Université Lille 1 - Sciences et Technologies

Département de formation doctorale en informatique **École doctorale SPI Lille**
UFR IEEA

Exécution d'applications stockées dans la mémoire non-adressable d'une carte à puce

THÈSE

présentée et soutenue publiquement le 13 décembre 2012

pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille
(spécialité informatique)

par

Geoffroy Cogniaux

Composition du jury

Président :	Pierre Boulet – LIFL, Université Lille 1 Sciences et Technologies
Rapporteurs :	Didier Donsez – LIG, Université Joseph Fourier Grenoble 1 Gaël Thomas – LIP6, UPMC Sorbonne Universités Paris
Examinateur :	Guillaume Salagnac – CITI, INSA de Lyon
Directeur de thèse :	Gilles Grimaud – LIFL, Université Lille 1 Sciences et Technologies
Co-Encadrant de thèse :	Michael Hauspie – LIFL, Université Lille 1 Sciences et Technologies
Invité :	François-Xavier Marseille – Gemalto, Technology & Innovation France

Laboratoire d'Informatique Fondamentale de Lille
Université Lille 1 - Sciences et Technologies
Gemalto Technology and Innovation, France

Numéro d'ordre: XXXXX

gemalto security to be free

lifL

Université Lille1 Sciences et Technologies

« *Nous ne savons jamais où nos actions nous mèneront.*
Mais nous savons que si nous ne faisons rien,
elles ne nous mèneront nulle-part. »

Gandhi

Table de matières

1 Introduction **1**

2 État de l'art **3**

2.1 Embarquer du logiciel dans une carte à puce 3

2.1.1 Architecture processeur d'une carte à puce 4

2.1.2 Architecture mémoire d'une carte à puce 4

2.1.2.1 Une mémoire non-volatile adressable : la Flash NOR . . . 5

2.1.2.2 Une mémoire non-volatile non-adressable : la Flash NAND 5

2.1.2.3 Limites de la flash NAND 6

2.1.3 Modèle applicatif maître-esclave 7

2.1.3.1 Protocole de communication 7

2.1.3.2 Applications événementielles 8

2.1.4 Ajouter du logiciel à une carte à puce 8

2.1.4.1 Mutations successives 8

2.1.4.2 Nouvelles plateformes applicatives 9

2.1.5 Les défis de la *post-issuance* 9

2.1.5.1 Le défi de la sécurité 9

2.1.5.2 Machines défensives et analyseurs de code 10

2.1.5.3 Le défi du stockage 13

2.1.6 Embarquer plus de logiciels dans une carte à puce 13

2.2 De Java à JavaCard 14

2.2.1 Le langage Java et sa machine virtuelle 14

2.2.1.1 Processus virtuels d'exécution 14

2.2.1.2 Processus de gestion de la mémoire 15

2.2.1.3 Classes, objets et méta-données 16

2.2.2 La plate-forme JavaCard 2.2: Java dans une tête d'épingle 17

2.2.2.1 Une Machine Virtuelle scindée 17

2.2.2.2 Conversion 17

2.2.2.3 Briques logicielles 18

2.2.2.4 JavaCard2 : Un sous-ensemble du langage Java 18

2.2.3 Synthèse 19

2.3 Cache mémoire 19

2.3.1 Fonctionnement général 19

2.3.2 Stratégies de placement 20

2.3.2.1 Placement par allocation 21

2.3.2.2 Placement par segmentation 23

2.3.3 Stratégies de renouvellement 23

2.4 Synthèse 24

3 Problématique **25**

3.1 Contexte et opportunités 25

3.1.1 *Post-issuance* massive 25

3.1.2 Bénéfices potentiels 26

3.2 Problématique : le mur des temps de latence 27

3.2.1 NAND versus NOR, le couperet de la réalité 27

3.2.2 Insuffisance de l'approche par tampon 28
3.3 Approches 29
3.3.1 Première approche : un cache mémoire 29
3.3.1.1 Un cache d'instructions en logiciel 29
3.3.1.2 Les clés de l'efficacité d'un cache 30
3.3.1.3 Limitations d'un cache 31
3.3.2 Deuxième approche : le recouvrement d'opérations 32
3.3.2.1 Présentation 32
3.3.2.2 Recouvrement par ré-ordonnancement 32
3.3.2.3 Pré-chargement sans matériel 32
3.3.3 Notre approche : regroupement d'accès au cache 35
3.3.3.1 Contraintes structurelles d'un cache 35
3.3.3.2 Vers une découverte dynamique de groupes d'accès 36
3.3.3.3 Méthode de découverte dynamique des groupes 37
3.4 Méthodologie 38
3.4.1 Critères d'évaluation 38
3.4.2 Programmes de tests 39
3.4.3 Protocoles expérimentaux 41

4 Caches logiciels pour cartes à puce 43
4.1 Cache logiciel à faible empreinte mémoire 43
4.1.1 Propriétés généralisables des blocs de données 45
4.1.1.1 Taille 45
4.1.1.2 Degré de séquentialité 45
4.1.1.3 Points chauds 45
4.1.2 Stratégies de placement et empreinte mémoire 47
4.1.2.1 Impact sur l'allocation par taille variable 47
4.1.2.2 Impact sur l'allocation par taille fixe 47
4.1.2.3 Impact sur la segmentation 48
4.1.3 Limite théorique du renouvellement 50
4.1.4 Stratégies d'accès au contenu d'un cache 51
4.1.4.1 Algorithmes de recherche 51
4.1.4.2 Empreinte mémoire des algorithmes de recherche 53
4.1.4.3 Débit pour des empreintes mémoires comparables 54
4.1.5 Coût d'exécution d'un cache logiciel 55
4.1.5.1 Coût des stratégies de placement 55
4.1.5.2 Coût en instructions 56
4.1.5.3 Analyse des coûts en instructions 56
4.2 Cache d'instructions logiciel pour cartes à puce 58
4.2.1 Placement de blocs de base 58
4.2.1.1 Principe de localité 58
4.2.1.2 Tailles des blocs de base 59
4.2.1.3 Fréquences de réapparition de tailles de bloc 59
4.2.1.4 Dispersion des points chauds 61
4.2.1.5 Synthèse 65
4.2.2 Évaluation croisée du placement et du renouvellement 65
4.2.3 Coût d'exécution d'un cache d'instructions logiciel 67
4.2.3.1 Unification des paramètres 67
4.2.3.2 Résultats expérimentaux 68
4.2.3.3 Évaluation et analyse du coût global 68

4.3 Synthèse et conclusions 70

5 Cache logiciel de méta-données Java et JavaCard 73

5.1 Introduction au modèle de méta-donnée Java 73
5.1.1 Désambigüisation 73
5.1.2 Objectif et méthode 74
5.1.3 Fichier de classe : méta-données brutes 75
5.2 Chargement des méta-données Java 77
5.2.1 Chargement de classes 78
5.2.1.1 Acquisition des méta-données 78
5.2.1.2 Édition des liens 78
5.2.1.3 Initialisation 79
5.2.2 Pré-chargement de classes 79
5.2.2.1 Chargement précoce 79
5.2.2.2 Format pré-chargé 80
5.2.3 Synthèse 80
5.3 Modéle KVM 80
5.3.1 Motifs d'accès aux méta-données 81
5.3.2 Cas d'étude : INVOKEVIRTUAL 82
5.3.3 Spécificités de KVM 84
5.4 Modèle JavaCard 85
5.4.1 Un modèle condensé 85
5.4.2 Motifs d'accès revisités 85
5.5 Vers la mise-en-cache des méta-données 88
5.5.1 Préparation des expérimentations 88
5.5.1.1 Modification de KVM 88
5.5.1.2 Modification de la JCVM 88
5.5.2 Répartition des méta-données 89
5.5.2.1 Répartition des méta-données brutes 89
5.5.2.2 Répartition des méta-données dans KVM 90
5.5.2.3 Répartition des méta-données dans JavaCard 90
5.5.3 Utilisation des méta-données 91
5.6 Cache de méta-données logiciel 93
5.6.1 Cartographies 93
5.6.1.1 Cartographie de KVM 93
5.6.1.2 Cartographie de JavaCard 95
5.6.1.3 Dilution spatiale 96
5.6.2 Cache de méta-données 96
5.7 Synthèse 98

6 Pré-interprétation de code JavaCard 99

6.1 Vers une pré-interprétation de code 99
6.1.1 Interactions entre l'interpréteur et le cache logiciel 99
6.1.2 Problème posé par une interaction systématique 102
6.1.3 Dépasser le modèle d'interactions systématiques 102
6.1.3.1 Identification du verrou 102
6.1.3.2 Clés de l'efficacité 103
6.1.4 Vers une pré-interprétation de code 103
6.1.4.1 Clés de conception 103
6.1.4.2 Détermination des limites de plages d'adresses physiques . 104

6.1.4.3 Préparation précoce de l'interprétation concrète 104
6.2 Conception d'un pré-interpréteur 105
6.2.1 Architecture générale 105
6.2.1.1 Surcharge de la pile d'accès à la mémoire non-adressable . 105
6.2.1.2 Principe de fonctionnement 106
6.2.2 Objectif : maîtriser le coût d'exécution 109
6.3 Pré-interprétation de code JavaCard 109
6.3.1 Code JavaCard 2.2 109
6.3.2 Pré-décodages : mode opératoire 110
6.3.2.1 Support d'analyse 110
6.3.2.2 Éviter la redondance des décodages 111
6.3.3 Analyse et rupture de flots 113
6.3.3.1 Preuve d'une analyse bornée 113
6.3.3.2 Pré-décodage de méta-données 114
6.3.4 Pilotage actif du cache 116
6.3.4.1 Forcer la politique de remplacement 116
6.3.4.2 Pré-chargements 116
6.3.5 Synthèse 117
6.4 Confrontation du gain et du coût 117
6.4.1 Maîtrise du coût d'exécution 118
6.4.2 Évaluation expérimentale de la pré-interprétation 118
6.4.2.1 Protocole expérimental 118
6.4.2.2 Évaluation de la pré-interprétation 119
6.4.2.3 Évaluation du pré-chargement 119
6.4.3 Preuve de concept 122
6.5 Conclusions 124

7 Conclusion 125
7.1 Synthèse 125
7.2 Perspectives : transposer la pré-interprétation 126

Bibliographie 129

A Compléments cartographiques 137

B Compléments graphiques sur les débits 141

C Structures de données 145

CHAPITRE 1

Introduction

Contexte

Les cartes à puce sont des systèmes informatiques miniatures principalement dédiés à la sécurité. Elles sont ainsi connues pour être des systèmes fermés embarquant du logiciel opaque. Néanmoins, ces dernières années ont vu ces systèmes s'ouvrir sur l'extérieur en proposant désormais la possibilité d'y installer des applications après leur mise en circulation. Cette mutation n'est toutefois pas intervenue sans introduire de nouvelles problématiques de sécurité et de génie logiciel, alors que le matériel n'a quant à lui que très peu évolué. Dans ces circonstances, la capacité de stockage mémoire d'une carte à puce reste un frein à la quantité d'applications embarquables.

Dans ces conditions, il apparait alors opportun de stocker ces nouvelles applications dans la mémoire Flash série que les carte à puce possèdent de plus en plus souvent. Cependant cette mémoire bien que large est non-adressable par la processeur et n'est donc pas propice à l'exécution de code en place comme la Flash interne. De plus, ces mémoires ont un temps de latence très élevé.

Plusieurs approches sont alors envisageables mais la plus efficace est sans conteste l'utilisation d'un cache d'instructions. Mais malheureusement, une carte à puce n'a pas de support matériel pour ce type d'actions.

Thèse

Dans ce mémoire, nous défendons la thèse qu'il est possible d'exécuter des applications, natives ou Java, stockées dans la mémoire non-adressable d'une carte à puce, sans support matériel couvrant la latence de celle-ci. Nous présentons donc dans ce document une étude sur les caches logiciels, en revisitant l'état de l'art sur ceux-ci par le prisme d'une empreinte mémoire faible, condition *sine qua none* d'une solution encartable.

La faiblesse connue et admise de cette thèse réside justement dans l'empreinte mémoire qui est le facteur de performances communément acquis d'un cache, qu'il soit matériel ou logiciel, comme le montrent les tailles des caches processeurs, des disques durs,s des bases de données ou des serveurs web. D'autre part, le simple fait de parler de cache logiciel sous-entend un coût d'exécution de celui-ci bien supérieur à celui d'un cache matériel, rendant la récupération d'une instruction prohibitive par rapport à son exécution pure.

Dans ce mémoire, nous appliquons notre étude des caches logiciels dans un premier temps à la problématique des caches d'instructions, puis aux caches de méta-données `Java/JavaCard`, une autre famille de données exécutables. Enfin, nous montrerons comment les faiblesses des caches logiciels peuvent être dépassées par une préparation précoce de données exécutables contenues dans l'espace de cache.

Structure de ce mémoire

Le **chapitre 2** présente un état de l'art des plateformes applicatives pour cartes à puces, puis présente les spécificités et fonctionnalité de la plateforme la plus répandue qu'est `JavaCard2`. Enfin, dans une troisième partie est présenté un état de l'art sur les cache mémoires, fil conducteur de tout ce document.

Le **chapitre 3** présente le contexte des travaux présentés dans ce mémoire, puis en extrait la problématique. Il présente ensuite les principales approches qui peuvent être suivies pour la résoudre. Enfin, nous présentons notre méthodologie, basée sur une approche phénoménologique du problème.

Le **chapitre 4** présente notre étude sur les caches logiciels contraints par une faible empreinte mémoire. Puis dans un second temps, nous évaluons nos constats et conclusions sur le cas d'un cache d'instructions logiciel, pour en montrer les forces mais aussi malheureusement les faiblesses, préjudiciables pour une carte à puce.

Le **chapitre 5** se focalise sur l'étude des méta-données Java/JavaCard qui sont à leur niveau également des données exécutables sans être du code ni des données pures. Nous montrons comment la construction d'un modèle de méta-données influencent directement les performances de la JVM, puis dans un second temps, celles d'un cache de méta-données logiciel.

Le **chapitre 6** présente l'approche que nous proposons pour améliorer suffisamment les performances d'un cache logiciel à faible empreinte pour qu'il puisse être embarqué dans une carte à puce.

Enfin, le **chapitre 7** apporte une synthèse de ces travaux et en donne quelques perspectives.

Chapitre 2

État de l'art

Les cartes à puce autrefois fermées s'ouvrent aujourd'hui au monde extérieur en permettant de télécharger de nouvelles application alors que la carte a déjà été mise en circulation. Cette nouvelle approche ouvre sur plusieurs problémes comme la sécurité et l'espace disponible pour stocker ces applications, alors que dans le même temps, le support matériel n'a quant à lui que peu évolué.

