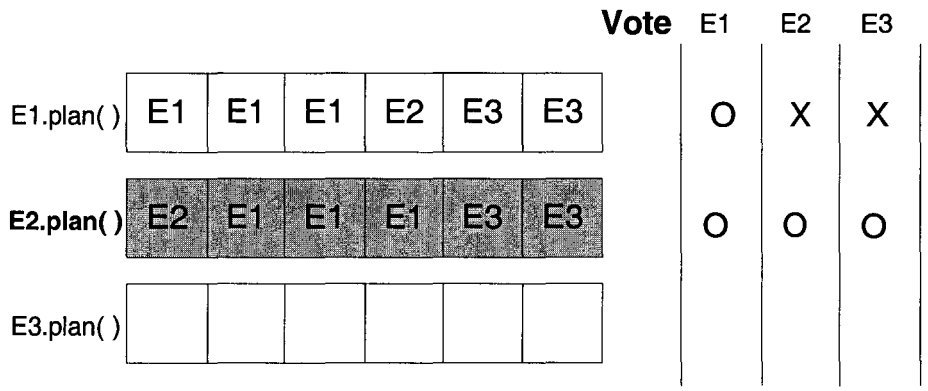


FIG. 4.7: Choix du plan d'ordonnancement.

le refusent. L'exo-noyau essaye donc le plan d'ordonnancement suivant. Ce plan reçoit l'approbation des trois extensions. L'exo-noyau peut donc accepter l'installation de l'extension candidate car il y a eu un consensus sur la taille du plan d'ordonnancement, sur la taille des quanta de temps et l'implantation du plan d'ordonnancement de la deuxième extension retient l'approbation des trois extensions. L'exo-noyau va ensuite réinitialiser les états internes de cette fonction, puis se mettre à l'utiliser pour allouer les quanta de temps. Pour chaque quantum, l'exo-noyau interroge le plan élu qui lui fournit le numéro de l'extension à activer. L'exo-noyau réveille puis rendort le processeur virtuel associé à cette extension. À chaque fin du cycle d'ordonnancement, l'exo-noyau réinitialise les états internes du plan d'ordonnancement afin de garantir qu'il fournira le même plan d'ordonnancement d'un cycle à l'autre. L'exploitation des modes d'accès pour réinitialiser le plan d'ordonnancement permet de garantir son déterminisme.

L'utilisation de l'exo-noyau en tant que coordinateur permet donc de retirer l'ordonnanceur *round-robin* initialement mis en place dans l'exo-noyau et de le remplacer par une politique plus évoluée amenée par une des extensions. Cela permet par exemple d'utiliser un ordonnanceur gérant la précédence entre plusieurs extensions si cela ne dérange pas les autres extensions. Le partage du microprocesseur peut ainsi être adapté dynamiquement au fil de l'évolution du système suite aux ajouts et retraits d'extensions sous réserve d'un consensus de l'ensemble des extensions en fonctionnement.

Le fait de remplacer l'ordonnanceur *round-robin* par un plan d'ordonnancement d'une des extensions introduit un problème concernant la différence entre la taille d'un quantum de temps et la taille de la partie exploitable par les applications. Nous avons vu précédemment (cf formule 4.1), qu'une partie du quantum de temps est consommée par la politique d'ordonnancement des processeurs virtuels ainsi que par le mécanisme de notification des réceptions des interruptions. La politique d'ordonnancement *round-robin* initiale faisant partie de l'exo-noyau, ses temps d'exécution sont donc connus et maîtrisés par le concepteur du sys-

tème. De plus, du fait de sa grande simplicité, ses coûts d'exécution sont faibles. Pour pouvoir remplacer cette politique par une politique plus complexe qui a été chargée dynamiquement avec une extension et conserver les garanties d'accès à la ressource d'exécution, l'exo-noyau doit avoir à sa disposition le temps d'exécution au pire cas du plan d'ordonnancement de remplacement. Ainsi, lors du chargement d'une nouvelle sous-classe de processeur virtuel surchargeant la méthode `Plan( )`, l'exo-noyau doit calculer son temps d'exécution au pire cas afin de pouvoir, par la suite, garantir l'accès au microprocesseur à l'ensemble des extensions en fonctionnement dans le cas d'une utilisation de cette implantation.

4.3.3 Mise en œuvre sur une hiérarchie d'ordonnanceur temps réel

L'architecture de virtualisation du microprocesseur que nous venons de décrire permet aux extensions de garantir à leurs applications l'accès à leur processeur virtuel. L'exo-noyau joue le rôle du coordinateur chargé de l'élection du plan d'ordonnancement des extensions et aussi des négociations de la taille et du nombre des quanta de temps associés à chaque extension.

Toutefois, cette architecture ne permet pas de résoudre directement le problème de l'ordonnancement aveugle que nous avons décrit dans le chapitre 2 (cf section 2.2.2 page 42). Ce problème est issu de l'isolation entre les extensions instaurée par les principes architecturaux des exo-noyaux et aussi de l'absence de confiance existant entre les extensions. L'utilisation d'un coordinateur permet de rétablir une certaine forme de confiance entre les extensions fédérées par celui-ci.

Nous souhaitons pousser plus avant la coopération entre extensions. Nous allons former un groupe d'extensions temps réel partageant une même politique d'ordonnancement et mutualisant leurs accès au microprocesseur. Ceci leur permet de mieux servir les besoins temporels de leurs applications. Cette collaboration avancée entre extensions temps réel est une solution permettant de répondre au problème de l'ordonnancement aveugle (cf section 2.2.2 page 42).

Ces extensions manipulent toutes des tâches temps réel périodiques, aussi elles utilisent des contextes d'exécution étendus appelés `CardRTCTX`. Ces contextes comprennent :

- une pile d'exécution (par héritage des contextes d'exécution classiques) ;
- les attributs temps réel classiques :
  - un temps minimal d'activation,
  - une période,
  - une quantité de microprocesseur (temps d'exécution au pire cas ou `WCET`),
  - une échéance temporelle.

Pour représenter les extensions qui veulent collaborer, nous leurs assignons une instance de processeur virtuel collaboratif supportant les opérations de base nécessaires au partage d'une politique d'ordonnancement des tâches temps réel et à la mutualisation des quanta de temps. Un processeur virtuel collaboratif est une sous-classe du composant `CardVCPU` spécialisée (étendue) avec les deux méthodes suivantes :

- `CardByte PlanRT(CardInt nb_task, CardTOMem32 *task_list, CardInt curr_slice, CardInt hyperperiod, CardTB32 *cmem)`
- `void PreSliceEx(CardByte ctx_id)`

La première interface est responsable de l'ordonnancement d'un ensemble de tâches temps réel. Elle prend en paramètre une liste de tâches temps réel, leur hyperpériode et le numéro du quantum courant. Elle doit renvoyer l'identifiant de la tâche élue pour le quantum courant en se basant sur les attributs temporels de l'ensemble des tâches. Le coordinateur des extensions temps réel collaborative fournit à chacune de ces fonctions une zone de mémoire qu'elle peut utiliser comme état interne pour stocker par exemple la progression des tâches. Cette zone de mémoire est réinitialisée à chaque cycle d'ordonnancement par le coordinateur afin de garantir que le plan d'ordonnancement sera identique d'une hyperpériode à l'autre.

L'interface `PreSliceEx( )` permet à la politique d'ordonnancement d'une extension collaborative de demander à une autre extension l'activation de la tâche qui est associée au contexte d'exécution d'identifiant `ctx_id`.

Le test d'admission d'une nouvelle extension collaborative se fait en deux étapes :

1. Une extension collaborative étant avant tout une extension comme les autres, elle doit tout d'abord être acceptée par l'ensemble des extensions classiques et collaboratives déjà présentes. Elle va donc négocier la taille et le nombre de quanta de temps dont elle a besoin pour servir ses applications auprès de l'exo-noyau comme nous l'avons décrit précédemment.
2. Une fois que les extensions en fonctionnement sont unanimes, l'extension collaborative candidate doit passer le test d'admission pour rejoindre le groupe des extensions collaboratives. Le coordinateur de celles-ci va lui faire passer le test.
   (a) Dans un premier temps, si l'extension candidate amène sa propre fonction d'ordonnancement de tâche, le coordinateur va vérifier que cette fonction respecte la signature (cf section 3.3.2 page 66) concernant les modes d'accès à ses arguments et au système (par exemple une signature du type ($world : \bot$, $this : \bot$, $nb\_task : \bot$, $task\_list : \bot$, $curr\_slice : \bot$, $hyperperiod : \bot$, $cmem : \top$) n'autorise la fonction à n'utiliser que la zone de mémoire fournie par le coordinateur comme état interne)
   (b) Le coordinateur va ensuite tester les politiques d'ordonnancement fournies par les extensions collaboratives. Il va leur demander de calculer le plan d'ordonnancement des tâches des extensions collaboratives sur l'ensemble des quanta de temps qu'elles mettent en commun. Il va ensuite soumettre le résultat à l'approbation de toutes les extensions. Dans le cas où une extension n'est pas en accord avec le résultat obtenu, le coordinateur teste l'ordonnanceur suivant.
   (c) Si aucune implantation des politiques d'ordonnancement ne retient l'unanimité, le coordinateur refuse l'installation de l'extension candidate.
   (d) Dans le cas où une implantation est acceptée par toutes les extensions collabora-

tives, l'extension associée devient l'extension élue qui sera en charge d'ordonnancer les tâches de toutes les extensions collaboratives.

(e) Le coordinateur va installer la nouvelle extension collaborative et se mettre à utiliser la politique d'ordonnancement élue après réinitialisation de ses états internes.

```
void _CardCRTVCPUCoordinator_GrantSlice(CardCRTVCPUCoordinator *this_)
{
    CardCRTVCPU *crtvcpu;

    this_->taskid=CardCRTVCPU_PlanRT(
            choosen, this_->ctx_list,
            this_->nb_ctx, this_->curr_slice,
            this_->hyperperdiod, this_->workingmem);

    if (this_->taskid == TASK_IDLE)
      CardKernel_SetCTX(CNULL(CardRTCTX));
    else
    {
      // retrieve crtvcpu supporting taskid
      ...
      this_->curr_crtvcpu=crtvcpu;
      CardCRTVCPU_PreSliceEx( crtvcpu, this->taskid );
    }
}

void _CardCRTVCPUCoordinator_PostSlice(CardCRTVCPUCoordinator *this_)
{
  CardCRTVCPU_PostSliceEx( this_->curr_crtvcpu, this->taskid );

  this_->currslice=CardKErnel_GetSlice( ) + 1;

  if( this_->curr_slice == this_->hyperperiod )
  {
    this_->curr_slice=0;
    CardTB32_Reset( this_->workingmem );
  }
}
```

FIG. 4.8: Activation et gel d'un processeur virtuel collaboratif.

Une extension collaborative, lorsqu'elle reçoit le microprocesseur, le donne au coordinateur à l'aide de son interface `GrantSlice( )` dont le code est donné figure 4.8 (une version complète de cette méthode est donnée en annexe de ce document 5.7 page 119). Le coordinateur va utiliser la politique d'ordonnancement de l'extension élue pour déterminer quelle est la tâche qui doit être exécutée pendant le quantum courant. À partir de son identifiant il peut retrouver l'extension qui la supporte et lui demander de l'activer en invoquant son interface `PreSliceEx( )` (cf 4.8). L'extension qui reçoit finalement le quantum n'a plus qu'à demander l'activation du contexte correspondant à l'exo-noyau. Au bout du temps correspondant à la durée d'un quantum, le processeur virtuel collaboratif correspondant à l'extension qui détient le quantum courant est prévenu par l'exo-noyau de sa future mise en sommeil. Cette extension transmet cette information au coordinateur qui la retransmet à l'extension supportant la tâche qui vient d'être exécutée. Lorsque le coordinateur atteint l'hyperpériode des tâches, il réinitialise la zone de mémoire de travail qu'utilise la politique d'ordonnancement de l'extension élue afin de garantir le déterminisme du plan qu'elle produit.