Dans ce chapitre, nous présentons un état de l'art de ces nouvelles problématiques, en trois temps. Nous commençons par aborder l'aspect embarquement sécurisé d'applications tiers dans une carte à puce. Puis nous décrivons la plate-forme technologique JavaCard, qui est le support de génie logiciel le plus répandu à ce jour pour ces nouvelles cartes à puce. Enfin, nous abordons l'aspect espace de stockage de code et d'applications en revenant sur la technologie des caches mémoires, solution la plus commune pour étendre un espace mémoire principal en s'aidant d'une mémoire secondaire, bien que beaucoup plus lente.

2.1 Embarquer du logiciel dans une carte à puce

Les cartes à puce sont des petits objets informatiques dont la vocation est d'avoir à portée de la main en toutes circonstances des données personnelles sensibles et/ou secrètes. Le couple matériel/logiciel formant une carte à puce est ainsi conçu d'une part pour la sécurisation de ces données et d'autre part pour l'extrême portabilité d'un objet discret et nomade. Mais glisser une clé cryptographique et des données bancaires ou médicales dans une poche de veston ou un porte-feuille implique à l'origine des choix de conception radicaux.

Dans le matériel d'abord, qui doit allier coût de production unitaire extrêmement faible avec robustesse, fiabilité, et sécurité. L'information secrète ne doit ni être altérée dans le temps, ni fuir du matériel par un dysfonctionnement de celui-ci, ni être transmise sous une forme erronée.

Pendant longtemps, le logiciel a quant à lui été conçu par co-conception avec le matériel. Car cette approche permet une convergence de la performance, de la sécurité et du coût de production dans un système dédié à une tâche bien précise. Toutefois aujourd'hui, la politique applicative des cartes à puce est à l'ouverture sur le monde extérieur avec la possibilité pour le porteur de carte de pouvoir ajouter lui-même de nouvelles applications dans certaines de ses cartes à puce. Ce nouveau mode d'utilisation de la carte à puce, s'il n'a pas réellement entraîné de modifications dans la conception du matériel, a largement modifié l'architecture logicielle pour offrir un support à la personnalisation.

Bien évidemment, cette capacité à pouvoir modifier à volonté le contenu applicatif, implique de nouvelles problématiques de sécurité à résoudre. Cet aspect est aujourd'hui un des domaines de recherche les plus animés dans la catégorie de ces cartes à puce dites « ouvertes ». Néanmoins, la personnalisation connait une autre limite que nous adressons dans cette thèse. Cette limite est le volume applicatif que les cartes ouvertes sont capables d'accepter. En effet, le matériel n'ayant pas beaucoup évolué, la quantité d'espace

mémoire disponible pour le stockage et l'exécution de nouvelles applications n'a ainsi que peu augmenté.

Dans cette section, nous présentons un état des lieux de l'architecture matérielle puis logicielle des cartes à puces « ouvertes » actuelles. Nous regarderons plus particulièrement quels sont les freins à la personnalisation massive et les opportunités qui restent offertes avec le matériel existant.

2.1.1 Architecture processeur d'une carte à puce

L'unité centrale d'une carte à puce est de la taille d'un micro-contrôleur. C'est cette unité centrale qui rend la carte à puce programmable et « intelligente » [1]. Ce micro-contrôleur intègre également un peu de mémoire volatile pour l'exécution et plusieurs types de mémoires non-volatiles détaillées un peu plus bas. Tous ses composants communiquent entre eux au travers d'un bus. Les cartes recourant au chiffrement possèdent généralement un co-processeur cryptographique gérant au niveau matériel des algorithmes de chiffrement symétrique comme DES ou AES, ou asymétrique comme RSA [Eisenbarth 2007].

Si tout ceci est à l'image de n'importe quel système informatique, la différence fondamentale se trouve dans les tailles et puissances de chaque composant. Historiquement, les processeurs des cartes à puce sont des processeurs 8 bits connus pour produire des applications très compactes et être peu gourmands en énergie. Mais les besoins évoluant, on trouve maintenant des processeurs 32 bits suffisamment légers pour convenir aux cahiers des charges d'une carte à puce. La plupart d'entre elles sont cadencées à des fréquences allant de 4 à 20 MHz (voir tableau 2.1). Côté mémoire, une carte à puce est tout aussi limitée, car essentiellement conditionnée par la taille restreinte du silicium qui ne peut dépasser quelques millimètres-carrés. La quantité de RAM d'une carte à puce peut commencer à seulement 512 octets et ne dépassent que rarement les 48 Ko. Les cartes à puce modernes contiennent par contre maintenant jusqu'à 256 Ko de Flash interne adressable (*i.e.* incluse dans l'espace d'adressage du CPU). À l'heure actuelle, cet espace mémoire est le seul contenant des binaires exécutables, du système d'exploitation aux applications.

Tableau 2.1: Exemples d'architectures matérielles de micro-contrôleurs pour cartes à puce

Modèle	Architecture	Taille Bus	Registres	Fréquence
68H05	CISC	8 bits	2 * 8 bits	4,77 Mhz
80xx51	CISC	8 bits	6 * 8/16 bits	4,77 Mhz
AVR AT90	RISC/CISC	8 bits	32 * 8/16 bits	4,77 Mhz - 44,7 Mhz
ARM7xx	RISC	32 bits	16 * 32 bits	4,77 Mhz - 30 Mhz
R4KSC	RISC	32 bits	32 * 32 bits	4,77 Mhz - 100 Mhz

2.1.2 Architecture mémoire d'une carte à puce

Une carte à puce embarque plusieurs types de composant mémoire. Le premier bien évidement est de la mémoire vive, RAM[2] ou SRAM[3] . Elle contient principalement la pile d'exécution. On trouve également un peu de mémoire morte (ROM[4]) pour le stockage de

[1] Le terme désignant une carte en anglais est d'ailleurs encore plus explicite, puisqu'elle se traduit par SmartCard, la carte intelligente.

[2] Random Access Memory

[3] Static Random Access Memory

[4] Read-Only Memory

code critique et le code de démarrage[5], ainsi que de la mémoire Flash pour le stockage du code exécutable, de données applicatives, et parfois d'un système de fichiers.

Les mémoires Flash sont des mémoires à stockage persistant et effaçables électriquement inventées par Toshiba vers le milieu des années 80. Leur principale caractéristique par rapport à l'EEPROM[6], et leur mode d'effacement qui se fait par blocs et non plus par octets, ce qui rend les écritures plus rapides que dans cette dernière.

Il existe aujourd'hui deux types de mémoire Flash: la NOR et la NAND (figure 2.1). Elles tirent chacune leur nom de la manière dont sont utilisées les portes logiques de leurs transistors. Bien qu'ayant une origine commune, elles ont toutefois plusieurs différences majeures qui les renvoient chacune à des usages bien différents.

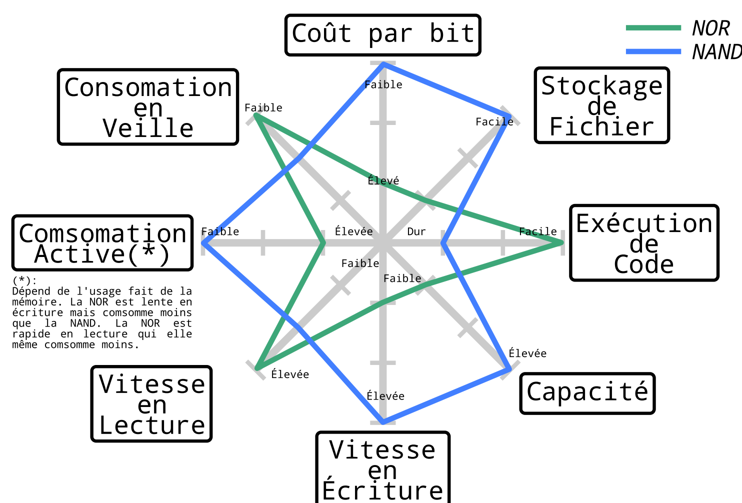


Figure 2.1: Comparaison entre NOR et NAND
- source [Toshiba 2003], traduite en français.

2.1.2.1 Une mémoire non-volatile adressable : la Flash NOR

La NOR est inscriptible par blocs mais permet un accès en lecture à la taille d'un mot machine, permettant une performance remarquable pour des accès aléatoires. Cette caractéristique fait d'elle depuis son invention une mémoire idéale pour le stockage de code dans un micro-contrôleur en remplacement des anciennes ROM, moins souples, et autres EEPROM plus lentes. Si, comme dans un micro-contrôleur, la RAM et la NOR sont sur le même bus, lire un octet en NOR se fait à la même vitesse que dans la RAM. Le réel handicap de la flash NOR est son temps d'écriture qui reste tout de même très lent. Écrire un bloc peut prendre jusqu'à 900 milli-secondes dans les modèles bas de gamme.

2.1.2.2 Une mémoire non-volatile non-adressable : la Flash NAND

La Flash NAND est accessible par blocs, que se soit en lecture ou en écriture. La NAND est en fait une NOR dont on a réduit le nombre de portes par cellule de stockage [Micron 2006], ce qui l'a rendu plus dense. Cette technologie permet donc de stocker plus de données que la

[5] Bootloader en anglais

[6] Electrically-Erasable Programmable Read-Only Memory, une autre mémoire effaçable électriquement, plus ancienne.

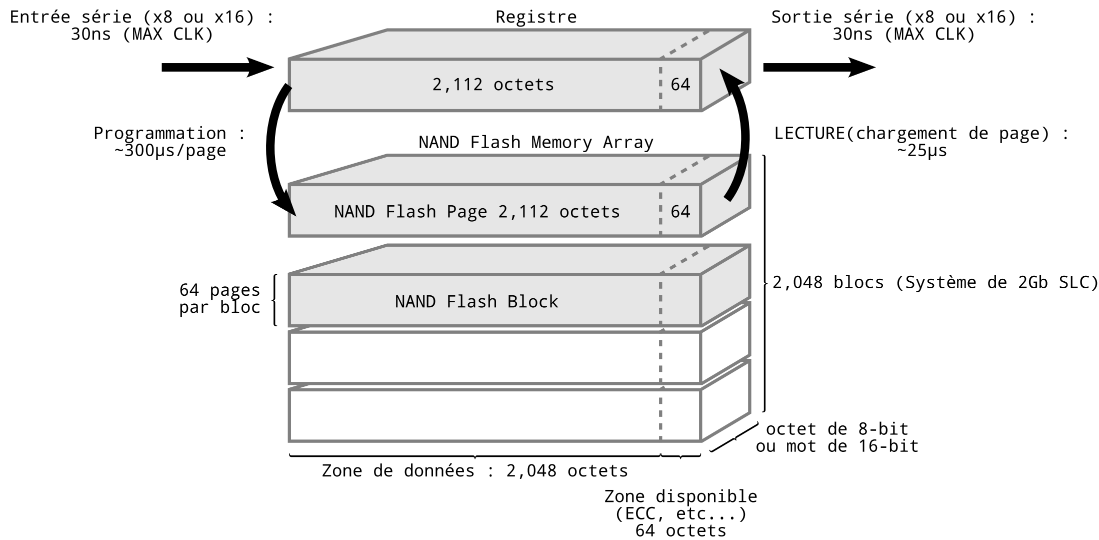


Figure 2.2: Organisation interne de la flash NAND - source [Micron 2006]

NOR à surface de silicium équivalente. L'autre avantage de cette réduction est de diminuer fortement les temps d'écriture et d'effacement.

Du fait de sa densité plus élevée, la NAND a un coût par octet plus faible de 30 à 40 % par rapport à la NOR [Toshiba 2003].

2.1.2.3 Limites de la flash NAND

La flash NAND est organisée en blocs, un bloc étant la plus petite unité d'effacement. Chaque bloc est lui même découpé en pages, la plus petit unité d'écriture et de lecture. Ces lectures et écritures se font par un registre matériel de la taille d'une page. Une demande d'accès se déroule en deux étapes. Tout d'abord, le contrôleur de NAND charge la page demandée dans le registre. Ensuite et seulement lorsque la page est complète dans le registre, chaque octet peut y être lu séquentiellement par l'applicatif.

Une page de NAND supporte un nombre limité d'écritures, au-delà duquel la cohérence des données de la page n'est plus garantie. La lecture est elle aussi non-fiable et quelques bits peuvent être mal positionnés. Ces deux défauts ont amené l'ajout d'une étape intermédiaire de calcul d'un code de correction d'erreur (ECC). Dans la plupart des NAND, ce calcul se fait dans le matériel pour ne pas être trop pénalisant, mais doit être vérifié au niveau logiciel. Ces données de contrôle sont stockées dans une extension de la page (ou *spare*), ce qui augmente le volume de données lues par page (figure 2.2).

Dans les NAND de petite taille prévues pour fonctionner sur des micro-contrôleurs, la gestion des blocs défectueux est complètement ignorée par la NAND et doit donc être gérée au niveau logiciel. La technique utilisée s'appelle le *wear-levelling*[7] dans une couche d'abstraction logicielle de la NAND appelée *Flash Translation Layer* (FTL, couche de traduction de la Flash). L'objectif du *wear-leveling* est de prolonger la durée de vie de chaque bloc en répartissant le plus uniformément possible les écritures et les effacements sur tout l'espace physique disponible. Des informations contenues dans une page physique P peuvent ainsi être déplacées vers la page $P + i$. La FTL pilote donc le *wear-leveling* par un algorithme cartographiant ces déplacements et mémorisant une association 1 vers 1 entre

[7] mise à niveau de l'usure

une localisation logique immuable et une localisation physique modifiable à l'intérieur de la NAND. Pour un approfondissement de ce sujet, le lecteur peut se référer à la veille technologique de Chung *et al* [Chung 2006].

L'information à retenir ici sur ce point est que le temps de latence des lectures dans la NAND ne se limite pas à la seule latence du matériel, mais comprend également la traversée de cette couche logicielle qu'est la FLT.

2.1.3 Modèle applicatif maître-esclave

Les cartes à puce sont normalisées selon le standard ISO-7816. La taille du substrat en plastique et le positionnement du connecteur sont définis dans les documents 7816-1 (caractéristiques physiques) et 7816-2 (dimensions et positionnement des contacts). Cette norme définit également une pile de communication (figure 2.3) basée sur une architecture à commande/réponse. Une carte à puce, bien que dotée d'un microprocesseur n'en reste pas moins passive par rapport au terminal. Lui seul peut envoyer des commandes, la carte n'envoyant que des réponses. Si des données sont à renvoyer au terminal, elle ne pourra le faire que si celui-ci le lui demande.

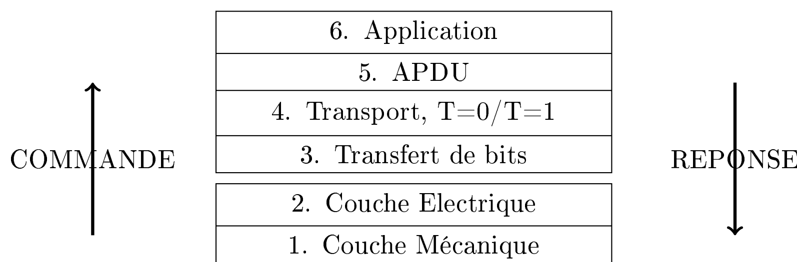


Figure 2.3: Pile de communication ISO-7816

2.1.3.1 Protocole de communication

Au dessus des couches physiques (mécanique, électrique) se trouve la couche de transfert de bits, cadencée au niveau signal. Ce signal est à faible fréquence, rendant la communication relativement lente. La fréquence de ce signal est négociée entre le lecteur et la carte à puce au moment de la mise sous-tension (démarrage à froid) ou d'un redémarrage à chaud (RESET). Cette négociation, ou ATR[8], comporte ensuite une phase de négociation portant sur le type de protocole de transport qui sera utilisé. Il existe actuellement deux protocoles possibles, définis dans ISO-7816-3. Le premier, T=0, est un protocole série orienté octet. Le second, T=1, est orienté bloc pour une transmission en rafale.

Au dessus de la couche transport, la norme ISO-7816-4 définit le format d'échange de messages qui est indifféremment l'enveloppe d'une commande ou d'une réponse. Ce format de message appelé APDU[9], comporte une entête et optionnellement un corps de message de longueur variable. L'entête contient : la commande sous forme de code normalisé, les paramètres pour cette commande, une indication concernant la présence de données dans le corps de l'APDU et sa longueur. La carte répond en envoyant deux octets de status. La réponse ne contient pas immédiatement des données. Le code de retour doit indiquer soit une erreur, soit un succès soit signifier au lecteur que des données sont disponibles en envoi.

[8] Answer To Reset
[9] Application Protocol Data Unit

2.1.3.2 Applications événementielles

La plupart des systèmes embarqués fonctionnent sur un mode dit événementiel. Dans ces systèmes dont l'objectif n'est pas le calcul mais l'interaction avec un environnement, la notion d'événement est omniprésente. Un événement comme la réception d'une trame réseau, un changement de température, le déclenchement d'une alarme, provoque l'exécution d'une routine associée qui ne se serait jamais exécutée d'elle-même.

Dans ce genre de systèmes, les applications embarquées sont donc conçues sur ce modèle de ré-exécution périodique. Elles doivent établir clairement un début et une fin de traitement de l'événement et gérer de possibles réentrances pour traiter des événements concourants.

Le mode maître-esclave des cartes à puce pousse le modèle événementiel vers un mode unidirectionnel où l'application s'exécutant sur la carte à puce doit réagir à un seul type d'événement : la réception d'une commande. La différence est qu'une carte à puce peut être mise hors-tension entre deux réceptions de commande. Les applications doivent donc être conçues pour être capables de gérer de la corrélation d'événements entre-coupés d'arrêt du système.

2.1.4 Ajouter du logiciel à une carte à puce

L'intégration de logiciels dans une carte à puce a évolué au fil du temps suivant les besoins utilisateurs et les contraintes du marché. Elle a commencé par le tout intégré et figé à la fabrication pour aboutir aujourd'hui à la possibilité d'ajout d'applications après mise en circulation de la carte.

Pour comprendre la situation actuelle, cette sous-section décrit les tournants majeurs dans le mode de conception des systèmes d'exploitation et supports applicatifs des cartes à puce.

2.1.4.1 Mutations successives

Première génération Dans la première génération de cartes à puce, l'utilisateur n'avait accès à aucune donnée ni aucune application de la carte. Il devait se « contenter » d'utiliser ce que le fabricant avait bien voulu y mettre. En l'occurrence, il s'agissait le plus souvent de données à usage unique qui étaient la plupart du temps confidentielles, connues seulement du fournisseur, gérées par des systèmes d'exploitation minimalistes.

Deuxième génération La deuxième génération de carte à puce a eu pour objectif de réduire le *time-to-market*[10] inhérent à un succès commercial comme celui de la carte à puce. Dans ce type de nouvelle carte, l'application était toujours dédiée, mais l'espace de données était devenu accessible et était organisé en interne sous forme de système de fichier, ce qui permettait un contenu modifiable dans le temps. Le système d'exploitation devient alors un peu plus sophistiqué pour pouvoir gérer ces nouveaux types d'accès. On peut citer comme exemple les cartes vitales ou les premières générations de carte SIM permettant à leurs utilisateurs de personnaliser leur base de données de contacts, un agenda, etc...

Troisième génération La troisième génération permet désormais le téléchargement d'applications sur la carte. Cette génération de systèmes d'exploitation a clairement été conçue pour séparer l'aspect fabrication et industrialisation, de l'aspect développement d'applications

[10] Délai de mise sur le marché.

et donc de personnalisation. C'est ce qu'on appelle la *post-issuance*, cette possibilité de modifier le contenu de la carte à puce, même après sa mise en circulation.

Cette génération de carte est aussi celle qui a vu des langages de haut niveau comme le Java ou le C# franchir le barrière de la miniaturisation. L'objectif de cette nouvelle approche est alors d'offrir une grande souplesse dans le développement de nouveaux contenus grâce à des technologies réputées pour leur agilité.

2.1.4.2 Nouvelles plateformes applicatives

Avec la possibilité de télécharger de nouvelles applications sur une carte *post-issuance*, la coupure est désormais nette entre les deux rôles de fournisseur de supports physiques et fournisseur de logiciels. Le co-conception matériel/logiciel au niveau systèmes d'exploitation existe toujours du fait des spécificités d'une carte à puce, mais même là, de plus en plus de solutions basées sur des systèmes d'exploitation sur l'étagère voient le jour (Aspect, IBM JCOP, Ecebs Multfile,...).

Idem au niveau applicatif où l'offre en plateformes de développement s'est beaucoup élargie. Si la plateforme JavaCard basée sur la technologie Java reste dominante, d'autres plateformes comme Microsoft .NET, Multos, et d'autres ont rejoint désormais cette offre.

Ces plateformes sont à la fois des environnements de développement et des environnements d'exécution. Dans ce cadre, une application est construite sur station de travail à l'aide de *framework* et de bibliothèques, qu'elle pourra retrouver à l'identique sur la carte, dans la version exécutable du même environnement. Cette approche allège la taille des applications et le volume des téléchargements, offrant ainsi énormément de souplesse et répondant aux contraintes des cartes à puce.

Dans ce contexte, la conception d'applications pour carte à puce est désormais bien loin des boites noires inaccessibles, et difficiles à programmer par les non-spécialistes.

2.1.5 Les défis de la *post-issuance*

2.1.5.1 Le défi de la sécurité

Si la *post-issuance* ajoute indéniablement de la flexibilité et une mise sur le marché accélérée, cela ne se fait malheureusement pas sans introduire de nouvelles problématiques, et notamment des problématiques de sécurité.

Les générations de cartes « fermées » garantissaient un haut niveau de sécurité car le système complet pouvait être validé *in-vitro* par le fabricant, par l'utilisation de méthodes formelles par exemple. Même si cette preuve n'était pas fourni au grand public, la clôture définitive du système juste avant sa mise en circulation garantissait de manière tacite que le système ne pourrait plus faillir.

De ce point de vue, la *post-issuance* a complètement changé la donne. Cette garantie tacite d'un système stable et fiable est brisée par la possibilité d'ajouter de nouvelles applications, potentiellement malveillantes, même involontairement. En effet, dans l'absolu, rien ne prouve à l'utilisateur de la carte ni à son système d'exploitation que la nouvelle application provient d'une source fiable. De plus, et même si c'était le cas, rien ne garantit que la nouvelle application prouvée fiable d'une manière ou d'une autre soit à 100 % compatible avec ce que la carte contient déjà.

Dans ces circonstances, la nécessité d'analyser puis vérifier, voire « prouver » l'application est obligatoire et ne peut être réalisé que par le système d'exploitation de la carte lui-même. C'est à dire en toute fin de déploiement, entre le moment où la carte a entièrement récupéré le binaire de la nouvelle application et l'instant de sa première exécution. Il peut toutefois se passer un certain temps entre ces deux moments.

Une machine virtuelle (VM) est un logiciel reproduisant le comportement d'une machine réelle. L'application s'exécutant dans la machine virtuelle est appelée *invité* (*guest*), tandis que la machine réelle sur laquelle s'exécute la VM est appelée *hôte* (*host*). Le terme « machine » signifie qu'une VM donne l'illusion à *guest* d'évoluer sur un matériel typique mais qui est en fait différent du matériel sur lequel s'exécute *host*. Ce matériel est donc virtuel car il n'existe que du point de vue de *guest*. Il comprend un espace mémoire, des registres virtuels, un jeu d'instructions, etc. à l'image d'une machine Von Neumann classique.
Une VM peut s'insérer dans un système existant à plusieurs endroits. Entre le matériel physique et un système d'exploitation (OS), entre un OS *host* et un autre OS *guest*, ou alors sous un langage dit de haut niveau - dans le sens de « éloigné des problématiques d'accès au matériel ». Les deux premiers types de VM, ou VM systèmes, permettent d'exécuter en même temps plusieurs OS de manière concourante sur une même plateforme matérielle. La dernière, ou VM processus, est un processus utilisateur classique. Ce type de VM émule un système générique et s'assure que *guest* reste isolé du système hôte.
Portabilité Pour un langage de haut niveau, ce type d'approche par VM offrant la vision d'un système complet et « générique » , permet d'assurer une entière portabilité de l'application *guest* sur des machines hôtes hétérogènes. Seule la VM doit être portée et compilée sur l'architecture hôte, ce qui simplifie le développement d'applications multi-plateformes.
Isolation Développée dans un langage sans VM, l'application *guest* serait un processus utilisateur classique. L'utilisation d'une VM permet également d'isoler l'application *guest* du reste des applications s'exécutant sur *host*. Cette approche permet de protéger *host* de toutes fautes logicielles générées par *guest* et notamment les erreurs liées à un mauvais usage de la mémoire (pointeurs nuls, dépassement de capacité, fuites,...). C'est pourquoi, ce type de VM utilise des gestionnaires automatiques de mémoire, libérant le développement d'applications d'une étape complexe et délicate.
Jeu d'instructions Un langage de haut niveau basé sur une VM, comme par exemple le Java, n'est pas compilé dans le jeu d'instructions natif de la machine hôte mais dans une jeu d'instructions intermédiaires propre à la VM (ex: *bytecodes* pour le Java). C'est pourquoi ces langages sont aussi appelés à code semi-compilé. Tel un processeur exécutant du micro-code pour une instruction assembleur, la VM exécute une routine spécifique à une instruction intermédiaire ou *handler*. Cette routine est l'`interprétation` d'une instruction intermédiaire.

Figure 2.4: **Machines Virtuelles**

La vérification ne peut pas se faire *a priori* sur du code partiel, et le système ne peut exécuter que du code vérifié. De plus, le système doit également garantir que la fiabilité soit encore vraie à l'exécution. Telles sont les nouvelles garanties de sécurité que doivent offrir les cartes supportant la *post-issuance*.

2.1.5.2 Machines défensives et analyseurs de code

L'introduction de machines virtuelles (Encadré 2.4) en même temps que la *post-issuance* facilite la résolution de cette nouvelle problématique car le code intermédiaire qu'elles utilisent

est généralement plus facile à analyser que le code compilé. Ce dernier subit en effet les optimisations parfois agressives du compilateur qui entraînent des graphes de flots d'instructions plus difficiles à construire et à suivre. Son autre inconvénient est la quasi-absence de typage des variables une fois compilé qui ne permet pas une analyse stricte du flot d'informations et donc la validation de la bonne utilisation des données qu'il manipule. Beaucoup de travaux se basent donc sur l'analyse d'applications interprétées par une machine virtuelle car comme nous allons le voir par la suite, la connaissance du type est une information primordiale.

Dans le cas des machines virtuelles, la protection contre les applications malveillantes peut se faire sur plusieurs plans, qui peuvent bien évidemment se cumuler. Le premier est la vérification du code intermédiaire [Rose 1998, Deville 2002, Bernardeschi 2008], le deuxième est l'analyse du flot d'informations [Ghindici 2007, Fontaine 2011] et le dernier la mise en œuvre de machines virtuelles dites défensives [Stärk 2001].

L'objectif commun de ces stratégies de sécurisation est de s'assurer que :

1. le programme fait ce qu'il doit faire,
2. le programme ne fait pas ce qu'il n'est pas autorisé à faire.

Cette assurance dépend alors des propriétés à vérifier, qui doivent être définies à l'avance, comme par exemple l'absence de débordement sur la pile d'exécution, le passage de type autorisé à une méthode, la non-divulgation d'un secret, etc.

Voici quelques définitions et une brève description des principaux mécanismes de protection de l'état de l'art :

Un bloc de base est une séquence d'instructions exécutables consécutivement et dont la condition d'arrêt est une instruction de branchement conditionnel ou inconditionnel, ou la fin du programme. La première instruction d'un bloc de base, ou instruction de tête, est quant à elle déterminée de la façon suivante :

1. La première instruction d'un programme est une instruction de tête.
2. Toute instruction pouvant être atteinte par un branchement est une instruction de tête.
3. Toute instruction suivant immédiatement un branchement est une instruction de tête.

Le graphe de flot de contrôle est un graphe orienté utilisé pour représenter tous les chemins pouvant être suivis par un programme lors de son exécution. Dans ce graphe, chaque nœud représente un bloc de base et un arc représente un saut menant d'un bloc de base à un autre. Ce saut et sa destination sont déterminés par une instruction de branchement clôturant le bloc de base. De ce cas, l'instruction de branchement est aussi appelée « `rupteur de flot de contrôle` » .

L'interprétation abstraite formalisée par [Cousot 1977], suit le graphe de flot de contrôle d'une application, à la recherche d'informations sémantiques dans et sur le code. Les différences avec une interprétation classique sont qu'un interpréteur abstrait manipule des types et non pas des valeurs et que son interprétation est sans exécution concrète de code. Dans cette approche, tous les arcs du graphe doivent être parcourus. Comme la terminaison d'un programme est indécidable, l'interprétation abstraite doit donc trouver des conditions de sortie pour espérer se terminer, en cherchant ce qu'on appelle des points fixes (*i.e.* lorsque

les valeurs abstraites[11] des variables découvertes ne changent plus quand l'interprétation revient dans un nœud du graphe déjà visité au moins une fois). Dans l'optique de la protection contre les applications malveillantes, l'utilisation de l'interprétation abstraite est très utile car elle s'approche du vrai comportement d'une application mais sans réellement l'exécuter. Donc en dehors de tout effet de bord qui serait provoqué par l'exécution de code malicieux.