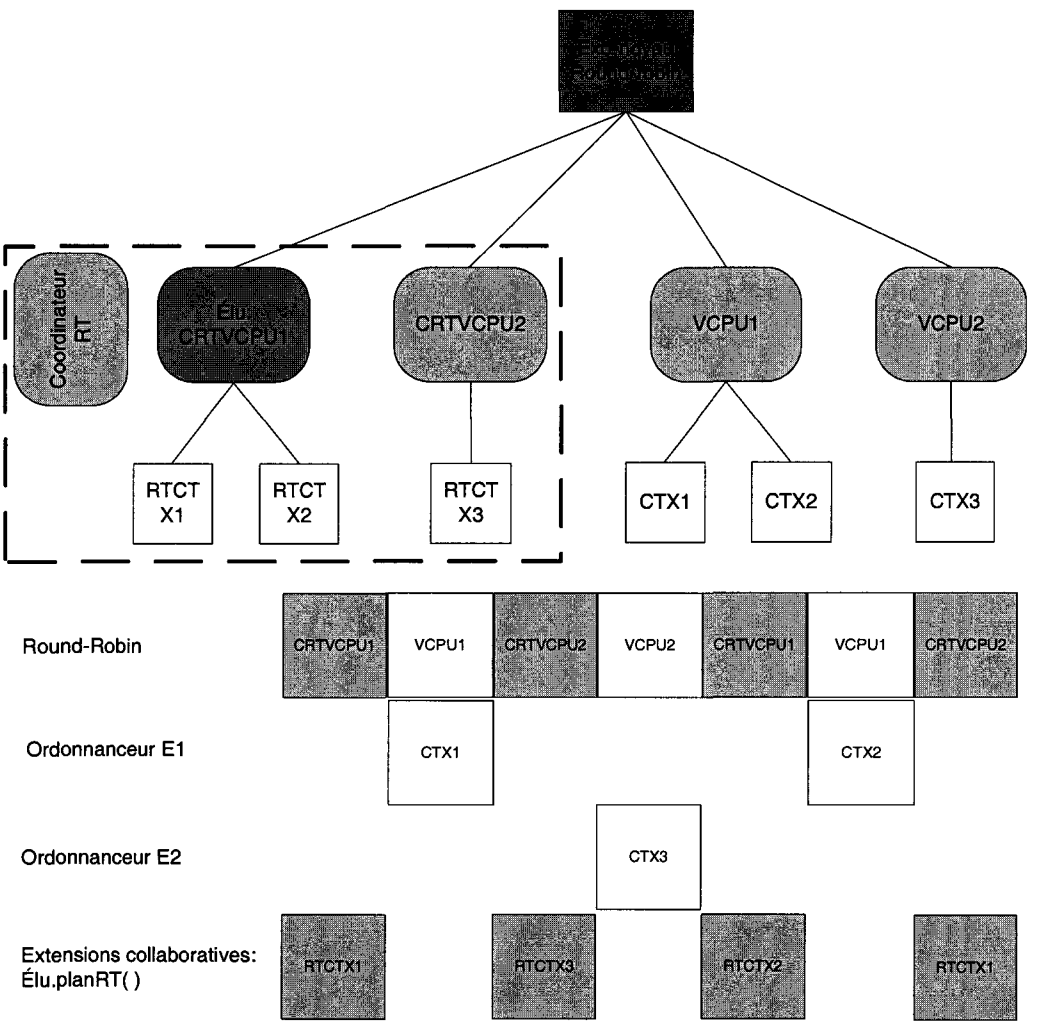


FIG. 4.9: Extensions classiques et collaboratives.

4.4 Synthèse

L'architecture à base d'extensions simples et collaboratives que nous venons de décrire permet aux extensions de faire coexister des applications classiques et des applications ayant des besoins temps réel. Un exemple complet de cette architecture est donné sur la figure 4.9.

1. Une extension, comme l'extension associée au processeur virtuel numéro deux, peut partager sa fraction du microprocesseur entre ses applications classiques en utilisant un simple ordonnanceur *round-robin* implanté en surchargeant les méthodes `PreSlice( )` et `PostSlice( )` de son processeur virtuel.
2. Une extension peut garantir la disponibilité du microprocesseur à ses applications ayant des besoins temps réel. Elle peut les ordonnancer à l'aide de sa propre politique d'ordonnancement temps réel au sein de son processeur virtuel comme le fait l'extension associée au `VCPU1`.
3. L'utilisation de l'exo-noyau comme tiers de confiance permet aux extensions de remplacer la politique initiale (*i.e.* l'ordonnanceur *round-robin*) de partage du microprocesseur entre les extensions.
4. Enfin, des extensions peuvent collaborer et partager à la fois une de leurs politiques d'ordonnancement mais aussi mutualiser leurs accès au microprocesseur afin de mieux servir l'ensemble de leurs applications en optimisant l'accès à la ressource d'exécution.

Remplacer le *round-robin* de l'exo-noyau entraîne un surcoût concernant les temps de déploiement et d'installation d'une nouvelle extension et de son processeur virtuel. Cette opération contribue aussi à la diminution de la partie des quanta de temps utile aux applications. Pour que ce remplacement soit possible, l'exo-noyau doit obligatoirement être capable de calculer le temps d'exécution au pire cas des ordonnanceurs des extensions candidates au remplacement de la politique d'ordonnancement *round-robin* de l'exo-noyau.

La collaboration étendue (partage d'un ordonnanceur temps réel et mise en commun des quanta de temps), augmente encore les temps de déploiement d'un nouveau processeur virtuel collaboratif. De même, la quantité utile sur un quantum de temps se voit aussi diminuée de par la plus grande complexité des opérations servant le partage.

Nous évaluerons, dans le chapitre suivant, les performances, les tailles en mémoire de code, ainsi que les consommations mémoires des différents composants que nous avons mis en place dans notre prototype d'exo-noyau CAMILLERT. Ces composants supportent des opérations que l'ont peut ranger en plusieurs familles :

1. les services élémentaires de l'exo-noyau liés aux contextes d'exécution, aux interruptions et à l'ordonnancement des extensions ;
2. les opérations supportées par les processeurs virtuels ;
3. les opérations des processeurs virtuels collaboratifs ;
4. les services du coordinateur des extensions collaboratives servant à supporter le partage.

Chapitre 5

Évaluation expérimentale de CamilleRT

«L'imagination peut être comparée au rêve d'Adam: à son réveil, c'était devenu la réalité. John Keats, dans une lettre à un ami» – **Dan Simmons, Hyperion 3, Endymion, 1996.**

Tout au long de ce mémoire, nous avons présenté des éléments de solution permettant de garantir et d'optimiser l'accès à la ressource d'exécution. Nous avons mis en place un canevas pour partage sûr d'un service entre plusieurs extensions. Nous avons proposé d'exploiter ce canevas pour que plusieurs extensions supportant des tâches temps réel puissent utiliser une même politique d'ordonnancement chargée dynamiquement par l'une d'elles.

Dans un premier temps, nous présentons le contexte expérimental dans lequel nous avons évalué ces différents travaux. Nous nous intéressons, tout d'abord, à évaluer la faisabilité, en milieu fortement contraint, de l'algorithme de calcul des modes d'accès d'un service à ses arguments. Nous évaluons quel impact sur la chaîne de compilation externe a l'algorithme chargé de calculer une proposition de signature d'un code mobile. Nous nous arrêtons, ensuite, sur les performances de l'algorithme de vérification embarqué des modes d'accès, en mesurant notamment sa consommation mémoire. Nous mesurons ensuite la pertinence des modifications de l'architecture initiale de CAMILLE que nous avons mises en place afin de supporter du temps réel au niveau des extensions de l'exo-noyau. Nous nous intéressons tout d'abord aux composants de l'exo-noyau rendant possible le chargement dynamique d'ordonnancement par les extensions. Nous évaluons ensuite les performances d'une extension temps réel. Nous concluons enfin en mesurant les sur-coûts engendrés par la coopération entre extensions temps réel rendue possible par l'utilisation d'un composant de coordination.

5.1 Contexte d'expérimentation

Nous avons mené nos expérimentations sur la nouvelle version du prototype de l'exo-noyau CAMILLE. À l'inverse du prototype initial entièrement écrit en assembleur [Gri00], cette nouvelle implantation utilise un sous-ensemble du langage C. Nous avons modifié la suite de compilation GCC afin qu'elle puisse traduire un programme écrit en C dans le langage intermédiaire FAÇADE. Le code FAÇADE est ensuite converti à l'aide d'un assembleur dans un format binaire que le système embarqué peut charger, compiler puis exécuter. Cet assembleur est notamment en charge d'intégrer les éléments de preuve permettant la vérification embarquée du typage du programme.

Notre sous-ensemble du langage C impose quelques restrictions au programmeur concernant les régles de typage. Le programmeur doit respecter la hiérarchie de types de CAMILLE et seules les conversions statiques d'une référence vers une de ses super classes sont autorisées. Il n'a pas à sa disposition de librairie C standard, mais peut, en remplacement, exploiter les interfaces des différents composants du noyau qui exposent le matériel. Nous avons ajouté à ce sous-ensemble du langage C les notions liées au modèle objet de CAMILLE (héritage, méthode de classe et d'instance, champ de classe et d'instance ...).

Le nouveau prototype de CAMILLE est entièrement écrit à l'aide de ce sous-ensemble du langage C. Il peut être traduit en FAÇADE à l'aide du compilateur GCC modifié. Ceci confirme que notre sous-ensemble du langage C reste utilisable pour écrire du code système malgré les restrictions qu'il impose.

Nous avons testé l'implantation de notre prototype sur une architecture qui intègre :

- un processeur RISC ARM7TDMI cadencé à 16.78 Mhz ;
- 32Ko de mémoire de travail sur le corps du processeur (IWRAM) ;
- 256Ko de mémoire de travail externe (EWRAM) ;
- 4 compteurs matériels ;
- une ligne série pouvant atteindre un débit de 115000 bauds.

Cette architecture est très proche de ce que l'on peut trouver sur une carte à microprocesseur à l'exception de l'absence de mémoire persistante. Le projet européen CASCADE [EP898] a notamment utilisé un tel processeur dans une carte à puce. Nous avons choisi d'utiliser la mémoire de travail externe pour simuler le fonctionnement de la mémoire persistante normalement présente sur une carte à microprocesseur dans des quantités similaires.

Notre prototype représente un peu moins de 20000 lignes de C et comporte pas moins de 50 composants. Il peut être traduit en 5000 lignes de code FAÇADE. Sa taille actuelle est de l'ordre de 100 ko (dont 40 ko de métadonnées). La différence de taille par rapport au prototype initial, entièrement écrit en assembleur, s'explique de plusieurs façons. Tout d'abord, nous sommes passés d'un processeur 16 bits avec un jeu d'instructions mixé 8-16 à une architecture 32 bits, ceci explique un premier facteur d'expansion de l'ordre de 2. Le premier prototype était écrit

entièrement en assembleur car il s'intéressait à démontrer la faisabilité d'un exo-noyau dans une taille de code acceptable pour les capacités des cartes de l'époque. Les cartes modernes ont une taille de ROM de l'ordre de 128 ko, nous étions intimement convaincus que notre prototype allait posséder une empreinte mémoire inférieure à cette limite. Ainsi, nous avons affiné l'architecture initiale et avons décidé de la concevoir en langage de plus haut niveau afin d'en simplifier le développement. Ceci explique une expansion supplémentaire d'un ordre de 2 concernant la taille du code et des métadonnées qui y sont associées. Nous avons de plus ajouté des fonctionnalités au prototype initial comme par exemple le support du temps réel qui sera évalué dans la suite de ce chapitre. Ainsi, notre prototype est passé d'une taille de 20 ko à une taille de 100 ko. L'expansion la plus marquante reste quand même liée aux métadonnées sur lesquelles nous pouvons gagner en termes d'empreinte mémoire en utilisant des techniques de compression similaires à celles décrites dans l'article [RD04].

5.2 Algorithme de calcul des modes d'accès

5.2.1 Déploiement dans l'architecture de CamilleRT

Nous avons vu précédemment que l'algorithme de calcul de la signature d'une méthode possède une complexité algorithmique qui va bien au delà des capacités d'une carte à microprocesseur. Ainsi, nous avons dû distribuer ce calcul à la manière des codes autocertifiants entre le terminal et la carte (cf section 3.3.3 page 3.3.3). Le terminal possède la puissance de calcul et la quantité de mémoire suffisante pour effectuer le calcul de la signature. La carte, quant à elle, doit être capable de vérifier la concordance entre le code FAÇADE d'un traitement et la signature proposée par le terminal en exploitant les éléments de preuve qui sont fournis avec le code. Le déploiement de l'algorithme de calcul des modes d'accès a donc nécessité des modifications, illustrées sur la figure 5.1, au niveau :

1. de la chaîne de compilation externe ;
2. du processus de compilation embarqué qui traduit le code intermédiaire FAÇADE en code natif.

La chaîne de compilation externe doit tout d'abord être capable de calculer la base de signatures correspondant à l'ensemble des méthodes formant l'exo-noyau CAMILLE qui sont indépendantes du fonctionnement d'un matériel. Nous avons, au préalable, signé manuellement toutes les autres méthodes du noyau dont le comportement est dépendant du fonctionnement d'un matériel. Nous avons modifié notre assembleur afin qu'il puisse calculer une proposition de signature et les éléments de preuve associés à tout code mobile écrit à l'aide du langage intermédiaire FAÇADE.

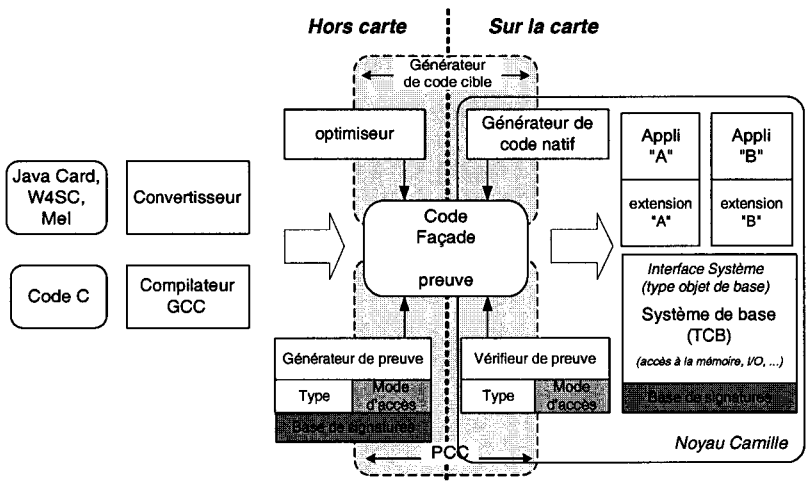


FIG. 5.1: Distribution du calcul des modes d'accès dans CAMILLERT.

5.2.2 Évaluation du générateur de signatures

Nous avons évalué l'algorithme chargé de construire la base de signatures sur l'ensemble des méthodes du noyau CAMILLE (cf section 3.3.2 page 66). Ce cas d'étude comporte un peu moins de 320 méthodes représentant 20000 lignes de code C qui sont traduites en 5000 instructions FAÇADE. Le tableau 5.1 résume les chiffres pertinents extraits de ces sources FAÇADE. La dernière ligne de ce tableau donne les pires cas concernant le nombre d'instructions, de points de saut et de variables.

	Nombre d'instructions	Nombre de Labels	Nombre de Arguments	Nombre de Locales	Nombre de Temporaires
Total (sur le noyau)	3114	393	396	240	289
Moyen (par méthode)	17	2	2	1	1
Pire cas (par méthode)	167	22	9	11	5

TAB. 5.1: Statistiques extraites du code FAÇADE de CAMILLE.

L'algorithme chargé de calculer les signatures des 320 méthodes de l'ensemble des composants formant CAMILLE nécessite 4 itérations successives. Parmi ces 320 méthodes, 120 sont des méthodes dites «natives», c'est-à-dire des méthodes dépendant du fonctionnement d'un matériel qui n'ont pas d'implantation en code FAÇADE et qui sont prises en charge par le mécanisme de liaisons flexibles des composants de CAMILLE [DRG04]. Parmi ces méthodes, on trouve par exemple les opérations arithmétiques et logiques des composants `CardBool`, `CardByte`, `CardShort` et `CardInt`. Ces méthodes sont donc signées à la main par l'expert du système. Elles forment la base de confiance initiale de signatures.

Les 4 itérations de l'algorithme de calcul des signatures sont issues des remises en cause

des signatures calculées aux itérations précédentes ou qui ont été attribuées par défaut[1]. Ce nombre relativement élevé confirme la difficulté d'embarquer l'algorithme de calcul des signatures dans sa totalité en lieu et place de l'algorithme de vérification (cf section 3.3.3 page 69) de la signature proposée par le terminal qui, lui, est linéaire en fonction de la taille du code et qui ne remet pas en cause les signatures déjà calculées.

En étudiant l'ensemble des signatures des méthodes du noyau CAMILLE, nous nous sommes rendu compte que le seul mode d'accès complexe de type $Link_{\alpha}$ détecté était le mode $Link_{\{R\}}$. Ainsi, dans le code de CAMILLE, aucune méthode ne lie un de ses arguments à un de ses autres arguments. L'unique exception à cette règle consiste à renvoyer une valeur ou une référence issue d'un calcul sur un de ses arguments qui se voit donc assigner le mode d'accès $Link_{\{R\}}$. Ceci nous amène à proposer une approximation du modèle initial. Nous proposons de conserver uniquement un état tri-valué pour les arguments ($\bot$, $\top$ ou $Link_{\{R\}}$) et d'associer les autres états en $Link_{\alpha}$ à l'état $\top$. Cette simplification peut nous amener à considérer une méthode comme plus intrusive qu'elle ne l'est en réalité. En contrepartie, elle permet d'économiser de la mémoire pour l'encodage des signatures.

En effet, sans cette simplification nous devions considérer $2 + 2^{19}$ états possibles par argument (16 arguments classiques, le `this`, le pseudo argument `world`, ainsi que le retour de la fonction). La signature d'une méthode est donc encodée à l'aide de 342 bits sans la simplification. Ce chiffre est réduit à 29 bits par méthode avec la simplification en états tri-valués. Chaque méthode du noyau CAMILLE est décrite par une instance du composant `CardCode` regroupant diverses informations comme sa signature de type, sa visibilité, ... Dans un premier temps, nous ajoutons un membre de type `CardInt` au composant `CardCode` afin d'encoder la signature des modes d'accès de la fonction qu'il décrit. La base de signatures s'encode donc à l'aide de 10240 bits, soit 1280 octets sans tenir compte du partage possible des structures de données entre les classes d'une même famille.

Une étude plus fine de la base des signatures a mis en avant que beaucoup de celles-ci étaient identiques. Ceci s'explique tout d'abord par le fait qu'une grande majorité des méthodes ne font rien de dangereux avec leurs arguments et donc peuvent avoir une signature commune avec d'autre méthodes possédant le même nombre d'arguments. Mais aussi par le fait que nous imposons qu'une méthode qui surcharge une autre méthode possède la même signature que l'implantation qu'elle redéfinit. Ainsi, la base se réduit à 31 signatures différentes. En regroupant les signatures dans un composant et en attribuant à chaque méthode un numéro identifiant sa signature, on peut diminuer la quantité de mémoire nécessaire pour son stockage, mais on augmente, par la même occasion, le temps nécessaire pour obtenir la signature d'une méthode car on ajoute une indirection. En changeant le membre de la classe `CardCode` en `CardShort`, on peut référencer 65536 signatures différentes et on diminue la consommation mémoire nécessaire pour encoder les identifiants de signature des 320 mé-

[1] Une méthode appelée par une autre méthode, mais qui n'a pas été encore rencontrée, se voit attribuée une signature par défaut avec tous ses arguments à l'état $\bot$.

thodes du noyau de 1280 à 640 octets de mémoire. Les 31 signatures différentes s'encodant à l'aide de 124 octets. Cette quantité de données est tout à fait acceptable dans le contexte de CAMILLE.

5.2.3 Évaluation du vérifieur embarqué

Nous avons modifié le processus de compilation embarqué afin qu'il soit capable d'exploiter les signatures des méthodes embarquées pour vérifier la proposition de signature de tout code mobile FAÇADE chargé dynamiquement.

Comme nous l'avons vu précédemment, cet algorithme consiste à réaliser une évaluation abstraite linéaire de la méthode en vérifiant la concordance entre le code de la méthode, la signature proposée par le terminal et les éléments de preuve (cf section 3.3.3 page 69).

La preuve est composée des lignes de dépendances aux différents points de saut de la méthode. Chacune de ces lignes encode la dépendance de toutes les variables de la méthode envers ses arguments. Ainsi, une preuve de mode d'accès à une taille de $(|labels| * ((|arguments| + |locales| + |temporaires|) * (|arguments|))$ bits en utilisant un unique bit pour encoder la dépendance d'une variable envers un argument. À ceci doit être ajouté la taille de la signature finale encodée à l'aide de 29 bits, en tenant compte du modèle simplifié.

Le langage intermédiaire FAÇADE impose des limitations concernant les nombres de variables, d'arguments et de points de branchement. La méthode la plus complexe qu'il est possible d'écrire en FAÇADE est limitée à 256 labels et utilise au maximum 512 variables de travail (locales et temporaires). Le nombre d'arguments pour une méthode FAÇADE est limité à 16. Ainsi, en théorie, la plus grande preuve que nous pouvons rencontrer représente environs 300Ko. En pratique cette limite n'est jamais atteinte. Pendant le calcul de la base de signatures de toutes les méthodes du noyau (cf tableau 5.1), la plus grande des preuves que nous avons rencontrée ne dépassait pas 25 octets. Cet ordre de grandeur est tout à fait acceptable pour les capacités de mémoire et de communication d'une carte à microprocesseur.

Un point important concerne la gestion de l'héritage et de la surcharge de méthode. Nous avons proposé précédemment qu'une méthode surchargeant une autre méthode possède la même signature que l'implantation qu'elle redéfinit. Cette première proposition a tendance à imposer trop de restrictions et donc à diminuer fortement le pouvoir d'extensibilité du noyau. En effet, si une classe possède une méthode qui a été signée comme «inoffensive», il n'y aura plus aucun moyen de la surcharger autrement. Pour résoudre ce problème, nous nous proposons d'autoriser une méthode surchargeant une autre méthode à être plus restrictive que la méthode surchargée.

L'algorithme de vérification des modes d'accès a été intégré au processus de chargement de CAMILLE. Ce processus était initialement capable de vérifier et de traduire en code natif les instructions d'un code FAÇADE à la volée, afin de minimiser la consommation mémoire nécessaire au chargement et à la compilation. Au même titre que la complexité du vérifieur

de type [GD01], celle de la phase de vérification des mode d'accès est grandement masquée par les latences des entrées/sorties, mais aussi par les écritures en mémoire persistante[2] des instructions natives générées. La consommation en mémoire de travail du vérifieur des modes d'accès est un peu plus élevée que celle du vérifieur de type, mais au même titre que ce dernier, la vérification des modes d'accès ne nécessite pas de conserver la preuve qui peut donc être effacée une fois la vérification terminée.

5.3 Architecture extensible pour traitements en temps réel

Nous allons, dans cette section, évaluer les différents composants qui ont été mis en place pour partager et garantir l'accès au microprocesseur à toutes les extensions qui le nécessitent. Pour chaque acteur de l'ordonnancement, nous mesurons tout d'abord la taille de son code C compilé pour notre plateforme expérimentale. À titre d'information, nous donnons aussi le nombre de lignes de code C de chacun de ces composants. Le processeur que nous avons utilisé possède deux jeux d'instructions de tailles et de performances différentes. Il possède tout d'abord un jeu d'instructions de taille 32 bits manipulant 3 registres par instructions. Ce jeu d'instructions a été simplifié en une version 16 bits appelée `thumb` ne manipulant que deux registres par instructions. Le mode `thumb` nécessite donc plus d'instructions que le mode `arm` pour effectuer la même opération. Mais, du fait de la taille réduite de ses instructions, il contribue à diminuer la taille des programmes compilés.