La vérification du code est une analyse par interprétation abstraite du programme à installer pour valider son innocuité avant utilisation. Cette analyse s'effectue méthode par méthode, et sur toutes les méthodes contenues dans l'application. Chaque instruction est déclarée valide si elle respecte la sémantique décrite dans les spécifications du langage (binaire), voir [Lindholm 1999] pour le langage Java. Une méthode n'est déclarée valide que si toutes ses instructions le sont.

En Java, la vérification s'accompagne également d'un contrôle de types, pour s'assurer que les instructions manipulant des objets et des types primitifs le font en fonction des règles de typage du langage. Si nous prenons l'exemple du *bytecode* Java, les types stricts des variables locales ne sont pas tous explicités par le *bytecode* car la pile Java ne contient que des entiers 32 bits ou des références d'objets (elles aussi sur 32 bits).

Il est donc nécessaire de vérifier le code en évaluant au fil des instructions les contenus possibles de la pile d'exécution et en comparant les types découverts. Dans une méthode en cours de vérification, un type peut être placé sur la pile par le retour d'une autre méthode, la vérification doit alors se faire également sur tous les types possibles retournés par cette méthode. De la même manière, les branchements conditionnels à l'intérieur d'une méthode sont autant de chemins dans le graphe de flot de contrôle et autant de versions de pile possibles qui doivent également être vérifiées, une par une, jusqu'à trouver des points fixes et un typage correct et valide.

Preuve accompagnant le code Cette autre technique de vérification de code[12] a été proposée par [Necula 1997], adaptée au Java par [Rose 1998] et implémentée avec succès par [Grimaud 1999] dans une carte à puce. Dans cette approche, une preuve est générée au moment de la compilation de l'application et accompagne celle-ci sur sa cible de déploiement. À l'arrivée, l'installateur de la VM n'a plus qu'à vérifier le code par rapport à sa preuve. Cette approche est très utilisée et souvent recommandée. Elle a néanmoins deux inconvénients. Le premier est la taille de la preuve qui augmente de 10 à 30 % la taille de l'application téléchargée. Le second est l'impossibilité de charger une application fiable si elle ne possède pas de preuve.

L'analyse de flots d'informations consiste à analyser les dépendances entre les entrées et les sorties d'un programme de manière à vérifier qu'il satisfait certaines propriétés de confidentialité et/ou d'intégrité vis-à-vis des données qu'il manipule. L'analyse du flot d'informations a, comme la vérification, besoin de connaître les types. Cette fois, le but est d'identifier qu'une application A ne tente pas d'accéder à une donnée secrète contenu dans B, ou à un niveau plus complexe, que B puisse partager son secret avec C, sans que C ne puisse le partager avec A. Ces contraintes formelles sont les propriétés que l'interprétation abstraite doit identifier en parcourant tous les chemins possibles qui pourraient conduire A à accéder au secret de B.

[11] Union de tous les types possibles que peut prendre une variable
[12] En anglais, Proof-Carrying Code.

Machines défensives À l'inverse des solutions statiques par interprétations abstraites, construire une machine virtuelle défensive consiste à attendre le temps de l'exécution de l'application et à mettre en place des procédures de contrôles actifs de chaque instruction interprétée. Les défenses de la VM ont ainsi une vue nette de l'état applicatif à un instant T pour juger si une action est malicieuse ou non. Dans ce contexte, l'utilisation de machines défensives ajoute un haut niveau de sécurité et divise la complexité de la protection en la répartissant. Mais elles introduisent forcément un surcoût d'exécution en surchargeant les instructions à interpréter, ce qui détériore les performances de la VM. Tandis que l'autre approche par vérification réalisée à l'installation nécessite une charge importante à un instant précis du cycle de vie de la VM puis plus rien.

Status actuel de la *post-issuance* Dans le cas d'applications déployées *post-issuance*, l'enjeu majeur est de par exemple protéger une application bancaire de toutes fuites de données dans un environnement d'exécution qui ne lui est plus dédié. Toutes les solutions et approches présentées dans cette sous-section ont été montrées viables pour le monde des cartes à puce et surtout embarquables. Une carte peut donc être autonome dans la vérification des applications qui lui sont ajoutées.

La principale information qui nous servira par la suite est que l'analyse de code bien que d'apparence complexe peut être utilisée, au prix de quelques efforts, dans une carte à puce. Les travaux sur la sécurité le prouvent. Il est donc tout à fait envisageable d'utiliser cette approche pour réaliser d'autres opérations en-ligne (*i.e.* sur carte).

2.1.5.3 Le défi du stockage

Au-delà de la sécurité, la *post-issuance* est également un défi aux contraintes physiques de la carte à puce. D'une part à cause de son interface de communication relativement lente et d'autre part par sa faible capacité de stockage applicative. Ces deux contraintes obligent alors à avoir autant que possible des applications téléchargeables compactes.

Toutefois, la compacité n'est pas une réponse définitive car cette approche a elle-même sa limite. L'espace de stockage des applications, natives et semi-compilées, est aujourd'hui encore cantonné à un espace mémoire limité. De plus, cet espace de stockage doit être accessible à un coût faible car une machine virtuelle reste un opérateur d'exécution lent. Elle ne peut donc pas être encore un peu plus ralentie lorsqu'elle accède au code qu'elle doit interpréter. C'est pourquoi, les applications téléchargées sont stockées en NOR, l'espace mémoire le plus rapide disponible, et donc à côté du système d'exploitation et de la VM qui y occupent déjà un large espace. Tout ceci limite donc le volume total d'applications téléchargées, dans une mémoire qui n'est pas très large.

2.1.6 Embarquer plus de logiciels dans une carte à puce

En résumé de cette section, il est important de noter que la volonté d'accélérer les processus de mise en circulation de nouveaux modèles de cartes à puce a modifié radicalement la philosophie jusque là attachée à celles-ci. La *post-issuance*, cette capacité de personnaliser le contenu applicatif de la carte, a remplacé l'ancienne philosophie où la sécurité était principalement basée sur un monde fermé et opaque. Certes, la sécurité reste un point primordial pour un objet dont c'est la principale vocation. Nous avons vu toutefois que les failles à combler se sont déplacées en suivant le même mouvement, vers les couches applicatives plus hautes.

Cependant, l'utilisation d'un intergiciel comme une machine virtuelle, placée entre le système d'exploitation et l'application, permet d'avoir à la fois des applications plus faciles

à développer et mais aussi un haut niveau de protection contre les fautes. De plus, les recherches très actives dans le domaine de la sécurité embarquée permettent de plus en plus facilement de valider les impératifs de sécurité requis par de puissants acteurs exigeants comme les vendeurs de cartes bancaires.

Nous avons aussi noté que l'engouement grandissant pour ces cartes « ouvertes » se voit désormais limité par les contraintes inhérentes à une carte à puce et principalement le peu de mémoire dont elle dispose pour le stockage de code exécutable.

2.2 De Java à JavaCard

Dans cette section, nous présentons JavaCard 2.2, la plateforme applicative pour cartes à puce ouvertes la plus répandue. Cette plateforme est basée sur la technologie Java et s'articule donc autour d'une machine virtuelle. Nous commencerons par présenter la technologie Java puis nous verrons comment celle-ci a pu être introduite dans un système aussi petit qu'une carte à puce grâce à la technologie JavaCard.

2.2.1 Le langage Java et sa machine virtuelle

Le langage Java [Gosling 2005] est de nos jours l'un des langages de programmation les plus répandus. Ce langage tire sa popularité des avantages apportés par l'utilisation sous-jacente d'une machine virtuelle.

La devise du langage Java imaginée par ces concepteurs est « *compile once, run everywhere* » [13]. Pour que cette compilation unique et indépendante du système soit possible, le code Java n'est pas compilé en langage machine mais dans un langage intermédiaire qui sera interprété par la machine virtuelle.

Il existe autant de façons de concevoir une machine virtuelle Java (JVM) que de JVM, mais toutes se doivent de respecter la signification propre à chaque instruction, qui sont définies dans les spécifications de la machine virtuelle Java [Lindholm 1999]. En interne, une JVM se compose généralement de plusieurs sous-systèmes qui forment trois grands groupes ; un groupe dédié à l'exécution et l'ordonnancement de code, un groupe dédié à la gestion automatique de la mémoire et un groupe garantissant le chargement dynamique et sécurisé du code.

2.2.1.1 Processus virtuels d'exécution

Interpréteur Le cœur de la machine virtuelle est son interpréteur. Cet interpréteur est une boucle infinie qui récupère, décode et exécute des instructions Java (Listing 6.1, page 100).

Ce *listing* présente la récupération d'une instruction dans un tableau d'octets en mémoire adressable. Cependant, ce code peut être stocké ailleurs, dans une mémoire série par exemple, et peut donc exiger un mode d'accès plus complexe. D'où le terme communément utilisé de « récupération » plutôt que lecture, trop réducteur.

Définition. *Dans un interpréteur, l'étape de* ***décodage*** *consiste à trouver la correspondance entre l'instruction récupérée depuis l'espace de code et la routine implémentant l'action de cette instruction.*

Une fois cette routine trouvée, celle-ci est exécutée et le processus d'interprétation recommence au début après avoir avancé le pointeur d'instruction vers le *bytecode* suivant.

[13] *i.e.* compiler une fois, exécuter (tel quel) n'importe où

Listing 2.1: Implémentation basique d'un interpréteur

```
int pc;
bytecode *code;

while(true){
   bytecode bc= code[pc]; /* Récupère    */
   switch(bc){            /* Décode   */
      case ADD:
      do_add();           /* Exécute */
      pc ++;              /* Incrémente */
      break;
   ...
```

Compilateur Pour des raisons de performances, les machines virtuelles Java modernes embarquent de plus en plus souvent un compilateur de code à la volée [Adl-Tabatabai 1998, Krall 1998, Arnold 2000]. Dans l'absolu, un interpréteur standard est très lent, de 5 à fois 20 plus lent que du code natif. La compilation à la volée permet de transformer et surtout d'optimiser pour la plate-forme cible une instruction ou un groupe d'instructions Java en blocs d'instructions machine directement exécutées par le processeur et non plus par l'interpréteur.

Des travaux comme [Grimaud 1999] ont montré qu'une telle infrastructure pouvait être transposée dans des systèmes aussi petits qu'une carte à puce.

Ordonnanceur L'ordonnanceur de tâches est un autre processus d'exécution commun à bon nombre de machines virtuelles. Il est présent dans une machine virtuelle supportant les processus légers pour répartir la charge et le temps alloué entre des processus applicatifs concourants. Il peut prendre une autre forme lorsque le modèle d'exécution est événementiel comme dans le cartes à puce. Dans ce cas, il active ou désactive une application sur la base d'événements qu'il aura attrapé puis envoyé à l'application destrinatrice. Il peut s'agir d'une interruption matérielle, d'une alarme programmée, la réception d'une APDU, etc.

2.2.1.2 Processus de gestion de la mémoire

Une machine virtuelle permet également une gestion automatique de la mémoire. Automatique signifie que la machine virtuelle ne laisse pas les applications, allouer ou libérer des données en mémoire mais prend en charge cette gestion, évitant ainsi certaines erreurs de développement aux conséquences désastreuses à l'exécution. Deux processus sont chargées de cette mission, un allouer et un collecteur.

L'allouer[14] de mémoire est chargé comme son nom l'indique d'allouer des espaces mémoires à la demande d'une application Java. Cette allocation se fait généralement dans un espace mémoire dévolu et appelé le tas[15]. Lorsque la tas est plein, l'allouer passe la main au collecteur ou « ramasse-miette ».

Le ramasse-miette est le pendant de l'allouer et se charge de la libération de données stockées dans le tas. Généralement, le ramasse-miette fonctionne en deux étapes. La première consiste à identifier toutes les données encore présentes dans le tas qui ne sont plus utilisées. La deuxième étape consiste quant à elle à effectivement libérer la mémoire de ces données inutiles. Un ramasse-miette inclut parfois une troisième étape consistant à

[14] Parfois aussi appelé allocateur
[15] Heap en anglais

compacter le tas en regroupant au début de celui-ci toutes les données encore présentes dans le tas. Ce rassemblement permet de retrouver un plus grand espace contigüe de mémoire libre.

2.2.1.3 Classes, objets et méta-données

La conception et la mise en œuvre d'une machine virtuelle Java se doit de respecter les spécifications du format du fichier binaire exécutable Java, le fichier de classe, et la sémantique de chaque instruction du langage intermédiaire (*bytecode*). Ces spécifications sont définies dans [Lindholm 1999], 2nd Édition. Cet impératif est pris en charge par le processus de chargement de classes.

Le langage Java est un langage orienté *objet* où un *objet* représente un concept de la vie courante ou applicatif, plus ou moins concret. En programmation orienté *objet*, un *objet* est le résultat de l'instanciation d'une classe. Cette classe est la structuration précise du sens que veut donner un programmeur à un *objet* lorsqu'il sera créé à l'exécution. La classe est donc dotée d'attributs représentants les propriétés et l'état de l'*objet*, ainsi que son comportement par le biais de méthodes. Une méthode a principalement trois usages : décrire le comportement de l'*objet* une fois instancié, modifier les propriétés de l'*objet* et permettre son interaction avec d'autres objets ou avec le système.

En conséquence, une classe est l'entité contenant les informations utiles, nécessaires et suffisantes pour que la VM puisse créer une ou plusieurs instances de cette classe lorsqu'elle en reçoit l'instruction. Ces informations constituent les `méta-données` du langage.

*

Définition. *Une **méta-donnée** est une donnée valuée, statique et* a priori *constante fournie par une classe et décrivant un comportement, une propriété et/ou une capacité d'un objet, instanciation de cette classe.*

*

Suivant cette définition, un *objet* est donc une donnée volatile qui n'existe qu'à l'exécution, et une classe est un ensemble de `méta-données` définissant champs, méthodes, et tout autres attributs propres de ces derniers. Dans le fichier de classe, ces méta-données sont reliées entre elles sous formes de références symboliques textuelles.