5.3.1 Support du temps réel dans le noyau

Commençons tout d'abord par nous intéresser à la couche basse chargée de la virtualisation de l'état du microprocesseur et des contextes d'exécution associés aux traitements. En effet, ces opérations sont à la base de l'activité d'ordonnancement. Ce sont sur elles que les extensions vont reposer pour servir les besoins de leurs applications temps réel.

Composant	lignes de code C	taille en mode `arm`	taille en mode `thumb`
CardCTX	83	248	220
CardKernel	349	1896	1236
CardITManager	284	836	560
CardVCPU (abstrait)	120	124	84

TAB. 5.2: Taille des composants supportant le temps réel.

Le tableau 5.2 résume les tailles des trois composants supportant la base de l'ordonnancement dans CAMILLERT. Nous trouvons tout d'abord le composant `CardCTX` virtualisant le contexte d'exécution d'un traitement. Ce composant a une taille très faible car il est utilisé

[2]Dans une carte à microprocesseur, les temps d'écriture en mémoire persistante sont dix mille fois plus élevés qu'en mémoire volatile.

principalement en tant que structure de données. Il contient la pile d'exécution du traitement auquel il est associé, ainsi que quelques attributs permettant sa réactivation et son gel. Nous avons isolé du composant `CardKernel` tous les traitements relatifs à la gestion des contextes d'exécution, du partage du microprocesseur entre les extensions et de l'ajout et du retrait d'une extension. Cette partie de `CardKernel` possède une taille relativement élevée qui s'explique par le fait qu'il regroupe la majorité des opérations qui sont à la base de notre architecture extensible pour applications temps réel. Le dernier composant de cette couche basse est en charge du démultiplexage des interruptions du matériel vers les extensions. Il comporte notamment les opérations nécessaires au partage de la ligne de communication série entre les extensions. Chaque extension possède une instance d'un composant matérialisant la ligne série démultiplexée. Elle peut enregistrer cette instance auprès du gestionnaire d'interruptions qui lui délivrera les octets entrant sur le modèle décrit dans le chapitre précédent. À ces composant s'ajoute une instance abstraite de processeur virtuel, c'est-à-dire sans implantation des méthodes `PreSlice( )` et `PostSlice( )`. Cette couche élémentaire nécessite un total de 3 ko de mémoire de code en mode `arm` qu'il faut comparer aux 100 ko du noyau complet.

Service	nombre de cycles
Activation d'un contexte	128
Ordonnanceur *round-robin*	443
Gel d'un contexte	120

TAB. 5.3: Temps d'exécution du support du temps réel.

Intéressons nous maintenant aux temps d'exécutions des différentes opérations de notre couche basse de support du temps réel. En effet, les performances de l'acte d'ordonnancement des applications au niveau des extensions de l'exo-noyau dépendent directement des performances de ces opérations. Pour ce faire, nous avons mesuré les temps d'exécution en nombre de cycles de ces opérations compilées en code 32 bits `arm`. Le tableau 5.3 reprend les résultats obtenus. Nous pouvons voir que l'activation et le gel d'un contexte d'exécution sont des opérations très rapides; elles consistent à charger ou à écrire le contenu des registres du microprocesseur depuis la pile d'exécution du traitement. Ceci confirme la pertinence d'avoir un contexte d'exécution très proche du microprocesseur et de sa pile d'exécution. L'ordonnanceur d'extensions de type *round-robin* de l'exo-noyau à un temps d'exécution acceptable. De plus, il est indépendant du nombre d'extensions.

5.3.2 Processeurs virtuels

Nous avons implanté deux processeurs virtuels différents à partir des définitions que nous avons proposées dans le chapitre précédent. Le premier ne gère qu'un seul et unique traitement, le second est une sous classe du premier qui supporte plusieurs traitements ordonnancés à l'aide d'un simple *round-robin*. Ces deux types de processeurs virtuels possèdent une empreinte mémoire très faible comme illustré par le tableau 5.4. Ceci s'explique par le fait qu'ils ne font qu'exploiter les opérations des couches basses de l'exo-noyau pour partager une partie du microprocesseur entre leurs applications.

Composant	lignes de code C	taille en mode `arm`	taille en mode `thumb`
CardVCPUS	130	172	108
CardVCPUM	137	320	184

TAB. 5.4: Tailles des processeurs virtuels.

Service	nombre de cycles
`CardVCPUS::PreSlice( )`	160
`CardVCPUS::PostSlice( )`	140
`CardVCPUM::PreSlice( )`	320
`CardVCPUM::PostSlice( )`	320
Total pour ordonnancer un contexte	1336

TAB. 5.5: Temps d'exécution des activations et gel des processeurs virtuels.

Nous avons ensuite mesuré les temps d'exécution correspondant à la réception des évènements de début et de fin du quantum au niveau des processeurs virtuels. Le tableau 5.5 reprend les résultats obtenus. Nous disposons donc de toutes les informations permettant de quantifier le coût total des opérations de l'exo-noyau et du processeur virtuel sur un cycle complet d'ordonnancement, c'est-à-dire :

1. choix d'un processeur virtuel par l'exo-noyau ;
2. appel de sa méthode `PreSlice( )` ;
3. choix d'une tâche par l'ordonnanceur de l'extension ;
4. demande d'activation de cette tâche ;
5. gel de la tâche à la fin du quantum.

Le coût total est de 1336 cycles en utilisant une instance de `CardVCPUM` ce qui représente une augmentation de 50% par rapport à un simple ordonnanceur au niveau de l'exo-noyau en se basant sur les temps d'exécutions de la couche précédente et en supposant qu'une politique d'ordonnancement temps réel au niveau de l'exo-noyau aurait un temps d'exécution du même ordre que le *round-robin* que nous avons évalué.

Nous avons vu, dans le chapitre précédent, que les extensions pouvaient jouer sur la taille du quantum de temps élémentaire pour agir sur la réactivité aux interruptions matérielles.

taille en ms	taille en cyles	cycles utiles	pourcentage utile
5	83904	82568	98.40
10	167808	166472	99.20
20	335616	334280	99.60
40	671232	669987	99.80
100	1678080	1676744	99.92

TAB. 5.6: Pourcentages du quantum de temps utiles aux applications.

À partir du nombre total de cycles consommés par l'exo-noyau et par l'ordonnanceur des extensions, nous pouvons mesurer la quantité du quantum de temps qui reste à la disposition des applications. Le tableau 5.6 résume les pourcentages que nous avons obtenus pour différentes tailles de quantum de temps. Au pire cas, avec un quantum de 5 millisecondes, le support de l'ordonnancement coûte 1.60%. Avec un quantum de 20 millisecondes qui est une taille de quantum courante dans un système temps réel embarqué, 19.92 millisecondes sont directement utilisables par l'application. Pour des tailles plus grandes du quantum de temps, la présence de l'exo-noyau et du processeur virtuel devient de plus en plus invisible.

5.3.3 Coordinateur temps réel et processeurs virtuels collaboratifs

Nous avons vu précédemment que la collaboration entre extensions est une solution au problème de l'ordonnancement aveugle (cf section 2.2.2 page 42). De plus, elle permet aussi d'optimiser l'usage de la ressource d'exécution en se débarrassant des problèmes engendrés par l'isolation naturelle qui existe entre les extensions. Toutefois, pour que cette collaboration soit viable, nous avons proposé d'introduire un composant de coordination de ces extensions temps réel chargé de rétablir la confiance mutuelle entre extensions. Intéressons nous maintenant aux différents coûts engendrés par la coopération entre extensions.

Composant	lignes de code C	taille en mode `arm`	taille en mode `thumb`
CardRTCTX	67	84	68
CardCRTVCPU	194	708	432
CardCRTVCPUCoordinator	386	2012	1264

TAB. 5.7: Taille des composants supportant la coopération.

Nous avons tout d'abord implanté une sous classe de contexte d'exécution en lui ajoutant les attributs temps réel classiques. Ce composant à une taille très faible (comme illustré à l'aide du tableau 5.7) car il hérite de la quasi totalité des opérations du contexte d'exécution standard. Un processeur collaboratif possède une empreinte mémoire deux fois plus élevée qu'un processeur virtuel de type `CardVCPUM` mais sa taille reste acceptable. La différence de taille s'explique par la politique d'ordonnancement temps réel et par le support de la coopération qui sont présents au niveau des processeurs virtuels collaboratifs. Le composant

que nous avons implanté est conforme à la description faite dans le chapitre précédent. Il est doté d'une politique d'ordonnancement de type *Least Laxity First* possédant la signature de modes d'accès suivante : (*world* : $\perp$, *this* : $\perp$, *nb_task* : $\perp$, *task_list* : $\perp$, *curr_slice* : $\perp$, *hyperperiod* : $\perp$, *cmem* : $\top$). Le code de cette implantation est disponible en annexe de ce document (cf 5.6 page 118). Le gros morceau de la coopération se trouve dans le composant coordinateur comme illustré par sa taille relativement élevée. Le support de la collaboration entre extensions a donc un coût en termes d'empreinte mémoire que l'on peut qualifier de non négligeable. Sa taille s'explique par le fait qu'il effectue les tests d'admission d'une extension collaborative.

Service	nombre de cycles
`PreSlice( )`	2944
`PostSlice( )`	832
Total pour ordonnancer un contexte	4472

TAB. 5.8: Temps d'exécution des activations et gels des processeurs virtuels collaboratifs.

Pour mesurer l'impact de la coopération sur les performances à l'exécution, nous avons tout d'abord mesuré les temps d'exécutions des méthodes `PreSlice( )` et `PostSlice( )` de nos processeurs virtuels collaboratifs. Rappelons que ce sont ces méthodes qui donnent le quantum courant au coordinateur. Celui-ci appelle la politique d'ordonnancement qui a été élue lors du dernier ajout d'une extension collaborative, puis demande à l'extension qui supporte la tâche choisie de l'activer. Le coût total d'un cycle complet d'ordonnancement est ici de l'ordre de 4500 cycles du microprocesseur ce qui est 3.3 fois plus élevé que celui obtenu au niveau des simples processeurs virtuels.

taille en ms	taille en cyles	cycles utiles	pourcentage utile
5	83904	78096	93.07
10	167808	162000	96.53
20	335616	329808	98.26
40	671232	665515	99.13
100	1678080	1672272	99.65

TAB. 5.9: Pourcentages du quantum de temps utiles aux applications.

Nous avons, de même, évalué l'impact de la taille du quantum de temps sur le pourcentage du quantum restant à la disposition des applications. Le tableau 5.9 reprend les résultats obtenus. On observe bien que pour des petits quanta, l'impact de la coopération est plus forte qu'au niveau des processeurs virtuels. Toutefois, pour des quanta de temps de taille supérieure à 20 millisecondes, le coûts des différents niveaux d'ordonnancement repasse en dessous de la barre des 2%.