Dans la technologie Java, les `méta-données` sont acquises par la JVM lors du processus de chargement de classe, préalable à toutes exécutions. Cette étape, ré-itérée pour chaque classe, peut avoir lieu à plusieurs moments, durant l'exécution de la JVM. Cette phase se produit en premier lieu lors du démarrage de la JVM pour charger les classes de base, puis intervient au lancement d'une application pour charger les classes de cette application et leurs dépendances, et enfin lors d'appels explicites au chargement dynamique de classes pendant l'exécution.

L'acquisition des méta-données s'étale sur trois étapes. Ce processus commence par une vérification qui contrôle l'intégrité des méta-données à charger, *bytecodes* inclus, et leur respect des spécifications du langage. Dans une deuxième étape, les références symboliques[16] sont résolues, généralement en adresses physiques, en validant au passage que toutes les dépendances entre classes sont satisfaites. Enfin, la JVM peut engager l'initialisation de certaines données, notamment les champs statiques.

[16] Ce processus est abordé en détail dans notre étude approfondie des `méta-données` section 5.2.1.2, page 78.

Ces étapes ne sont pas nécessairement effectuées immédiatement à la suite des précédentes, et peuvent n'intervenir qu'au moment utile. Toutefois, l'ordre de ces trois étapes se doit d'être respecté. Une fois celles-ci terminées, la classe passe dans l'état « chargé » et est prête à être instanciée sous forme de nouveaux objets, et le code à être exécuté par la JVM.

2.2.2 La plate-forme JavaCard 2.2: Java dans une tête d'épingle

JavaCard [JCVM.2.2.1 2003] est une plate-forme de développement et d'exécution ayant pour objectif de permettre à des applications écrites en Java de s'exécuter sur des cartes à puce. Le grand challenge de JavaCard est d'intégrer une machine virtuelle dans ces matériels contraints, malgré sa réputation d'être gourmande en ressource.

Dans cette section, nous allons aborder les différences qui existent entre les plateformes Java standard et JavaCard. Ces différences se trouvent réparties sur trois niveaux : d'abord au niveau langage, puis au niveau environnement d'exécution, et enfin dans les caractéristiques propres au format binaire de JavaCard.

2.2.2.1 Une Machine Virtuelle scindée

L'environnement d'exécution JavaCard est scindé en deux parties selon une architecture dite Split Virtual Machine[17] (SVM). Nous avons vu que le chargement d'une application Java se déroulait en plusieurs étapes allant du chargement de classes jusqu'à l'exécution du code, en passant par la vérification du *bytecode* ou la résolution de liens symboliques. Dans une SVM, certaines de ces étapes sont réalisées hors-ligne (*i.e.* en dehors de la carte), le reste ayant lieu dans la carte à puce, en-ligne.

Les étapes de chargement de classes sont pour la plupart consommatrices de ressources, notamment mémoire, et sont ainsi effectuées en dehors de la carte, sur une station de travail où la mémoire n'est pas une contrainte. Le premier bénéfice de cette technique est un allégement significatif de la taille et de l'empreinte mémoire de la JVM à embarquer sur la carte à puce. Le second avantage est de permettre une génération agressive du binaire JavaCard pour le compacter avant de le déployer sur la carte .

2.2.2.2 Conversion

Dans le vocabulaire JavaCard2, les deux parties de la SVM sont la `conversion`, exécutée sur station de travail hors-ligne, et `l'interprétation` exécutée sur la carte, en-ligne.

L'interprétation correspond au processus classique d'exécution dans une VM. La conversion quant à elle rassemble la plupart des étapes de chargement de classes, ainsi que le compactage de l'application.

La conversion prend en entrée des fichiers de classes Java au format standard. Ce groupe de classes se limite un paquetage Java complet, et un seul est traité par une conversion. La conversion produit en sortie deux fichiers. Le premier, le fichier CAP (Converted APplet) contient les informations d'exécution : les méta-données Java de toutes les classes et le code de toutes les méthodes du paquetage. Le second, le fichier Export, définit l'interface publique du CAP qui vient d'être converti. Le fichier Export permet les liaisons externes entre fichiers CAP, et donc entre paquetages applicatifs.

La conversion est un processus de chargement de classes dans le sens où les classes converties sont d'abord vérifiées puis transformées dans une représentation propre à la

[17]Machine virtuelle scindée

JVM JavaCard. Enfin l'ensemble est compacté par une résolution précoce de nombreux liens symboliques.

Néanmoins, vérification et résolution ne peuvent pas être terminées hors-ligne.

L'étape de vérification, même complète et fiable, n'est qu'une pré-vérification. En effet, cette vérification hors-ligne ne remplace en rien les principes présentés section 2.1.5.1, page 9, *i.e.* la méfiance qu'une JVM doit garder vis-à-vis des compilateurs et donc la nécessité de recommencer ce travail en-ligne. De son côté, la résolution de liens symboliques lors de la conversion ne peut être que partielle, car les liaisons entre CAP, *i.e.* un paquetage, ne peuvent intervenir qu'en-ligne. C'est en effet le seul endroit où tous les CAP nécessaires à une application sont accessibles et vérifiés.

La conversion est donc d'abord et avant tout une transformation d'un modèle de données - le modèle Java standard -, vers un autre plus compact - le modèle JavaCard. Ces modèles de données, issus de leurs spécifications respectives, seront plus amplement décrits et commentés dans notre chapitre 5, page 73. En substance, nous retiendrons pour l'instant que la conversion est une transformation préalable de méta-données vers d'autres méta-données.

2.2.2.3 Briques logicielles

Pour répondre aux contraintes physiques et au mode de fonctionnement d'une carte à puce, les spécifications JavaCard2 définissent trois briques logicielles embarquées.

- La Java Card Virtual Machine (JCVM) [JCVM.2.2.1 2003] qui adapte le jeu d'instructions Java et son fonctionnement, aux spécificités des applications pour cartes à puce.
- Le Java Card Runtime Environment (JCRE) [JCRE.2.2.1 2003] qui cadre les comportements d'exécution, dont la gestion de la mémoire, la gestion des applications, des événements, ou l'interaction avec le système d'exploitation.
- Enfin, les spécifications JavaCard introduisent une interface applicative (API) dédiée, formée d'un ensemble de classes virtualisant les fonctionnalités d'une carte à puce, notamment le protocole APDU, ou encore les modules d'authentification.

2.2.2.4 JavaCard2 : Un sous-ensemble du langage Java

Le peu d'espace mémoire - mémoire vive ou espace de code - d'une carte à puce laisse peu de place au logiciel de manière générale. Y placer un environnement Java complet, avec sa machine virtuelle, ses bibliothèques de base, son espace de stockage dédié pour les objets créés dynamiquement, ne peut donc se faire sans quelques sacrifices. La plateforme JavaCard2 ne prend donc pas en charge les chaînes de caractères, le support des nombres flottants, et entiers 64 bits ou encore les processus légers. Cependant, nombreuses sont les implémentations de JCRE qui embarquent un mécanisme de gestion automatique de la mémoire capable de collecter et de supprimer des objets Java inutilisés. L'interface applicative est également beaucoup plus pauvre que les versions standards. Elle se limite à quelques classes de base, et à un *framework* définissant une abstraction de la carte à puce.

Le jeu d'instructions est aussi quelque peu différent. Les spécifications de la machine virtuelle Java standard définissent une pile d'exécution sur 32 bits. La JCVM est, elle, définie sur 16 bits. Toutes les instructions manipulant des données sur la pile ont donc des versions 16 bits et 32 bits, ce qui n'est pas le cas en Java standard. C'est lors de l'étape de conversion que cette transformation d'un *bytecode* vers un autre se produit. L'introduction d'une pile 16 bits permet de réduire la taille de celle-ci et donc introduit un gain non négligeable en mémoire vive.

Pour autant, la quasi totalité des *bytecodes* Java sont présents en JavaCard. La plateforme respecte également toutes les notions de la programmation orientée objet : utilisation de classes abstraites, d'interfaces, de polymorphismes, de surcharges de méthodes virtuelles, etc. Mais du code JavaCard ne pourra pas s'exécuter sur une machine virtuelle Java standard, et réciproquement. Des outils de développement spécifiques sont alors également nécessaires pour écrire et tester les applications JavaCard.

2.2.3 Synthèse

JavaCard est devenue *de facto* la plateforme privilégiée pour le développement et l'exécution d'applications téléchargeables *post-issuance* dans une carte à puce « ouverte ». Son approche par machine virtuelle scindée permet une répartition *hors-ligne/en-ligne* de la charge classique d'une JVM et libère ainsi la carte à puce de processus généralement coûteux.

Toutefois, l'utilisation de code semi-compilé interprété et de `méta-données` modifient le modèle d'exécution de code classique que constitue le code machine, reconnu plus rapide. Dans ces circonstances, les applications JavaCard sont elles aussi stockées dans l'espace de code adressable pour garder un minimum de rapidité, bien qu'il n'existe aucun frein d'ingénierie à ce qu'elles puissent être exécutées en place depuis une mémoire série plus lente.

L'usage d'un cache mémoire pourrait d'ailleurs faciliter cette limitation car il représente l'approche performante la plus commune à ce genre de problématique.

2.3 Cache mémoire

Dans cette section, nous présentons un état de l'art sur les caches mémoires. Après avoir décrit leur fonctionnement général, nous aborderons en détail les stratégies qui participent à leur mise-en-œuvre et nous verrons comment l'homme de l'art a approfondi ces stratégies pour en améliorer l'efficacité.

2.3.1 Fonctionnement général

Un cache est un espace mémoire de taille fixe et dédié au stockage temporaire de données issues d'une autre mémoire (ou mémoire secondaire), réputée plus lente. Il est généralement le regroupement d'une stratégie de placement des données à l'intérieur de l'espace temporaire, et de deux algorithmes : un algorithme pour la recherche d'information dans le contenu temporaire, un algorithme de renouvellement de ce contenu [Smith 1982]. Ces algorithmes forment le `gestionnaire de cache`. Dans le cas d'un cache d'instructions, le contenu du cache est une copie partielle et généralement désordonnée du binaire de une ou plusieurs applications.

Le premier des deux algorithmes d'un cache est chargé de la recherche d'une information - instruction ou donnée - à partir d'une adresse qui lui est fournie et qui correspond à un emplacement dans la mémoire secondaire (Fig. 2.5, étape 1). Cet algorithme est l'interface externe du cache et est exécuté à chaque demande de donnée. La caractéristique principalement attendue de l'algorithme de recherche est sa vitesse, car son objectif majeur est de servir le plus vite possible une requête.

Le second algorithme est chargé du renouvellement des informations contenues dans le cache. Cet algorithme est exécuté dans le cas d'un échec du premier à trouver l'information demandée. Le contenu du cache doit alors être renouvelé pour intégrer les informations manquantes. Le renouvellement consiste à récupérer les nouvelles données depuis la mémoire secondaire (Fig. 2.5, étape 4), et choisir un emplacement dans l'espace de stockage

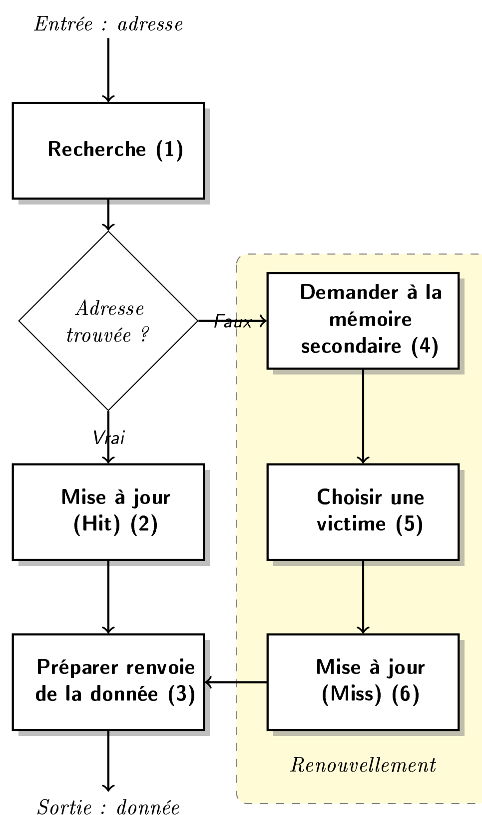


Figure 2.5: Fonctionnement général d'un cache

en évinçant une donnée temporaire déjà présente (Fig. 2.5, étape 5). La caractéristique principalement attendu de l'algorithme de renouvellement est donc de gérer au mieux le contenu du cache pour éviter un maximum d'échec de la part de l'algorithme de recherche.

Généralement, chaque algorithme utilise sa propre structure de données lui servant à mémoriser des informations et/ou des états lui permettant d'effectuer son traitement. Les étapes 2 et 6 correspondent aux mises à jour de ces structures de données, si elles existent. La terminologie des caches appelle l'étape 2 un *Hit*, *i.e.* un succès dans la recherche impliquant que la donnée recherchée était déjà présente en cache. Un échec de cette recherche est appelé *Miss* ou défaut de cache. Ce *Miss* est la principale complication dans l'efficacité globale du cache car il entraîne une lecture dans la mémoire secondaire, provoquant un temps d'attente long. Ce temps d'attente de la mémoire secondaire est appelé une *pénalité*.

2.3.2 Stratégies de placement

Une stratégie de placement a pour but d'optimiser l'espace de stockage temporaire. La stratégie mise en œuvre doit servir à maximiser le nombre de données pouvant résider en cache, en organisant aux mieux les données dans cet espace. Cette maximisation dépend de l'unité de stockage dans le cache qui est utilisée par la stratégie de placement.

Définition. *Une `unité de stockage`, du point de vue de la stratégie de placement, est un bloc d'octets de taille fixe ou variable qui correspond à la plus petite unité de `réservation d'espace` dans l'espace de stockage temporaire du cache. L'unité de stockage est donc une `valeur` définissant la taille d'un `bloc mémoire` dans le cache.*

Par analogie, l'unité de stockage est généralement aussi la plus petite décomposition de `données échangées` entre le cache et la mémoire secondaire par le gestionnaire de cache. Cette unité de stockage ne doit pas être confondue avec l'unité de stockage propre à la mémoire secondaire, par exemple une page pour la Flash NAND. En effet, la lecture dans une page de Flash NAND renvoie un bloc de données d'une certaine taille, qui peut ne pas correspondre à la taille de l'unité de stockage dans le cache. Charge alors au gestionnaire de cache de trouver au sein de la page de Flash lue, le ou les blocs de données qu'il souhaite réellement mettre en cache et qui correspondent à une ou plusieurs unités de stockage du cache.

Dans la pratique, les stratégies de placement sont généralement implémentées selon deux méthodes : l'allocation et la segmentation.