La courbe de la figure 5.2 résume les coûts de l'ordonnancement suivant la taille du quantum de temps pour des extensions isolés et des extensions collaboratives. On voit bien

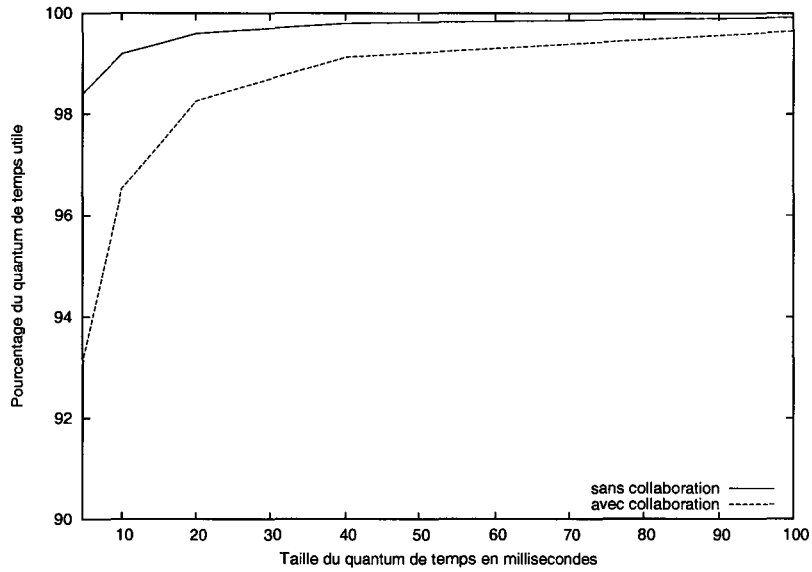


FIG. 5.2: Comparaison du coût de l'ordonnancement avec et sans collaboration.

les différences de performances avec et sans coopération de la part des extensions. De même, comme nous l'avons vu précédemment, la coopération entre extensions a aussi un coût en termes de taille de code concernant le support à la coordination. Toutefois, les coûts de la collaboration sont à mettre en balance avec les gains notamment concernant la capacité à ordonnancer des jeux de tâches là ou un système isolé n'aurait pas pu le faire (cf section 2.2.2 page 42). De plus la collaboration entre les extensions leur permet de ne pas toutes implanter de politique d'ordonnancement. Ainsi, la collaboration permet de factoriser du code et donc d'économiser de la mémoire de code des extensions.

Bilan et perspectives

«Toutes choses étant égales par ailleurs, la solution la plus simple est généralement la bonne.» – **Guillaume d'Occam - Le rasoir d'Occam, XVI siècle.**

Il est maintenant temps de conclure ces trois années de thèse autour de la problématique des architectures extensibles pour applications temps réel. Dans un premier temps, je donnerai une synthèse de mes apports. Ensuite, je listerai quelques enseignements que l'on peut extraire de mes travaux. Enfin, je m'intéresserai à donner quelques voies de recherche possibles, à court et à plus long terme.

Synthèse

Dans ce document, nous avons présenté des solutions au problème du temps réel dans un système d'exploitation extensible. Nous nous ne sommes pas intéressés à proposer un $n^{ième}$ ordonnanceur temps réel pour exo-noyau mais plutôt à définir l'ensemble minimal d'opérations permettant à une extension chargée dynamiquement de garantir l'accès à une portion du microprocesseur à ses applications. Ainsi, l'extensibilité de l'exo-noyau est conservée car il ne propose pas d'abstraction de haut niveau concernant le support du temps réel mais offre la capacité aux extensions de les construire.

Nous avons tout d'abord mis en avant les faiblesses des solutions initialement proposées pour supporter du temps réel dans un exo-noyau. Nous nous sommes en particulier arrêté sur l'isolation que met en place un exo-noyau entre ses extensions. Cette isolation, même si elle favorise la sécurité, introduit une duplication des données et des traitements au sein des extensions faute de confiance mutuelle. De plus, elle contribue à une mauvaise utilisation des ressources, ce qui dans le cas d'un système classique peut être acceptable, mais n'est pas envisageable dans le contexte de l'informatique embarquée sur des supports contraints. Nous avons illustré cette mauvaise utilisation des ressources à l'aide d'un exemple simple constitué de deux extensions supportant des tâches temps réel. Dans ce cas, l'isolation entre extension devient nuisible, car l'exo-noyau se retrouve dans l'incapacité de servir les besoins de ses extensions. Ce problème est appelé problème de l'ordonnancement aveugle.

Nous avons montré que le partage et la collaboration entre extensions sont des solutions permettant d'optimiser l'usage des ressources. Toutefois, pour qu'une extension accepte d'utiliser l'implantation d'un service fourni par une autre extension, elle doit lui faire confiance. Nous avons donc proposé un canevas permettant le partage sûr de services entre plusieurs extensions. Au travers de celui-ci, une extension peut tester le bon fonctionnement d'un service d'une autre extension, puis par la suite, l'utiliser en toute confiance pour résoudre un problème donné ou gérer des ressources. Ce canevas exploite un test exhaustif du service partagé dans une configuration similaire à celle dans laquelle il sera utilisé. Ce test repose sur une phase d'analyse statique du code du service, au moment de son chargement dans le système. Cette analyse permet la détection des états internes du services. L'extension utilisatrice va pouvoir réinitialiser les états internes du service partagé afin qu'il se comporte identiquement à la phase de test pendant chaque exécution. Le mécanisme de détection des états internes a été prouvé par Yann HODIQUE et Isabelle SIMPLOT-RYL, et c'est sur la base de cette preuve que repose finalement toute la sécurité de nos travaux.

Notre démarche initiale était de proposer une architecture permettant le support d'applications temps réel au sein d'un exo-noyau extensible. Nous avons choisi de ne mettre que le support minimal à l'acte d'ordonnancement dans l'exo-noyau afin de permettre aux extensions de charger dynamiquement leur propre politique d'ordonnancement. L'exo-noyau fait office de garant de l'accès à la ressource d'exécution et il est chargé de remonter les interruptions du matériel aux extensions. Nous avons proposé un moyen permettant aux extensions de remplacer la politique *round-robin* de l'exo-noyau partageant le microprocesseur entre les différentes extensions. Cette première architecture a fait preuve de bonnes performances. Elle represente moins de 3 ko de code sur notre plateforme expérimentale et consomme 1336 cycles pour choisir et activer le contexte d'exécution d'une application. Nous avons testé cette couche minimaliste d'exposition et de partage du microprocesseur avec des quanta de temps de différentes tailles. L'acte d'ordonnancement de cette couche represente dans le pire des cas 1.60% du quantum de temps, laissant les 98.40% restant à la disposition des applications.

Nous avons aussi proposé une architecture mettant plus en avant la coopération entre les extensions. Celle-ci repose sur un composant de coordination permettant à un ensemble d'extensions de partager une politique d'ordonnancement et de mettre en commun leurs accès au microprocesseur pour mieux servir les besoins de leurs applications. Ce composant rétablit la confiance entre les extensions qu'il fédère, en testant l'admission d'une extension au sein du groupe. Cette collaboration est rendu possible par utilisation du canevas de partage sûr de services. La collaboration à un coût, en termes de performance d'exécution, 3.3 fois plus élevé que la couche minimaliste d'exposition et de partage du microprocesseur. Au pire cas, l'ordonnancement collaboratif consomme 7% du quantum de temps. Toutefois, pour des quanta de temps de taille supérieure à 20 millisecondes, le coût de l'ordonnancement collaboratif redescend en dessous de la barre des 2%.

Par dela ces résultats purement quantitatifs, nous pouvons extraire certains enseignements des travaux menés atours des architectures extensibles pour applications temps réel, notamment du prototype que nous avons réalisé.

Enseignements

1. **Il faut choisir avec soin la limite entre ce qui est du domaine de l'exo-noyau et ce qui est du domaine des extensions.** En effet, comme nous l'avons expliqué dans les chapitres précédents, nous aurions pu mettre un ordonnanceur temps réel au plus bas niveau de l'exo-noyau. Ce faisant, nous aurions été à l'encontre direct des principes architecturaux d'un exo-noyau qui sont de ne pas proposer d'abstractions de haut niveau dans l'exo-noyau [Eng98, EK95]. Aussi, nous avons choisi de définir l'ensemble minimal d'opérations et de composants nécessaires pour supporter du temps réel au niveau des extensions de l'exo-noyau. Ainsi, l'extensibilité de l'exo-noyau est conservée, mais comme notre prototype a pu le mettre en avant, cette extensibilité entraîne un surcoût de l'ordre de 50% concernant les performances à l'exécution de l'acte d'ordonnancement d'une tâche. De plus, elle nécessite d'être capable de calculer le temps d'exécution au pire cas de certaines implantations d'un service.

2. **Il est plus simple de vérifier une garantie non fonctionnelle qu'une garantie fonctionnelle.** Nous avons pu voir que vérifier le bon fonctionnement d'une politique d'ordonnancement est une chose complexe à mettre en œuvre. Notre proposition exploite une analyse statique du code du service au moment de son installation dans le système, un test en situation par l'extension qui veut l'utiliser et un contrôle à l'exécution afin de réinitialiser les états internes du service pour qu'il reste déterministe. La détection des états internes d'un code mobile est notamment plus coûteuse en terme de taille de preuve qu'une simple vérification de typage.

3. **La coopération des extensions permet une gestion plus efficace du microprocesseur, mais elle engendre des surcoûts non négligeables qui sont issus de la méfiance entre extensions.** Faire coopérer deux extensions dans un contexte de chargement dynamique de code est rendu complexe par l'absence de confiance mutuelle. Il est plus simple pour une extension de faire confiance à un service de l'exo-noyau, qu'à une autre extension dont l'origine n'est pas certifiée et qui peut donc ne pas appartenir à la base de confiance. Nos expérimentations ont mis en avant un surcoût de l'acte d'ordonnancement 3.3 fois plus élevé que celui au niveau des extensions non collaboratives.

Perspectives

Comme le disais Gilles GRIMAUD à la fin de sa thèse, l'architecture de CAMILLE reste un vaste champ de recherche. Nous avons tenté de répondre au problème du support du temps réel dans des architectures extensibles, mais il reste beaucoup de recherches possibles à venir.

La première de ces voies de recherche est déjà initiée dans le cadre des travaux de thèse de Nadia BEL-HADJ-AISSA. Leurs buts est de fournir une réponse au problème suivante :

- Comment distribuer le calcul des temps d'exécution au pire cas d'un code exprimé à l'aide du langage intermédiaire FAÇADE, de tel sorte que la carte à microprocesseur ait la capacité de finaliser le calcul du WCET, sans que trop d'informations ne sortent de la carte vers le terminal ?

En effet, donner des informations concernant les temps d'exécutions de certaines opérations effectuées par la carte soulève des problèmes de sécurité majeurs. Elles peuvent permettre d'attaquer les traitements cryptographiques de la carte en vue de l'extraction d'information concernant les données secrètes que celle-ci manipule. Le but est donc de proposer une distribution à la fois efficace et sécurisée du calcul des temps d'exécution au pire cas sur du code mobile FAÇADE [ARDG04, ARG04].

Nous avons vu que prouver le bon fonctionnement d'un ordonnanceur chargé dynamiquement en vue de le partager est une tâche complexe et coûteuse. Ceci vient surtout du fait que nous avons choisi de ne pas imposer de contraintes concernant la manière dont est écrite la politique d'ordonnancement. On pourrait chercher à définir un DSL adapté à l'écriture de politique d'ordonnancement (comme par exemple celui de BOSSA), qui contribuerait à simplifier la procédure de vérification du bon fonctionnement de la politique d'ordonnancement, ou à l'extraction d'informations sur son fonctionnement. Une telle approche contribuerait forcément à rendre plus complexe le processus de traduction de la politique en code natif nécessaire pour des raisons de performances d'exécution. Peut-on trouver un juste équilibre entre l'extensibilité du système, la simplification de la vérification du bon fonctionnement d'un service et sa transformation en code effectif ?

Une autre piste de recherche concerne la gestion d'autre modèle de tâche. Peut-on étendre notre canevas de support du temps réel afin de gérer des familles de tâches autres que les tâches périodiques ? En effet, des tâches comme les tâches apériodiques ou sporadiques remettent en cause la capacité du coordinateur à construire le domaine de test de la politique d'ordonnancement. Ces tâches peuvent arriver à n'importe quel moment de la vie du système. On connait au mieux leur pseudo période qui est le temps minimale entre deux arrivées potentielles de l'événement correspondant à l'activation de la tâche.

Enfin, à plus long terme, on pourrait s'intéresser sur la capacité de notre architecture à s'adapter à des propriétés autres que le temps. Peut-on l'étendre pour supporter des ordonnanceurs se focalisant sur des notions comme la minimisation de la consommation d'énergie ou de la consommation mémoire ? La gestion de l'énergie pourrait notamment tirer partie des

travaux sur la distribution du calcul des temps d'exécution au pire cas. Elle pourrait peut-être aboutir sur la définition d'un algorithme générique aux deux calculs. Pourrait-on enfin faire cohabiter ces différents modèles d'ordonnancement ?

Annexes

Calcul de la signature des modes d'accès

```
method::Sign()
{
  stack.push(entry_point);
  while(! stack.empty())
  {
    line = stack.pop();
    switch(instructions[line].type())
    {
      case Jump(target): // << est l'operateur d'injection
        if(deps[target] << deps[line])
          stack.pushIfNotIn(target);
      break;

      case JumpIf(target):
        if(deps[target] << deps[line])
          stack.pushIfNotIn(target);
        if(deps[line+1] << deps[line])
          stack.pushIfNotIn(line+1);
      break;

      case JumpList(targets):
        foreach i in targets
          if(deps[i] << deps[line])
            stack.pushIfNotIn(i);
      break;

      case Invoke(m):
        if(deps[line+1] << m.applyOn(deps[line]))
          stack.pushIfNotIn(line+1);
      break;

      case Return(ret): // args is the arguments' array
        foreach i in args do
        { // |= is an accumulative bitwise "or"
          if(deps[line][i].contains(ret))
            sign[i] |= LINK_R;
          foreach j in args do
            if(deps[line][i].contains(args[j]))
              sign[i] |= LINK_j;
        }
      break;
}}}
```

FIG. 5.3: Pseudo code de l'algorithme de signature d'une méthode.

```
m::applyOn(deps[line])
{ // receiver est la variable recevant le retour de fonction
  d = deps[line].remove(receiver);

  foreach i in m.nbArgs do
  {
   switch(m.sign()[i])
   {
     case BOT: break;

     case TOP:
        foreach j in args.size do
          if(deps[line][j].contains(m.args[i]))
            sign[j] = TOP;
     break;

     default:
       if(LINK_R)
         foreach j in args.size do
           if(deps[line][j].contains(m.args[i]))
             tmp.push(j);
       ...
       if(LINK_n)
       {
         if(m.args[n] == receiver) break;

         foreach j in args.size() do
           if(deps[line][j].contains(m.args[i]))
             d[j].push(m.args[n]);
       }
       ...
   }
  }
  foreach i in tmp.size() do
    d[tmp[i]].push(receiver);

  return d;
}
```

FIG. 5.4: Pseudo code de l'algorithme de propagation des modes d'accès sur une ligne de dépendances.

Vérification de la signature des modes d'accès

```
bool verify(method,sign)
{
  dependencies_vector dep = initDeps();

  foreach instr in method.body() do
  {
    if(instr.isLabelled()) // instr is the target of a branch
    {
      if( !instr.proof.includes(dep) )
         Error;
      dep = instr.proof;
    }

    switch(instr)
    {
      case Jump:
      case JumpIf:
      case JumpList:
         foreach target in instr.targets do
         {
           if( ! target.proof.includes(dep) )
             Error;
         }
      break;

      case Invoke(m):
        dep = m.applyAndCheckOn( dep );
      break;

      case Return(ret):
        foreach i in args do
        {
          if(deps[line][i].contains(ret))
            if( ! sign[i].contains(LINK_R) )
              Error;

          foreach j in args do
            if(deps[line][i].contains(args[j]))
              if( ! sign[i].containes(LINK_j) )
                Error;
        }
      break;
    }
  }
}
```

FIG. 5.5: Pseudo code de l'algorithme de vérification embarqué de la signature d'une méthode.

Ordonnanceur collaboratif

```
CardInt _CardCRTVCPU_PlanGLLF( CardCRTVCPU *this_ , CardTOMem256 *ctx_list,
                               CardInt nb_task, CardInt curr_slice,
                               CardTBMem32 *green_square )
{
  CardRTCTX *ctx;
  CardByte curr_ctx, progress, laxity;
  CardInt best_laxity;
  CardByte best_ctx;

  CardInt a,b;

  /* green square contains the progression of the task: CardByte progress */
  best_ctx = TASK_IDLE ;
  best_laxity = 0xFFFFFFFF ; /* max laxity */

  /* scan all tasks */
  for(curr_ctx = 0; curr_ctx < nb_task; curr_ctx++)
  {
    ctx = CardObject_DynamicCast( CardRTCTX,
                                  CardTOMem256_Get( ctx_list, curr_ctx ) );

    progress = CardTBMem32_GetByte( green_square, curr_ctx );
    laxity = ctx->deadline - (ctx->execution_time - progress);

    if ( progress < ctx->execution_time )
    {
      if(laxity < best_laxity )
      {
        /* best */
        best_laxity = laxity;
        best_ctx = curr_ctx;
      }
    }
    else
    {
      if ( (curr_slice+1) % ctx->period == 0)
         CardTBMem32_SetByte( green_square, curr_ctx, 0 );

    }

    if( best_ctx != TASK_IDLE )
    {
      /* update task progression */
      progress = CardTBMem32_GetByte( green_square, best_ctx );
      CardTBMem32_SetByte( green_square, best_ctx, progress + 1 );
    }

    return best_ctx;
}
```

FIG. 5.6: Implantation d'un ordonnanceur *Least Laxity First* dans un processeur virtuel collaboratif.

```
void _CardCRTVCPUCoordinator_Grant(CardCRTVCPUCoordinator *this_)
{
  CardInt nvcpu;
  CardInt cpt;
  CardByte    taskid;
  CardCRTVCPU *crtvcpu;

  /* call choosen crtvpu plan to retrieve elected task number */
  crtvcpu= this_->choosen_vcpu_ptr;

  /* LLF Plan with Green Square */
  taskid=CardCRTVCPU_PlanGLLF( crtvcpu, this_->ctx_list, this_->nb_ctx,
                               this_->curr_slice, this_->choosen_workingmem);

  cpt = 0;

  if (taskid != TASK_IDLE)
  {

    for ( nvcpu=0; nvcpu < this_->nb_crtvcpu; nvcpu++ )
    {
      crtvcpu=CardObject_DynamicCast( CardCRTVCPU,
                      CardTOMem256_Get( this_->crtvcpu_list, nvcpu ) );

      if ( cpt+crtvcpu->num > taskid )
         break;

      cpt += crtvcpu->num;
    }

    taskid -= cpt;

    CardCRTVCPU_PreSliceEx( crtvcpu, taskid );
  }
  else
  {
    CardKernel_SetCTX(CNULL(CardRTCTX));
  }
}
```

FIG. 5.7: Selection d'une tâche par le coordinateur.

Table des matières

1 Systèmes embarqués pour le temps réel : objectifs et contraintes 1

1.1 L'informatique embarquée 1

1.1.1 Matériels de l'informatique enfouie 2

1.1.2 Exemple de la carte à puce 4

1.1.3 Systèmes d'exploitation enfouis 6

1.1.4 Extensibilité des systèmes embarqués 9

1.1.5 Sûreté de fonctionnement des systèmes enfouis 12

1.2 Le temps réel 14

1.2.1 La maîtrise du temps : motivations 15

1.2.2 La maîtrise du temps : mise en œuvre 17

1.2.3 Principes des technologies temps réel 20

1.2.4 Stratégies d'ordonnancement 22

1.2.5 Temps réel pour systèmes enfouis 25

1.3 Architectures extensibles pour le temps réel 27

1.3.1 Ordonnancement temps réel pour systèmes ouverts 27

1.3.2 Les ordonnanceurs hiérarchiques 28

1.3.3 Expression des politiques d'ordonnancement 30

2 Spécification d'un système extensible pour les applications en temps réel 33

2.1 Besoins temps réel pour systèmes embarqués ouverts 33

2.1.1 Réactivité du système 34

2.1.2 Exemple de la carte à puce 36

2.1.3 Sûreté de fonctionnement des systèmes ouverts 38

2.2 Les points clefs du support temps réel dans un exo-noyau 39

2.2.1 Quantifier les temps d'exécution de l'exo-noyau 40

2.2.2 Trouver un ordonnancement satisfaisant l'ensemble des tâches 42

2.2.3 Fiabiliser l'exo-ordonnancement 44

2.3 Stratégies pour support temps réel dans un exo-noyau 46

2.3.1 Maîtrise des temps d'exécution du noyau 47

2.3.2 Supporter des extensions «temps réel» 48
2.3.3 Au delà de l'isolation des extensions 50

3 Canevas pour le partage sûr de services dans un noyau 53
3.1 Extensibilité et factorisation 53
3.1.1 Isolation des extensions 54
3.1.2 Duplication au sein des extensions 55
3.1.3 Sécurité et partage des extensions 57
3.2 Validation de services étendus 59
3.2.1 Vérifier des garanties non fonctionnelles 59
3.2.2 Garanties fonctionnelles 61
3.2.3 Principe du «test au déploiement, contrôle à l'exécution» 62
3.3 Formalisation de la notion de déterminisme 64
3.3.1 États internes et FAÇADE 65
3.3.2 Détecter les états internes d'un code mobile 66
3.3.3 Vers une preuve d'inférence du déterminisme 69
3.3.4 Vérifier le bon fonctionnement d'un service partagé 71

4 Exo-noyau pour applications en temps réel 75
4.1 Exposition du microprocesseur dans CAMILLE 76
4.1.1 Notion de processeur virtuel 76
4.1.2 Contexte d'exécution 78
4.1.3 Démultiplexage des interruptions 80
4.2 Temps réel et processeurs virtuels 82
4.2.1 Garanties d'accès à la ressource d'exécution 82
4.2.2 Maîtrise des retards sur les réceptions des interruptions 84
4.3 Ordonnancement collaboratif des extensions 86
4.3.1 Test d'admission générique 86
4.3.2 Support du partage dans les processeurs virtuels 89
4.3.3 Mise en œuvre sur une hiérarchie d'ordonnanceur temps réel 91
4.4 Synthèse 95

5 Évaluation expérimentale de CamilleRT 97
5.1 Contexte d'expérimentation 98
5.2 Algorithme de calcul des modes d'accès 99
5.2.1 Déploiement dans l'architecture de CAMILLERT 99
5.2.2 Évaluation du générateur de signatures 100
5.2.3 Évaluation du vérifieur embarqué 102
5.3 Architecture extensible pour traitements en temps réel 103

5.3.1 Support du temps réel dans le noyau 103
5.3.2 Processeurs virtuels 105
5.3.3 Coordinateur temps réel et processeurs virtuels collaboratifs 106

Bilan et perspectives **109**

Table des figures

1.1 Cycle de vie et de production du logiciel embarqué. 6
1.2 Modèle d'évolution des systèmes embarqués. 7
1.3 Architecture logicielle de CAMILLE. 10
1.4 Échantillonage de la ligne série. 15
1.5 Code FAÇADE du traitement d'un bit de communication. 16
1.6 Application de l'IPET sur un exemple simple. 18
1.7 Exemple d'arbre syntaxique d'un programme. 20
1.8 Exemple de tâche périodique. 21
1.9 Ordonnanceur hiérachique à la UNIX. 29
1.10 Exemple de traitement de l'évènement unblock d'une politique RM. 31

2.1 Système temps réel industriel. 34
2.2 Modéle d'exécution et de communication terminal / carte. 36
2.3 Demultiplexage du matériel. 41
2.4 Démultiplexage du microprocesseur. 43
2.5 Ordonnancement aveugle. 44
2.6 Collaboration et confiance. 46

3.1 Demultiplexage du matériel. 55
3.2 Duplication des services. 56
3.3 Schéma de confiance. 57
3.4 Partage de service entre extensions. 58
3.5 Principe du code auto-certifié. 60
3.6 Importance des garanties fonctionnelles. 61
3.7 Les deux phases d'utilisation d'un service partagé. 62
3.8 Service possédant un état interne. 63
3.9 Fiabilisation du partage de service entre extensions. 72

4.1 Couche basse d'ordonnancement de CAMILLERT. 77
4.2 Démultiplexage du microprocesseur. 78
4.3 Processeurs virtuels et contextes d'exécutions. 80

4.4 Notification des interruptions. 81
4.5 Retard sur les notifications des interruptions. 85
4.6 Test d'admission générique. 88
4.7 Choix du plan d'ordonnancement. 90
4.8 Activation et gel d'un processeur virtuel collaboratif. 93
4.9 Extensions classiques et collaboratives. 94

5.1 Distribution du calcul des modes d'accès dans CAMILLERT. 100
5.2 Comparaison du coût de l'ordonnancement avec et sans collaboration. 108

5.3 Pseudo code de l'algorithme de signature d'une méthode. 115
5.4 Pseudo code de l'algorithme de propagation des modes d'accès sur une ligne de dépendances. 116
5.5 Pseudo code de l'algorithme de vérification embarqué de la signature d'une méthode. 117
5.6 Implantation d'un ordonnanceur *Least Laxity First* dans un processeur virtuel collaboratif. 118
5.7 Selection d'une tâche par le coordinateur. 119

Liste des tableaux

1.1 Caractéristiques des processeurs les plus utilisés sur carte. 4
1.2 Caractéristiques des mémoires cartes. 5

5.1 Statistiques extraites du code FAÇADE de CAMILLE. 100
5.2 Taille des composants supportant le temps réel. 103
5.3 Temps d'exécution du support du temps réel. 104
5.4 Tailles des processeurs virtuels. 105
5.5 Temps d'exécution des activations et gel des processeurs virtuels. 105
5.6 Pourcentages du quantum de temps utiles aux applications. 106
5.7 Taille des composants supportant la coopération. 106
5.8 Temps d'exécution des activations et gels des processeurs virtuels collaboratifs. 107
5.9 Pourcentages du quantum de temps utiles aux applications. 107

Bibliographie

[ABB+86] M. Acetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach : A new kernel foundation for unix development. In *Proceedings of the Summer USENIX 1986 Technical Conference*, 1986.