2.3.2.1 Placement par allocation

Derrière le terme « allocation » se cachent plusieurs types d'algorithmes aux hypothèses de départ différentes. La définition que nous utiliserons dans ce document est celle de Wilson *et al.* dans un papier de veille très complet sur le sujet [Wilson 1995].

Définition. *Un allocateur est un gestionnaire dynamique de mémoire dont le seul propos est d'allouer ou dés-allouer des blocs de mémoire, en gardant une trace de quelle partie de la mémoire est utilisée ou libre.*

Suivant cette définition, un allocateur ne connait pas le contenu d'un bloc mémoire alloué. Ce contenu est donc géré par le composant appelant l'allocateur. Le moment et la manière dont le bloc mémoire alloué est alimenté, dépend des objectifs de ce composant. De la même manière, la libération ne peut intervenir que sur demande explicite de ce même composant. Un exemple d'allocateur répondant à cette définition est l'interface *malloc/free* de la bibliothèque standard C.

L'avantage de l'allocation est qu'elle peut travailler sur des unités de stockage de taille fixe ou de taille variable. Dans le cas d'un cache d'instructions, l'unité de stockage correspond généralement à un bloc de base. Ce qui garantit que le cache ne contiendra jamais d'instructions inutiles, car toutes auront au moins servi une fois, évitant ce que l'on appelle la pollution (*i.e.* l'obligation de maintenir en cache des données inutiles, accaparant de l'espace pour rien). L'allocation dynamique est toutefois confrontée à deux problèmes majeurs pour maximiser la quantité de données résidant en cache. Le premier est la ré-utilisation de l'espace libre et le second le taux d'occupation.

Fragmentation La fragmentation est un phénomène collatéral à l'utilisation de blocs mémoires de tailles variables. Dans le cas d'un cache, elle survient suite à des remplacements successifs de blocs à divers endroits de l'espace de stockage. Ce qui occasionne la formation de "trous" dans l'espace de stockage, *i.e.* un espace libre entouré d'espaces occupés. C'est cet état morcelé que l'on appelle fragmentation, facteur dégradant le taux d'occupation. Cependant, le vrai problème survient lorsque ces trous deviennent trop nombreux et trop petits pour satisfaire des nouvelles demandes d'allocation pour des blocs plus gros.

La fragmentation est donc potentiellement néfaste puisqu'elle réduit l'espace total utilisable par la stratégie de placement. Elle est un problème connu et abondamment étudié depuis de nombreuses années, que ce soit dans le domaine des systèmes de fichier, des bases de données ou des gestionnaires mémoires de systèmes d'exploitation. Elle est généralement résolue en partie par diverses techniques. Nous en retiendrons deux, qui sont les plus utilisées.

La première est triviale et consiste à fusionner des blocs libres adjacents, ce qui a pour effet de recréer des blocs plus gros, augmentant leurs chances d'être réutilisés.

La seconde technique est le compactage de l'espace alloué. Elle consiste à regrouper tous les blocs alloués à un même endroit de l'espace de stockage, en début ou en fin, pour ainsi n'avoir plus qu'un seul et unique bloc cumulant tout l'espace libre. Cette technique est très utilisée dans les ramasse-miettes [Wilson 1992, O'Toole 1993, Chen 2003] de bons nombres de machines virtuelles orientées langage. Elle est très complexe à mettre en œuvre, surtout lorsque le compactage doit se faire en place, sans espace de traitement temporaire supplémentaire, et donc sans pouvoir bouger la plus part du temps tous les blocs libres en une seule passe.

Gestion de l'espace libre Dans un allocateur, la gestion des espace libres est généralement basée sur une liste chaînées de blocs libres. La ré-utilisation de ces bloc libres se fait par différents algorithmes parcourant cette liste [Bays 1977]. Les principaux sont :

- *First fit* qui choisit d'utiliser le premier bloc libre suffisamment grand pour contenir la demande d'allocation,
- *Best fit* qui continue au contraire la recherche jusqu'à trouver le plus petit bloc libre pouvant contenir la demande.

Le premier est plus rapide mais introduit plus rapidement de la fragmentation. Ce qui est l'inverse pour le second, sans toutefois pouvoir garantir l'absence totale de fragmentation.

Blocs de taille fixe Le meilleur moyen d'éviter la fragmentation est l'utilisation de blocs de taille fixe en tant qu'unité de stockage. Car la fragmentation est par nature créée par l'allocation/dés-allocation de blocs de tailles trop disparates. Les blocs de tailles fixes résolvent également les problèmes adressés par *first fit*/*best fit*, simplifiant ainsi la réutilisation de l'espace libre.

Aligner l'unité de stockage sur une taille pré-définie[18] de 8, 16 ou 32 octets par exemple, permet d'assurer que si il existe un bloc libre dans l'espace mémoire, il est forcément de la taille demandée.

Dans cette approche, l'instruction de tête du bloc de base reste communément la première donnée de l'unité de stockage. Par contre, l'avantage à pouvoir stocker aisément des blocs de base se perd. En effet, si le bloc de base est plus gros que l'unité de stockage, il doit être scindé en autant de blocs de la taille d'une unité de stockage. Cette approche présente donc un inconvénient : si le bloc de base est plus petit que l'unité de stockage, le bloc de cache doit être « bourré » soit avec des instruction nulles (NOP pas exemple) soit d'autres instructions spécifiques garantissant la continuité fonctionnelle du programme, par exemple un saut inconditionnel vers le morceau de bloc suivant.

De nombreux travaux existent sur la ré-écriture de blocs de code pour ce genre d'usage, soit par une approche dynamique à l'exécution [Bala 2000, Desoli 2002, Verma 2004], ou alors à la compilation statiquement [Banakar 2002, Angiolini 2004, Miller 2006]. L'avantage de cette approche est donc l'absence de fragmentation, la simplification de l'algorithme d'allocation et un taux d'occupation optimale des espace libres. Les inconvénients sont au nombre de deux : l'augmentation de la taille du binaire exécutable (jusqu'à 30 % dans les travaux cités), et surtout la même proportion de pollution amenée par le bourrage. Ce phénomène est clairement le plus indésirable car en réduisant le nombre de données utiles, il dégrade fortement l'efficacité du cache.

[18] Plutôt que de parler de bloc de taille fixe, la littérature utilise souvent le terme de ligne lorsqu'il s'agit de cache matériel ou plus généralement le terme de page, par analogie aux mémoires virtuelles dites paginées

2.3.2.2 Placement par segmentation

La segmentation est une technique ancienne déjà utilisée depuis les années 70 dans les mémoires virtuelles des systèmes d'exploitation. Elle consiste à découper le binaire exécutable en blocs de taille fixe mais cette fois indépendamment du contenu de ces blocs, ici appelés `segments`, et eux même composés d'une ou plusieurs `pages`.

Les mémoires Flash accessibles par pages sont des mémoires segmentées. Dans celles-ci, copier et stocker des binaires exécutables revient déjà à appliquer une segmentation sur leur code. L'adresse d'une instruction dans un de ces binaires est donc l'adresse du segment plus un offset dans ce segment. Ce qui est exactement la commande matérielle envoyée à une Flash NAND pour lire une donnée.

Appliquer à une stratégie de placement en cache, la segmentation consiste généralement à transposer une page ou un segment et une unité de stockage dans le cache. Au niveau du placement, la segmentation fonctionne donc comme l'allocation par blocs de taille fixe et offre donc les mêmes avantages : absence de fragmentation et utilisation de l'espace avec un taux de remplissage optimal. La différence fondamentale est que l'instruction de tête d'un bloc de base n'est plus forcément la première donnée d'un bloc de cache. De plus, ce bloc peut contenir plusieurs blocs de base, mais également des blocs de base tronqués. L'inconvénient souvent reproché à la segmentation [Miller 2006] est donc son manque de discernement par rapport au contenu qu'elle véhicule, et par effet de bord, la quantité de données superflues récupérées et mises en cache. Superflues car un bloc de cache peut alors contenir des données qui ne seront jamais utilisées, augmentant la pollution jusque dans des proportions très élevées.

Toutefois, cet inconvénient est difficile à évaluer *a priori*. On constate en pratique que la segmentation peut entraîner deux phénomènes radicalement opposés qui peuvent se résumer ainsi :

- plus un segment est large, plus la probabilité d'y trouver plus tard des données utiles est élevée, et ce sans le vouloir forcement au départ ;
- plus un segment est large, plus la probabilité qu'il contienne de la pollution est élevée

Ces deux probabilités sont difficilement mesurables hors-contexte car elles sont liées à de nombreux paramètres, dont le principal est la structure des binaires exécutables eux-même. Puisque c'est d'elle que dépend la façon dont les blocs de base sont répartis entre segments.

2.3.3 Stratégies de renouvellement

Le renouvellement du cache est constitué des trois étapes entourées de jaune dans la figure 2.5. Il intervient lorsque le cache ne contient pas la donnée recherchée. Dans ce cas, celle-ci doit être récupérée depuis la mémoire secondaire. Lorsque le cache n'est pas plein, un nouveau bloc de cache est alloué et le bloc de mémoire secondaire y est copié. Une stratégie de renouvellement intervient lorsqu'aucun espace libre n'est disponible dans la cache. La seule démarche possible est alors de supprimer un ou plusieurs blocs de cache pour les remplacer par un nouveau bloc contenant les données manquantes.

Ce point central du renouvellement est géré par ce qui est appelé dans l'état de l'art une politique de remplacement.

Le renouvellement est la plus coûteuse étape dans la vie d'un cache puisqu'elle nécessite un accès à la mémoire secondaire, réputée plus lente. Pour réduire ces accès indésirables, il faut choisir minutieusement une « victime » à évincer parmi les données présentes en cache.

La solution la plus triviale est d'effectuer un remplacement aléatoire. Paradoxalement, elle n'est pas beaucoup moins efficace que des politiques beaucoup plus élaborées

[Al-Zoubi 2004]. Même si elle est rarement utilisée dans des contextes réels. La politique de remplacement qui a été prouvée comme étant la politique optimale, c'est-à-dire celle qui déclenchera le moins de défauts de cache, a été proposée par [Belady 1966] et est souvent connue sous le nom de MIN ou OPT. En réalité, cette politique n'est concrètement pas utilisable car elle se base sur une connaissance de l'avenir, ce qui lui permet de toujours faire les bons choix.

Tout comme MIN, la plupart des politiques de remplacement se basent sur des notions temporelles comme la fréquence et/ou la récence et l'élaboration d'un algorithme de remplacement réside souvent dans la manière dont seront triées et donc organisées les informations.

La politique de remplacement la plus communément utilisée est l'algorithme LRU (Least Recently Used). LRU se base sur une liste triée sous forme d'historique des accès au cache. Cette liste aura en première position la donnée la plus récemment utilisée, et en dernière position la donnée la plus anciennement utilisée. Lors d'un défaut de cache, LRU choisira automatiquement la donnée la plus ancienne comme victime. La complexité de cet algorithme réside dans le tri constant de son historique car à chaque accès au cache, l'entrée contenant la donnée qui vient d'être utilisée doit être repositionnée en tête, pour que la liste reste toujours cohérente. Du fait de cette complexité, de nombreuses approximations de l'algorithme LRU ont été proposées (CLOCK [Tanenbaum 2001], ou WSCLOCK [Carr 1981]), dédiés pour des implémentations matérielles. Dans le cas d'implémentations logicielles, comme par exemple des gestionnaires de base de données ou des serveurs Web, cet algorithme est le plus souvent utilisé dans son implémentation classique, sous forme de listes simplement ou doublement chaînées.

2.4 Synthèse

Déployer des applications *post-issuance* dans une carte à puce est encore fort peu connu du grand public. Cependant, l'état de l'art sur les nouvelles problématiques que cela introduit est déjà riche de solutions et d'enseignements.

Concernant la sécurité, il est très instructif de constater que des d'algorithmes d'analyse statique de code sont très souvent utilisés et viables. Alors que ces algorithme sont souvent considérés comme gloutons en mémoire vive et en temps de calcul. Ces possibilités laissent donc envisager de pouvoir utiliser sans crainte l'analyse statique de code et l'interprétation abstraite en-ligne pour d'autre contexte que la sécurité.

L'état de l'art montre ainsi qu'un modèle applicatif et génie logiciel complet, comme la plateforme Java et la programmation orientée objet, peut être transposé de l'informatique standard au monde miniature et sensible des cartes à puce. La technologie JavaCard apporte la souplesse et la ré-utilisabilité du développement Java. Tandis que l'usage d'une Machine Virtuelle assure portabilité, robustesse et isolation. C'est-à-dire les conditions requises pour marier *post-issuance* et sécurité.

Toutefois, le volume d'applications reste limité de part les contraintes mémoires de la carte à puce, mais aussi, comme nous le pensons, à cause de pistes qui n'ont pas encore été suffisamment explorées. Il est en effet tentant d'étendre l'espace de stockage de code en utilisant la mémoire secondaire disponible malgré sa lenteur, *i.e.* la Flash série.

Le chapitre suivant est consacré à la description de ce défi pour ensuite proposer une nouvelle approche utilisant les ressources et enseignements de l'état de l'art sur les caches mémoires et l'analyse statique de code en-ligne.

Chapitre 3

Problématique

Rendre les cartes à puce personnalisables ouvre vers de nouveaux usages. Dans ce contexte, une solution séduisante pour aller encore plus loin consiste à utiliser une mémoire à bas coût pour permettre à la carte de contenir encore plus d'applications. Cette mémoire est déjà disponible dans la plus part des cartes mais n'est pas adaptée a priori *pour ce genre d'utilisation de part son temps de latence important. Cet état de fait constitue notre problématique, dont les tenants et aboutissants sont démontrés dans ce chapitre. Nous présentons d'abord les avantages à étendre l'espace de stockage de code et l'intérêt que peut apporter une mémoire non-adressable dans ce contexte. Nous poursuivons avec les inconvénients et limites de cette approche en terme d'efficacité pour poser les bases de notre challenge. Pour terminer, nous listons diverses approches possibles pour résoudre cette problématique, dont une approche nouvelle que nous apportons et qui sera développée dans la suite de ce document.*

3.1 Contexte et opportunités

3.1.1 *Post-issuance* massive

Un des challenges actuels de la *post-issuance* est la capacité de stockage de nouvelles applications dans une carte à puce. Celle-ci est en effet limitée par une relativement faible quantité de mémoire de code disponible dans une carte à puce. Les velléités pour fusionner cartes SIM, cartes bancaires, ou identités, ou la volonté de construire des socles applicatifs plutôt que des systèmes complets et fermés nécessitent à terme d'augmenter cette capacité de stockage. Cependant, la solution évidemment la plus simple - modifier le matériel dans ce sens - est freinée par une problématique rédhibitoire dans l'industrie de la carte à puce: le coût unitaire de production. Augmenter la quantité de silicium pour étendre la capacité mémoire du micro-contrôleur est donc une solution à écarter.