[AMO+98] K. Asari, Y. Mitsuyama, T. Onoye, I. Shirakawa, H. Hirano, T. Otsuki, T. Baba, and T. Meng. Feram circuit technology for system on a chip. In *Procedings of the First NASA/DoD Workshop on Evolvable Hardware*, 1998.

[AP03] A. Arnaud and I. Puaut. Towards a predictable and high performance use of instruction caches in hard real-time systems. In *Proc. of the work-in-progress session of the 15th Euromicro Conference on Real-Time Systems*, pages 61–64, Porto, Portugal, July 2003.

[ARDG04] N. B. H. Aissa, C. Rippert, D. Deville, and G. Grimaud. A distributed wcet computation scheme for smart card operating systems. In *4^{th} International Workshop on Worst Case Execution Time (WCET) Analysis*, Catania, Sicily, Italy, June 2004.

[ARG04] N. B. H. Aissa, C. Rippert, and G. Grimaud. Distributing the wcet computation for embedded operating systems. In *25th IEEE International Real-Time Systems Symposium (RTSS) - Work-in-Progress Session*, Lisbon, Portugal, December 5-8 2004. Submitted.

[BCDR01] L. Burdy, L. Casset, D. Deville, and A. Requet. Installation de programme compilé notamment dans une carte à puce, 2001. International pending patent.

[Bla99] B. Blanchet. Escape analysis for object oriented languages. application to java. In *14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 1999.

[BM02] L. Barreto and G. Muller. Bossa: a language-based approach to the design of real-time schedulers. In *10th International Conference on Real-Time Systems (RTS'2002)*, pages 19–31, Paris, France, Mars 2002.

[BSF04] M. Q. Beers, C. H. Stork, and M. Franz. Efficiently verifiable escape analysis. In M. Odersky, editor, *ECOOP 2004, LNCS 3086*, pages 75–95, Oslo, 2004. Springer-Verlag.

[Cas02] L. Casset. *Construction correcte de logiciels pour carte à puces.* PhD thesis, Université de Marseille, Octobre 2002.

[CBR02] L. Casset, L. Burdy, and A. Requet. Formal Development of an embedded verifier for Java Card Byte Code. In *the IEEE International Conference on Dependable Systems & Networks*, Washington, D.C., USA, June 2002.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '77)*, pages 238–252, California, 1977.

[CDL02] L. Casset, D. Deville, and J.-L. Lanet. On-card bytecode verifcation the ultimate step. In *JavaOne Conference*, San Francisco, USA, 2002.

[Cha04] O. Charra. *Conception de noyaux de systèmes embarqués reconfigurables.* PhD thesis, Université Joseph Fourier — Grenoble 1, Mai 2004.

[Che00] Z. Chen. *Java Card™ Technology for Smart Cards : Architecture and Programmer's Guide.* The Java™ Series. Addison Wesley, 2000.

[CM94] C. Cardeira and Z. Mammeri. Ordonnancement de tâches dans les systèmes temps réel et répartis: Algorithmes et critères de classification. *Automatique, Productique et Informatique Industrielle*, 27(4):353 -- 384, Septembre 1994. Édition HERMES.

[CM98] C. Consel and R. Marlet. Architecturing software using a methodology for language development. In *Proceedings of the 10th International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP/ALP '98)*, pages 170–194, Pisa, Italy, September 1998.

[Col01] A. Colin. *Estimation de temps d'exécution au pire cas par analyse statique et application aux systèmes d'exploitation temps-réel.* PhD thesis, Université de Rennes I, Octobre 2001.

[CP00] 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.

[CP01a] 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.

[CP01b] 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.

[CPRS03] A. Colin, I. Puaut, C. Rochange, and P. Sainrat. Calcul de majorants de pire temps d'exécution : état de l'art. *Techniques et Sciences Informatiques (TSI)*, 22(5):651–677, 2003.

[CS02] O. Charra and A. Senart. Thinkrcx, un noyau de système d'exploitation extensible pour lego rcx. In *Actes des Journées sur les Systèmes à Composants Adaptables et Extensibles*, pages 239–244, Grenoble (France), 17-18 octobre 2002.

[CW00] K. Crary and S. Weirich. Resource bound certification. In *the 27th ACM Symposium on Principles of Programming Languages*, 2000.

[Der74] M. L. Dertouzos. Control robotics: The procedural control of physical processes. In *Proceedings of IFIP Congress 74*, pages 807–813, Stockholm, Sweden, August 1974.

[Dev01] D. Deville. Développement d'un vérifieur de bytecode javacard optimisé carte à microprocesseur. Master's thesis, Université de Lille1, 2001.

[DG02] D. Deville and G. Grimaud. Building an "impossible" verifier on a Java Card. In *2nd USENIX Workshop on Industrial Experiences with Systems Software*, Boston, USA, 2002.

[DGGJ03a] D. Deville, A. Galland, G. Grimaud, and S. Jean. Assesing the future of Smart Card operating systems. In *International Conference on Research in Smart Cards, E-smart*, Sophia-Antipolis, France, September 2003.

[DGGJ03b] 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.

[DGL98] D. Donsez, G. Grimaud, and S. Lecomte. Recoverable persistant memory of smartcard. In *Proc. 3rd Smart Card Research and Advanced Application Conference (CARDIS'98)*, Louvain-la-Neuve, Belgique, September 1998.

[DGR01] D. Deville, G. Grimaud, and A. Requet. Efficient representation of code verifier structures, 2001. International pending patent.

[DHSR05] D. Deville, Y. Hodique, and I. Simplot-Ryl. Safe collaboration in extensible operating systems: A study on real time extensions. *International Journal on Computers and Applications Special Issue on 'System & Networking for Smart Objects'*, 2005. (to appear).

[DKL+00] J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestré, J.-J. Quisquater, and J.-L. Willems. A practical implementation of the timing attack. In *Proceedings of the The International Conference on Smart Card Research and Applications*, pages 167–182. Springer-Verlag, 2000.

[DL97] 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.

[DMZ02] A. Dudani, F. Mueller, and Y. Zhu. Energy-conserving feedback edf scheduling for embedded systems with real-time constraints. In *ACM SIGPLAN Joint Conference Languages, Compilers, and Tools for Embedded Systems (LCTES'02) and*

Software and Compilers for Embedded Systems (SCOPES'02), pages 213–222, Berlin, Germany, Jun 2002.

[DRG04] D. Deville, C. Rippert, and G. Grimaud. Flexible bindings for type-safe embedded operating systems. In *ECOOP Workshop on Programming Languages and Operating Systems (ECOOP-PLOS)*, Oslo, Norway, June 2004.

[EA03] F. V. E. Akhmetshina, Pawel Gburzynski. Picos: A tiny operating system for extremely small embedded platforms. In *Proceedings of the International Conference on Embedded Systems and Applications, ESA '03*, Las Vegas, Nevada, USA, June 2003.

[EES00] J. Engblom, A. Ermedahl, and F. Stappert. Comparing different worst-case execution time analysis methods. In *Work-in-Progress session of the 21st IEEE Real-Time Systems Symposium (RTSS 2000)*, Orlando, Florida, USA, December 2000.

[EES01] J. Engblom, A. Ermedahl, and F. Stappert. A worst-case execution-time analysis tool prototype for embedded real-time systems. In *Workshop on Real-Time Tools (RT-TOOLS 2001) held in conjunction with CONCUR 2001*, Aalborg, Denmark, August 2001.

[EK95] D. R. Engler and M. F. Kaashoek. Exterminate All Operating System Abstractions. In *the 5th IEEE Workshop on Hot Topics in Operating Systems*, pages 78–94, Orcas Island, USA, May 1995.

[Eng98] D. R. Engler. *The Exokernel operating system architecture.* PhD thesis, Massachusetts Institute of Technology (MIT), 1998.

[EP898] P. E. EP8670. Cascade: "chip architecture for smartcard and intelligent device", 1998. projet Européen.

[FM02a] X. A. Feng and A. K. Mok. A model of hierarchical real-time virtual resources. In *Procedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, pages 26–39, December 2002.

[FM02b] K. Flautner and T. Mudge. Vertigo: Automatic performance-setting for linux. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation OSDI'2002*, December 2002.

[FSLM02] J.-Ph. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. THINK: A software framework for component-based operating system kernels. In *Proceedings of Usenix Annual Technical Conference*, Monterey (USA), June 10th-15th 2002.

[GB03a] A. Galland and M. Baudet. Controlling and Optimizing the Usage of One Resource. In A. Ohori, editor, *1st Asian Symposium on Programming Languages and Systems (APLAS)*, volume 2895 of *Lecture Notes in Computer Science*, pages 195–211, Beijing, China, November 27-29 2003. Springer-Verlag.

[GB03b] A. Galland and M. Baudet. Économiser l'or du banquier. In *3ème Conférence Française sur les Systèmes d'Exploitation (CFSE) - French Chapter of ACM-SIGOPS*, pages 638–649, La Colle sur Loup, France, October 14-17 2003. INRIA.

[GD01] 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.

[GLV99] G. Grimaud, J.-L. Lanet, and J.-J. Vandewalle. FAÇADE: A Typed Intermediate Language Dedicated to Smart Cards. In *Software Engineering–ESEC/FSE*, pages 476–493, 1999.

[GMGA02] A. Goodloe, M. McDougall, C. A. Gunter, and R. Alur. Predictable programs in barcodes. In *Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems*, pages 298–303. ACM Press, 2002.

[Gri00] G. Grimaud. *CAMILLE: un système d'exploitation ouvert pour carte à microprocesseur.* PhD thesis, Laboratoire d'Informatique Fondamentale de Lille (LIFL), 2000.

[Gut] S. Guthery. Mobile mind web site. `www.mobile-mind.com/htm/qna.htm`.

[Hil92] D. Hildebrand. An architectural overview of qnx. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126. USENIX Association, 1992.

[HKM03] T. A. Henzinger, C. M. Kirsch, and S. Matic. Schedule carrying code. In *Proceedings of the Third International Conference on Embedded Software (EMSOFT).* Lecture Notes in Computer Science, Springer-Verlag, 2003.

[HKQ99] G. Hachez, F. Koeune, and J. J. Quisquater. Timing attack: what can be achieved by powerful adversary. In A. M. B. et. al., editor, *20th Symp. on Information Theory in the Benelux*, pages 63–70, Haasrode (B), 27-28 1999. Werkgemeenschap Informatie- en Communicatietheorie, Enschede (NL).