D'autant que les matériels actuels possèdent déjà une autre mémoire, série, souvent de la Flash NAND, qui fournit beaucoup plus d'octets de stockage que la mémoire interne adressable. Son utilisation pour agrandir l'espace de code à coût de production constant est donc indéniablement une opportunité à saisir.

L'inconvénient majeur de la Flash NAND réside dans sa caractéristique d'être à la fois non-adressable et paginée qui introduit un temps de latence élevé. Non-adressable, elle n'est pas accessible directement par le processeur pour lire le code à exécuter. Paginée, elle n'est pas accessible octet par octet mais par bloc et implique donc un temps de latence important et gênant pour réaliser des accès aléatoires comme l'exige l'exécution de code.

Dans cet objectif, de nombreux travaux ont vu le jour pour rendre la Flash NAND exécutable en place, en modifiant son interface matérielle pour la rendre accessible par octets [Park 2003a, Park 2003b, Joo 2006, Lin 2007, Chang 2010, Baiocchi 2011].

Définition. ***L'exécution en place*** *est une méthode consistant à exécuter un programme directement depuis là où il est stocké, sans passer par une copie de celui-ci en mémoire*

principale.

Ces travaux bien qu'intéressants n'en restent pas moins indisponibles sur le marché et ne changent donc pas le problème de latence de la Flash NAND série dans une carte à puce.

Le défi ainsi proposé par la Flash NAND pour l'exécution en place est néanmoins intéressant car le verrou ne se situe pas dans la mise en œuvre mais dans la recherche d'efficacité. Quoiqu'il arrive, il est possible d'utiliser la Flash pour exécuter du code en place. Le vrai challenge se situe dans la manière de rattraper la performance perdue, en passant d'une exécution en place depuis la Flash NOR[1] à l'exécution en place depuis une mémoire série comme la Flash NAND.

3.1.2 Bénéfices potentiels

Le premier bénéfice à résoudre ce challenge est naturellement d'ouvrir massivement le champ applicatif. Étendre la capacité de stockage d'applications dans une carte à puce peut ouvrir vers une nouvelle ère de la post-issuance où une carte deviendrait massivement multi-applicative. Une technologie comme JavaCard permet déjà d'intégrer plus facilement de nouvelles applications dans une carte à puce après sa mise en circulation. Toutefois les contraintes matérielles d'une carte à puce, notamment le quantité de mémoire adressable, constitue toujours une limite à cette évolution en cours.

Pourtant, une JVM comme celle de JavaCard n'a pas réellement besoin que l'application qu'elle exécute soit stockée dans l'espace d'adressage du CPU. Son code n'étant pas fait de pointeur, elle pourrait se passer d'un stockage en NOR, comme actuellement. Toutefois, le temps d'accès au code reste une considération majeure pour un outil déjà réputé plus lent par rapport à un programme natif compilé. C'est pourquoi jusqu'à présent, les binaires JavaCard se trouvent encore dans la mémoire persistante disponible la plus rapide, à savoir le Flash NOR.

De son côté, le code natif compilé est difficilement chargeable à chaud et surtout dynamiquement, bien que des solutions existent à cette échelle [Dunkels 2006, Gu 2006]. Mais JavaCard est depuis le début, grâce à sa VM, un moyen simple de contourner ce problème. Dans le contexte JavaCard, pourquoi alors s'intéresser également à l'exécution en place de code compilé malgré la barrière de l'in-adressabilité de la Flash NAND ?

Une des fonctionnalités des machines virtuelles standards déjà présentée est la compilation de code à la volée ou juste à temps (JIT[2]). Cette technique consiste à recompiler dynamiquement du code semi-compilé vers du code natif dans le jeu d'instructions propre au processeur hôte. Car le code natif est selon les mesures de l'état de l'art de 5 à 20 fois plus rapide que l'interprétation logicielle de code semi-compilé [Hoogerbrugge 2000]. Cet outil est très souvent écarté dans le contexte des cartes à puce, car un compilateur JIT implique trois défis :

1. le coût de l'analyse dynamique du code pour isoler les points chauds (*i.e.* les bouts de code suffisamment pertinents pour la re-compilation coûte moins chère que l'interprétation),

2. générer dynamiquement le code natif en embarquant un compilateur,

3. être capable de stocker le code généré dans un endroit suffisamment grand pour accueillir un volume conséquent et beaucoup plus large que le code semi-compilé.

[1] Ou toutes autres mémoires adressables.

[2] Just-In-Time

Les deux premier défis ont été adressés avec succès par [Grimaud 1999], mais le troisième reste encore ouvert.

Ce document adressent indirectement ce troisième point : exécuter efficacement du code stocké dans une mémoire non-adressable, qui plus est suffisamment large pour accueillir du code compilé à la volée.

Un dernier bénéfice potentiel se situe dans les approches possibles qui peuvent être suivies. L'approche classique pour « accélérer » une mémoire secondaire lente est de mettre en place un système de cache mémoire. Une telle approche, adressée en pur logiciel, permet d'explorer des pistes qui ont depuis longtemps été abandonnées dans le domaine des caches d'instructions matériel par exemple. Ce type de cache sert directement la problématique de l'exécution de code depuis une mémoire aux propriétés inadaptées vers un opérateur d'exécution donné. Ce qui correspond à notre sujet.

Cependant, les caches matériels sont optimisés pour suivre la cadence imposée par le processeur et offrent une structure extrêmement figée et rigide par rapport au contenu. Il reste donc avec l'espoir de pouvoir fonctionner de manière adéquate au moins dans la plupart des cas [Smith 1982]. De nombreux sacrifices sont ainsi réalisés sur les stratégies de placement. Les stratégies d'éviction quant à elles n'y sont que des approximations, bien souvent de LRU, ou tout bonnement inexistantes. Les algorithmes de recherche sont tellement réduits à l'essentiel qu'ils impactent directement le mode de stockage et réduisent ainsi le panel de solutions.

Du logiciel peut représenter ici un gain - à mesurer - en flexibilité et en richesse, jusqu'à permettre même de proposer une toute autre approche qu'un cache.

3.2 Problématique : le mur des temps de latence

3.2.1 NAND versus NOR, le couperet de la réalité

Malgré des caractéristiques techniques et des performances décrites par les constructeurs comme quasiment équivalentes, la Flash NAND et la Flash NOR montrent des résultats bien différents lorsqu'elles sont accédées dans des conditions réelles. Nous allons voir ici que l'écart est surtout important lorsqu'il s'agit d'utiliser la NAND pour des lectures aléatoires comme dans le cas qui nous occupe.

Il est communément acquis que l'exécution de code implique beaucoup plus de lectures aléatoires au sein du fichier binaire exécutable que d'accès séquentiels comme lorsqu'il s'agit de lire un fichier de données : texte, image, son ou vidéo. La NOR est réputée très efficace pour la lecture aléatoire de données, de part son interface orientée octet. La NAND est de son côté réputée efficace pour les lectures séquentielles car son mode d'accès par page se prête extrêmement bien aux lectures groupées. L'écart qui existe entre ces deux mémoires se mesure de la manière suivante :

- À une fréquence de 20 MHz, une Flash NOR lit une donnée par cycles de 50 ns. Elle aura donc une bande passante maximale théorique de 19,07 Mo/s (ou débit crête). Ce débit est identique pour des lectures aléatoires et pour des lectures séquentielles (tableau 3.1, page 28).
- À la même fréquence (tableau 3.2, page 28), une Flash NAND charge d'abord une page de 2048 octets et 64 octets de données de contrôle dans son registre de transfert en $25\mu s$. Suite à cela, un octet est lisible séquentiellement toutes les 50 ns, ce qui donne un total de $130.9\mu s$ pour lire une page complète. La bande passante maximale théorique de la NAND est donc de 0,0073 Mo/s pour des lectures aléatoires et 14,92 Mo/s pour des lectures séquentielles.

En réalité les chiffres bruts des étapes de chargement et de récupération ne sont pas les seuls à prendre en compte. Ils excluent en effet le temps d'exécution du contrôleur de la NAND qui est responsable, d'abord de traiter les commandes matérielles qu'il reçoit du pilote et enfin de calculer et vérifier les codes de contrôle d'erreurs. Le tout est imbriqué enfin dans le temps d'exécution du pilote, logiciel, qui lui est aussi un temps incompressible. Dans ce cadre étendu, le temps de chargement d'une page que nous avons constaté sur du matériel existant repousse le temps total, hors pilote, à 145,6μs dans le meilleur des cas - abstraction faite des dérives d'horloge constatées - et 198.4μs dans le pire des cas. Les bandes passantes respectives sont reportées dans le tableau 3.3 (page 28), lignes une et deux.

Récupération	Lectures aléatoires	Lectures Séquentielles
50 ns	19,07 Mo/s	19,07 Mo/s

Tableau 3.1: Débit d'une NOR, cadencée à 20 MHz, sur la base de données constructeurs

Taille de page	Chargement	Récupération	Page complète	Lectures Aléatoires	Lectures Séquentielles
2048 + 64	25 μs	50 ns	130,9 μs	0,0073 Mo/s	14,92 Mo/s

Tableau 3.2: Latence et débit crête d'une NAND typique, cadencée à 20 MHz, sur la base de données constructeurs

	Chargement	Récupération	Page complète	Lectures Aléatoires	Lectures Séquentielles
Meilleur	25 μs	50 ns	145,6 μs	0,0065 Mo/s	13,41 Mo/s
Plus mauvais	25 μs	75 ns	198,4 μs	0,0048 Mo/s	10,04 Mo/s

Tableau 3.3: Latence et débit crête d'une NAND cadencée à 20 MHz, à partir de mesures obtenues dans des conditions réelles, et incluant le contrôleur.

	Lectures aléatoires	Lectures séquentielles
Meilleur	2933	1,46
Plus mauvais	3972	1,89

Tableau 3.4: NAND cadencée à 20 MHz : facteur de ralentissement par rapport à la NOR

Pour aller encore plus loin dans l'observation de l'écart réel qui existe entre une mémoire adressable et une mémoire non-adressable, le tableau 3.4 donne le facteur de ralentissement calculé à partir des bandes passantes présentées dans les tableaux précédents. Le constat est sans appel car dans le meilleur des cas en environnement réel, une mémoire non-adressable comme la flash NAND est 2933 fois plus lente qu'une mémoire adressable, dans l'exercice des lectures aléatoires.

3.2.2 Insuffisance de l'approche par tampon

L'exécution en place, dans sa stricte définition, n'est pas vraiment adaptée à une mémoire paginée comme la Flash NAND. Par contre, cette NAND étant accédée à travers une FTL comme cela a été présenté dans notre état de l'art, la lecture d'instructions peut se faire en

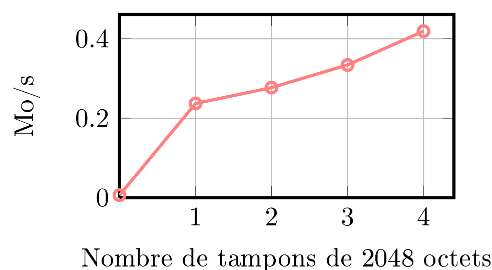


Figure 3.1: Exécution en place sur une puce à 20 MHz, avec des tampons de 2048 octets.

utilisant le tampon de cette FTL ou un tampon dédié.

Indépendamment de savoir comment et quel tampon utiliser, la figure 3.1 présente quelques résultats sur un petit programme de test de quelques centaines de milliers d'instructions seulement. Celui-ci est stocké en Flash NAND.

Dans ce test, ce programme est exécuté en place en utilisant de 1 à 4 tampons, de tailles égales à une page de Flash NAND et stockés en mémoire vive. On remarque d'emblée que bien qu'en sacrifiant déjà 8Ko de mémoire vive, le débit arrive péniblement à 0,42 Mo/s. Soit encore plus de 47 fois plus lent que la NOR. Sans compter le coût de gestion : des données, de leur mode de stockage ou de leur accès effectif et répété à l'intérieur de cet espace temporaire.

Dans ces conditions, comment faire alors pour lutter contre ces temps de latence et faire en sorte que le coût de récupération d'une instruction stockée dans une mémoire non-adressable et paginée ne soit plus prohibitif par rapport à son exécution pure ?

3.3 Approches

La définition de l'exécution en place laisse penser que le seul vrai problème est de rendre accessibles les données contenues dans la mémoire série. Cependant, cet objectif est simple à réaliser puisque un tampon permet de l'atteindre. Une fois le problème d'accessibilité résolu intervient comme un défi, le vrai problème qu'est l'efficacité. Ne pas atteindre ce stade rend tout simplement la solution non-viable, même si elle fonctionne.

Nous listons ici différentes approches possibles pour résoudre cette problématique, en présentant leurs avantages et leurs inconvénients.

3.3.1 Première approche : un cache mémoire

La solution de l'état de l'art qui s'impose tout naturellement et comme une évidence est celle de mettre en place un cache. Un cache est en effet reconnu comme une approche plus élaborée et plus efficace qu'un tampon. Son usage est d'ailleurs généralisé, chaque fois que surviennent des problématiques d'accès performants à des données d'une mémoire secondaire. La plupart du temps, un cache sert à réduire les temps d'accès à une mémoire à fort temps de latence, comme par exemple un disque dur. Ou lorsque la latence d'une mémoire, même faible, devient néanmoins un handicap à long terme. C'est le cas exemple des processeurs évolués face à la RAM.

3.3.1.1 Un cache d'instructions en logiciel

Plusieurs approches purement logicielles ont été proposées pour utiliser un cache comme passerelle d'accès à du code et des fichiers exécutables stockés dans une mémoire secondaire

lente.

Entrelacé dans le code binaire Une première façon de procéder consiste à intégrer un cache et la gestion de la mémoire secondaire - par exemple la NAND - dès la compilation de l'application [Park 2004, Kim 2011]. Bien que cela fonctionne, les inconvénients à gérer cette intégration à la compilation sont :

- l'augmentation significative de la taille du binaire,
- le peu de souplesse à avoir un cache par programme compilé,
- et un besoin en RAM d'au moins la moitié du binaire dans les solutions proposées par Park *et al.* et Kim *et al.* - avec des cas démontrés où le cache doit être quasiment aussi gros que ce binaire.