[Hof00] M. Hofmann. A type system for bounded space and functional in-place update or "how to compile functional programs into malloc()-free c". *Nordic Journal of Computing*, (7(4)):258–289, 2000.

[HSW+00] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[IIT04] T. Izu, K. Itoh, and M. Takenaka. Efficient countermeasures against power analysis. In *Proc. of the Sixth Smart Card Research and Advanced Application IFIP Conference, (CARDIS'04)*, Toulouse, France, August 2004. Fujitsu Laboratories Ltd., Japan.

[ISO87] I. S. O. ISO. Carte d'identification - carte à circuit intégré à contact - partie 1: Caractéristique physique, 1987.

[ISO88] I. S. O. ISO. Carte d'identification - carte à circuit intégré à contact - partie 2: Dimensions et emplacement des contacts, 1988.

[ISO89] I. S. O. ISO. Carte d'identification - carte à circuit intégré à contact - partie 3: Signaux éléctroniques et protocoles de transmission, 1989.

[ISO94] I. S. O. ISO. Carte d'identification - carte à circuit intégré à contact - partie 4: Commandes intersectorielles pour les échanges, 1994.

[ISO99] I. S. O. ISO. Integrated circuit(s) cards with contacts, parts 1 to 9, 1987-1999.

[Jea01] S. Jean. *Models and Software Architectures for Internal and External Cooperation in Open Smart Cards.* PhD thesis, Laboratoire d'Informatique Fondamentale de Lille (LIFL), 2001.

[Joy04] M. Joye. Smart-card implementation of elliptic curve cryptography and dpa-type attacks. In *Proc. of the Sixth Smart Card Research and Advanced Application IFIP Conference, (CARDIS'04)*, Toulouse, France, August 2004. Gemplus, France.

[JV02] M. Joye and K. Villegas. A protected division algorithm. In *Proc. of the Fifth Smart Card Research and Advanced Application Conference (CARDIS '02)*, San Jose, California, USA, November 2002.

[KK99] O. Kömmerling and M. G. Kuhn. Design principles for tamper-resistant smart-card processors. In *USENIX Workshop on Smartcard Technology*, pages 9–20, 1999.

[Ler02] X. Leroy. Bytecode verification for Java smart card. *Software Practice & Experience*, 32:319–340, 2002.

[Ler03] X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3–4):235–269, 2003.

[LGD99] S. Lecomte, G. Grimaud, and D. Donsez. Implementation of Transactional Mechanisms for Open Smartcards. In *GEMPLUS Developer Conference*, 1999.

[LL73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *ACM*, 20(1):46–61, January 1973.

[LM95] 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.

[LMB02] J. Lawall, G. Muller, and L. 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, 2002.

[LMM03] J. Lawall, G. Muller, and A.-F. L. Meur. Domain-specific verification for efficient operating system extensions. Technical Report 03/03/INFO, Ecole des Mines de Nantes, 2003.

[LMM04] J. Lawall, A.-F. L. Meur, and G. Muller. Modularity for the Bossa process-scheduling language. In *ECOOP Workshop on Programming Languages and Operating Systems (ECOOP-PLOS 2004)*, Oslo, Norway, June 2004.

[Mao] Maosco. Multos web site. `www.multos.com`.

[MCM+00] G. Muller, C. Consel, R. Marlet, L. Barreto, F. Mérillon, and L. Réveillère. Towards robust oses for appliances: A new approach based on domain-specific languages. In *ACM SIGOPS European Workshop 2000 (EW'2000)*, Denmark, September 2000.

[MDS99] 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.

[Mic] Microsoft. Smartcard for windows web site. `www.microsoft.com/windowsce/smartcard`.

[NL97] G. C. Necula and P. Lee. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pages 106–119, Paris, January 1997.

[Pua02] I. Puaut. Cache analysis vs static cache locking for schedulability analysis in multitasking real-time systems. In *Proc. of the 2nd International Workshop on worst-case execution time analysis, in conjunction with the 14th Euromicro Conference on Real-Time Systems*, Vienna, Austria, 2002.

[QH01] G. Quan and X. Hu. Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors. In *Design Automation Conference*, pages 828–833, 2001.

[Qui97] J.-J. Quisquater. *The adolescence of smart cards*, volume 13 of *Future Generation Computer Systems*. 1997.

[RCG00a] A. Requet, L. Casset, and G. Grimaud. Application of the B formal method to the proof of a type verification algorithm. In *Fifth IEEE International Symposium on High Assurance Systems Engineering (HASE 2000)*, pages 115–124, 2000.

[RCG00b] A. Requet, L. Casset, and G. Grimaud. Application of the B formal method to the proof of a type verification algorithm. *HASE*, 2000.

[RD04] C. Rippert and D. Deville. On-the-fly metadata stripping for embedded java operating systems. In *Proceedings of the 6th IFIP Smart Card Research and Advanced Application Conference (Cardis'04)*, Toulouse, France, August 2004. `http://www.cardis.org/`.

[RDG04] C. Rippert, D. Deville, and G. Grimaud. Alternative schemes for low-footprint operating systems building. Technical Report RR-5220, INRIA Research, June 2004. `http://www.inria.fr/rrrt/rr-5220.html`.

[Reg01] J. Regehr. *Using Hierarchical Scheduling to Support Soft Real-Time Applications on General-Purpose Operating Systems.* PhD thesis, University of Virginia, 2001.

[RR98] E. Rose and K. H. Rose. Lightweight bytecode verification. In *Workshop "Formal Underpinnings of the Java Paradigm", OOPSLA'98*, 1998.

[RRW+03] J. Regehr, A. Reid, K. Webb, M. Parker, and J. Lepreau. Evolving real-time systems using hierarchical scheduling and concurrency analysis. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003)*, Cancun, Mexico, December 2003.

[RS94] K. Ramamritham and J. Stankovic. Scheduling algorithms and operating systems support for real-time systems. In *Proceedings of the IEEE*, volume 82, pages 55–67, January 1994.

[RS01] J. Regehr and J. A. Stankovic. HLS: A framework for composing soft real-time schedulers. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, pages 3–14, London, UK, December 2001.

[RTE] RTEMS. RTEMS : Real time operating system for multiprocessor systems. `http://http://www.rtems.com/`.

[SCS02] A. Senart, O. Charra, and J.-B. Stefani. Developing dynamically reconfigurable operating system kernels with the think component architecture. In *Proceedings of the workshop on Engineering Context-aware Object-Oriented Systems and Environments, in association with OOPSLA'2002*, Seattle, WA (USA), November 4th-8th 2002.

[Sen03] A. Senart. *Canevas logiciel pour la construction d'infrastructures logicielles dynamiquement adaptables.* PhD thesis, Université Joseph Fourier — Grenoble 1, Novembre 2003.

[SG02] N. L. Sommer and F. Guidec. A Contract-Based Approach of Resource-Constrained Software Deployment. In *the 1st IFIP/ACM Working Conference on Component Deployment*, volume 2370 of *LNCS*, pages 15–30, Berlin, Germany, June 2002.

[SM] Sun Microsystem. The javacard™ 2.1.1 language and virtual machine specification. `java.sun.com/products/javacard/javacard21.html`.

[SSNB95] J. A. Stankovic, M. Spuri, M. D. Natale, and G. C. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 28(6):16–25, 1995.

[Sta96a] J. A. Stankovic. Real-time and embedded systems. *ACM Comput. Surv.*, 28(1):205–208, 1996.

[Sta96b] J. A. Stankovic. Strategic directions in real-time and embedded systems. *ACM Comput. Surv.*, 28(4):751–763, 1996.

[tGPPGa] 3rd Generation Partnership Project (3GPP). Specification of the SIM Application Toolkit - GSM 11.14.

[tGPPGb] 3rd Generation Partnership Project (3GPP). Specification of the Subscriber Identity Module - GSM 11.11.

[Wei93] M. Weiser. Some computer science issues in ubiquitous computing. *Communication of the ACM*, July 1993.

[WGB99] M. Weiser, R. Gold, and J. S. Brown. The origin of ubiquitous computing research at PARC in the late 1980s. *IBM Systems Journal - Pervasive Computing*, July 1999.

[WLAG93] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 203–216. ACM Press, 1993.

[WW94] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proc. of the First Symposium on Operating Systems Design and Implementation*, pages 1–11, Monterey, CA, November 1994. USENIX Assoc.

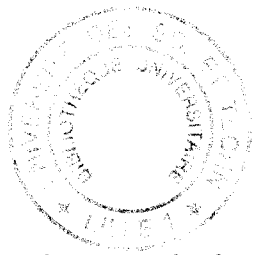
[WW95] C. A. Waldspurger and E. Weihl. W. Stride scheduling: Deterministic proportional-share resource management. Technical report, 1995.

[YK03] H.-S. Yun and J. Kim. On energy-optimal voltage scheduling for fixed-priority hard real-time systems. *Trans. on Embedded Computing Sys.*, 2(3):393–430, 2003.

[ZDS97] J. L. Z. Deng and J. Sun. A scheme for scheduling hard real-time applications in open system environment. In *Proceedings of the Ninth Euromicro Workshop on Real-Time Systems*, pages 191–199, June 1997.

[ZPS99] K. M. Zuberi, P. Pillai, and K. G. Shin. EMERALDS: a small-memory real-time microkernel. In *Symposium on Operating Systems Principles*, pages 277–299, 1999.

[ZS97] K. Zuberi and K. Shin. An efficient semaphore implementation scheme for small-memory embedded systems. In *Proc. of Real-Time Technology and Applications Symposium*, pages 25–37, 1997.



Résumé

Le logiciel carte est de plus en plus conçu pour supporter des contraintes temps réel. Par exemple, dans les cartes Java SIM, l'application principale est chargée de générer une clef cryptographique de session pour chaque unité de communication consommée faute de quoi l'infrastructure GSM rompt la communication. Actuellement, les systèmes d'exploitation pour carte à puce ne gèrent l'aspect temps réel qu'au cas par cas. Ils ne permettent pas aux applications «utilisateur» de signifier des besoins en termes d'accès au microprocesseur, ceci pour des raisons de sécurité.

Nous proposons une architecture logicielle embarquée autorisant le partage de la ressource microprocesseur entre les extensions de l'exo-noyau CAMILLE. Cette architecture permet aux extensions de supporter des tâches temps réel au dessus de l'exo-noyau garantissant la disponibilité du microprocesseur. Nous avons montré les faiblesses des solutions initialement préconisées pour supporter du temps réel dans les exo-noyaux et nous proposons un moyen de faire collaborer les extensions sous la forme d'un partage d'une de leurs politiques d'ordonnancement et d'une mutualisation de leurs accès au microprocesseur. Nous avons mis en avant les propriétés fonctionnelles que nous attendons de ces ordonnanceurs collaboratifs et nous avons proposé une architecture distribuée permettant de charger et de valider ces propriétés. Cette architecture de partage du microprocesseur a été validée expérimentalement dans CAMILLERT.

Mots-clef : Systèmes d'exploitation, Architectures extensibles, Temps réel, Systèmes embarqués, Carte à microprocesseur.

Abstract

Smartcards software is more and more designed to support real time constraints. For example, in a Java SIM card, the most important application is the one that generates cryptographic session keys. Those keys must be delivered within a firm deadline (one per communication unit used), otherwise the cell phone is disconnected from the network. Current smartcards operating systems only support real time constraints for applications designed by smartcard manufacturers. For security reasons, they are not available at the user application level.

We propose an embedded software architecture allowing to share the access to the microprocessor between all the extensions of the CAMILLE exokernel. This architecture enables extensions to support real time applications on top of the exokernel which guarantees the microprocessor availability. We have highlighted the weaknesses of previously proposed solutions to support real time applications in an exokernel. We propose to allow extensions to collaborate by sharing their access to the microprocessor and also their scheduling policies. We have formalized the functional properties we expect from these collaborative scheduling policies and proposed a distributed architecture that allows their validation. This architecture has been evaluated in the CAMILLERT prototype.

Keywords : Operating systems, Extensibles architectures, Real time, Embedded systems, Smartcards.