Mémoires Scratchpad Notre approche par cache d'instructions en logiciel ressemble en partie à un autre domaine, celui des mémoires Scratchpad[3] [Panda 2000, Banakar 2002, Angiolini 2004]. L'idée est ici de supprimer la logique câblée des défauts de cache des caches processeurs (caches de premier niveau, d'instructions et/ou de données), pour avoir un espace de stockage alors plus vaste en terme de silicium. Le « cablage » du cache est remplacée par une implémentation logicielle intégrée au micro-code du CPU. Pour des raisons de performance et de part leur proximité avec le processeur, l'allocation dans les mémoires Scratchpad est souvent déterminé à la compilation, ce qui représente une approche très rigide quant au contenu et élimine la stratégie de placement par segmentation. D'autre travaux proposent une approche plus dynamique [Kandemir 2001, Miller 2006, McIlroy 2008] où le compilateur insère cette fois des routines plutôt que des directives, ce qui laisse un peu plus de latitude quant au contenu de la mémoire à un instant T, mais dégrade les performances. Les Mémoires Scratchpad correspondent donc plus à des caches de données critiques, parfois hétérogènes (*e.g.* blocs de code « chaud » , copies de piles de processus, données produites par le CPU comme le déroulement de boucles, etc).

Imiter l'approche classique Une autre approche plus conventionnelle consiste à concevoir un cache d'instructions en s'inspirant des caches de données classiques où un cache est un composant logiciel indépendant placé entre la mémoire secondaire et un consommateur de données, comme une VM, un serveur web, une gestionnaire de base de données ou un OS. Ce type de configuration n'est pas directement lié aux données et son principe de base est la transformation d'une adresse logique en une adresse physique afin de renvoyer une copie de la donnée cherchée. Cette approche rend le cache indépendant des données en mémoire non-adressable mais également indépendant du type de données qu'il contient lui-même. Cette approche apporte également un plus haut degré de flexibilité, car si le cache est conçu pour, il peut alors très bien être reconfiguré à chaud pour s'adapter à un autre besoin. Généralement un cache logiciel de ce type utilise un placement par segmentation reposant sur la pagination à la demande [Park 2003a, Park 2004, Joo 2006].

3.3.1.2 Les clés de l'efficacité d'un cache

Les taux de *Hit* et de *Miss* sont les indicateurs les plus couramment utilisés pour estimer la performance d'un cache. Dans tous les cas, avoir un taux de *Hit* élevé est strictement équivalent à avoir un taux de *Miss* bas. Toutefois, ils ne donnent qu'une vision partielle de la performance d'un cache en occultant le temps nécessaire pour différencier un *Hit* d'un

[3] Littéralement, bloc-note de brouillon

Miss par exemple, contrairement à la mesure du temps d'accès moyen. En se basant sur le fonctionnement général d'un cache présenté en section 2.3.1, page 19, le temps d'accès moyen à une donnée à travers un cache se calcule par la formule suivante :

Équation 1. *Caractérisation du temps d'accès moyen à une instruction dans un cache*

$$\begin{cases} T_{Accès moyen} = T_{Recherche} + (1 - P_{Miss}) * T_{GestionHit} + P_{Miss} * T_{Miss} \\ T_{Miss} = T_{Récupération} + T_{Éviction} + T_{GestionMiss} \\ T_{Récuperation} = T_{NAND} + T_{FTL} \end{cases}$$

Le temps d'accès à une donnée, $T_{Accès moyen}$, est ainsi égal au temps nécessaire pour trouver une donnée, $T_{Recherche}$, auquel s'ajoute le temps nécessaire à la gestion d'un *Hit* ou d'un *un Miss*. Pour le temps d'accès moyen, ce temps de gestion est fonction de la probabilité d'occurrence d'un *Miss*, P_{Miss}, probabilité égale à 0 en cas de *Hit*.

Un défaut de cache se décompose quand à lui en trois temps. Le premier est le temps nécessaire pour récupérer une ou plusieurs données depuis la mémoire secondaire, temps supposé incompressible, $T_{Récupération}$. Le temps restant est consacré à choisir un emplacement dans le cache pour les données nouvellement récupérées, $T_{Éviction}$, et à maintenir un ordre susceptible d'aider à éviter d'autre défaut de cache, $T_{GestionMiss}$ (*cf* Politiques de remplacement, section 2.3.3 page 23).

De manière générale, cette équation donne les trois leviers principaux de performance d'un cache :

1. maximiser la probabilité de trouver une donnée dans le cache, ou de manière équivalente, minimiser la probabilité P_{Miss} d'un défaut de cache ;
2. minimiser le temps de recherche d'une donnée effectivement en cache ;
3. minimiser le délai d'un défaut de cache T_{Miss} en minimisant les temps $T_{Éviction}$ et $T_{GestionMiss}$ grâce à des stratégies de placement et de renouvellement efficaces.

3.3.1.3 Limitations d'un cache

Comme nous l'avons vu, de simples tampons pour régler le problème de l'accessibilité n'étant pas suffisants au global, une solution à base de cache est connue comme une meilleure approche. De plus, l'intégration d'un cache est un problème d'ingénierie qui peut être résolu assez facilement.

Néanmoins, une telle solution est toujours limitée, car un cache continue à constituer un goulot d'étranglement. Mettre en place un cache est (re-)connu à tort comme une sorte d'accélération. À tort, car il n'accélère pas les accès à la mémoire secondaire mais masque une partie de son temps de latence global. Du point de vue du processeur, un cache continue donc à être un ralentissement dans l'exécution d'un programme si on le compare à l'exécution en place depuis la mémoire adressable. Tout simplement parce que l'accès à une instruction en cache ne peut pas se faire *a priori* en un cycle CPU. Il faut au moins compter le coût supplémentaire de la recherche en cache : comme il n'est pas possible d'exécuter une instruction qui n'est pas en cache, il faut au moins s'assurer de sa présence ou de son absence, ce qui cause un temps de latence minimal. Même si cette étape est optimisée au maximum, finement au cycle près, pour être par exemple de 30 cycles, il en résulte encore un écart de performance de 1 pour 30 entre la Flash NOR et un cache.

Dans l'absolu, un cache est donc un bon point de départ pour résoudre notre problématique et représente un socle de performance non-négligeable. Mais il est évident qu'il ne peut pas combler le déficit de performance à lui-seul.

3.3.2 Deuxième approche : le recouvrement d'opérations

3.3.2.1 Présentation

Les politiques de remplacement sont considérés comme le principal levier algorithmique d'optimisation des caches, car ils travaillent directement sur ce qui coûte le plus cher dans l'usage d'un cache, à savoir l'accès à la mémoire secondaire. Certes, cet objectif est déjà l'essence même d'un cache, mais l'algorithme MIN montre que le point optimal n'a pas encore été atteint dans la gestion des défauts de cache. Cependant, il s'est très vite avéré que chaque nouvelle amélioration de ces politiques et que chaque apport à l'état de l'art n'amenait que des gains de plus en plus faibles par rapport aux précédents.

De nouvelles approches ont donc été proposées, dont la plus intéressante est le recouvrement d'opération. Elle se base sur l'hypothèse que la mémoire secondaire peut-être accédée de manière asynchrone. Le principe est de profiter du temps de préparation par la mémoire secondaire des données à récupérer lors d'un défaut de cache pour effectuer d'autres opérations en parallèle.

Deux directions sont utilisées dans la littérature pour occuper ce temps de recouvrement. La première est « réactive » et n'agit que lorsqu'un défaut de cache intervient. La seconde est « pro-active » et tente de prévoir à l'avance un défaut de cache pour anticiper son traitement. Cette approche est aussi connue sur les termes de pré-chargement[4]. Ces deux approches proposent donc des réponses différentes sur le type de tâches qui peuvent être exécutées en parallèle.

3.3.2.2 Recouvrement par ré-ordonnancement

L'approche réactive est très souvent mise-en œuvre sur des architectures multi-cœurs, sur des systèmes supportant les processus légers, ou des architectures matérielles dotées de DMA[5]. En cas de défaut de cache, l'ordonnanceur système suspend la tâche courante, engage le pré-chargement et passe automatiquement la main à une autre tâche jusque là en attente. Le recouvrement d'exécution des deux tâches permet ainsi un gain de temps global.

Lorsque le recouvrement peut être supporté par des environnements déjà orientés parallélisme, alors cette approche est un bénéfice indéniable sur les performances d'un cache. Malheureusement, cette configuration n'est pas celle des cartes à puce.

3.3.2.3 Pré-chargement sans matériel

Un algorithme de pré-chargement[Smith 1982, Callahan 1991, Griffioen 1994] commence d'abord par prédire, selon certains critères, qu'une information (*e.g.* donnée, instruction) nécessaire dans un futur proche n'est pas présente en cache. L'idée est ensuite de soumettre à l'avance la demande à la mémoire secondaire, laisser le système reprendre là où il en est resté, puis si besoin le bloquer au moment où il a effectivement besoin de l'information, si la mémoire secondaire n'a pas encore terminé sa tâche.

[4] En anglais, *prefetching*

[5] Direct Memory Access, module matériel permettant des échanges de données entre mémoires sans intervention du CPU.

Pour mettre en application cet algorithme, deux stratégies sont possibles : l'injection d'informations à la compilation, ou la découverte dynamique de ces mêmes informations pendant l'exécution du programme.

Injection à la compilation Lors de la compilation du binaire, des instructions spéciales ou un appel à une routine de pré-chargement sont insérées à des endroits permettant d'anticiper de futurs défauts de cache. Notons que pour que ce soit une instruction, il faut bien évidemment qu'elle soit supportée par le processeur s'il s'agit de code natif, ou la VM pour du code interprété, ce qui n'assure plus la portabilité.

Déterminer où un programme peut déclencher un défaut de cache est assez facile à isoler puisqu'il s'agit, pour une très grande majorité, de branchements, conditionnels ou inconditionnels. Ce qui rend ces lieux de défauts de cache potentiels facilement identifiables à l'aide du graphe de flot de contrôle.

Déterminer, quelle page de mémoire secondaire devra être pré-chargée devient légèrement plus complexe. Pour que cette information soit non-équivoque, il faut calculer d'une manière ou d'une autre, des transpositions d'adresses virtuelles fournies au cache en adresses physiques à l'intérieur de celui-ci. [Park 2004] *et al.* mettent en œuvre cette approche dans leurs travaux pour améliorer les performances de leur solution de cache « compilé ». Ils utilisent une table de mappage d'adresses *virtuelles/physiques* construite à la compilation à l'aide d'une approximation de l'algorithme de *clustering*, connu pour être NP-dur. Ils ne considèrent toutefois que les branchements de type appels de fonction C, pour simplifier fortement le problème et pour garder une table de taille raisonnable, et non la liste complète des nœuds du graphe de flots de contrôle.

Le dernier élément à prendre en compte, est la détermination du meilleur endroit dans le code où placer le pré-chargement. L'algorithme de *clustering* n'est déjà qu'une approximation ce qui montre que cet élément est non-trivial. Nous appellerons cet endroit dans le code, le point d'insertion.

Ce qui guide le choix du point d'insertion est dans un premier temps, la durée de préparation de données de la part de la mémoire secondaire. Si cette durée est de N cycles CPU, le point d'insertion optimal se situe donc N cycles avant l'exécution de la dernière instruction d'un bloc de base B. Cette optimalité est pratiquement inatteignable car ce point d'insertion optimal se situera majoritairement en dehors du B et donc dans un ou plusieurs nœuds indéterminés du graphe. Pour rappel, lire une page de Flash NAND dure un peu plus de 2900 cycles, ce qui pousse la profondeur de recherche du point d'insertion dans la graphe à des distances et un nombre exponentiel d'arcs à suivre.

Néanmoins, une version assez simple, consistant à positionner le point d'insertion au niveau de l'instruction de tête d'un bloc de base, pourrait convenir au code interprété. En effet, dans ce type d'exécution, l'interprétation d'une instruction prend beaucoup plus de cycles CPU qu'une instruction native, puisqu'elle consiste en l'exécution d'une routine plus ou moins complexe. Dans nos résultats qui seront présentés par la suite, lire une page de Flash NAND prend le même temps, en moyenne, que l'exécution de blocs de base de 4 à 6 *bytecodes*. Les points d'insertion en tête de bloc sont alors suffisamment fiables et évite tout pré-calcul.

L'inconvénient majeur de cette approche est son caractère systématique, qui fait que la détermination d'un pré-chargement peut rapidement devenir une surcharge de travail, comparé au nombre réel de pré-chargements déclenchés. En effet, le pré-chargement est avant tout l'évaluation d'une condition, qui ne déclenche donc pas nécessairement de renouvellement du cache et donc de recouvrement d'opérations.

Prédiction dynamique Chilimbi *et al.* présentent dans [Chilimbi 2002] une autre approche destinée à découvrir pendant l'exécution, les mêmes informations que celles produites précédemment par analyse statique lors de la compilation. La technique dynamique utilisée est un profilage des accès mémoires qui fournit des statistiques amenant à identifier des « points chauds » , *i.e.* des données très fréquemment utilisées. Après un certain temps d'exécution, lorsque des points chauds sont identifiés, leur système injecte dynamiquement du code machine de pré-chargement à des points d'insertion qu'il juge opportun (non-présentés). Dans son mode de fonctionnement, cette approche est donc une hybridation des approches réactive/pro-active. Dans le sens où elle réagit à un historique pour anticiper l'avenir. Ce qui est intéressant par rapport aux inconvénients de la compilation, c'est que cette approche insère des tests de pré-chargements uniquement des endroits anticipant des accès à une donnée vitale. Donc en enlevant le caractère systématique obligatoire dans l'approche statique.

Cependant, le profilage et la détection des points ajoutent un coût d'exécution supplémentaire (*overhead*) comme toute approche dynamique. Leur contribution est évaluée selon cet aspect et les conclusions sont riches d'enseignements. Il démontre que l'*overhead* de la partie profilage/analyse est plus que compensé par la qualité des points d'insertion et des pré-chargements déclenchés. Malheureusement, l'absence d'une comparaison avec d'autres approches de pré-chargement, notamment statique, rend leur contribution impossible à évaluer dans l'absolu. De plus, celle-ci se base sur des hypothèses de taux de *Miss* relativement élevés et est donc plutôt une stratégie palliant des caches déjà peu performants. Enfin, l'infrastructure complète utilisée dans ce papier ne peut être utilisée dans une carte à puce et n'est pas reproductible dans ce contexte.

Une autre stratégie un peu plus éloignée encore du contexte carte à puce, connue sous le nom de *Runahead Processing* (exécution à l'avance) [Mutlu 2003, Mutlu 2005], a été implémentée avec succès dans des processeurs Intel. Bien que matériel, l'esprit de cette approche est une piste intéressante.

Cette approche est également une hybridation dynamique des approches réactive/pro-active mais matérielle cette fois. L'idée principale est toujours d'attendre que se produise un défaut de cache, de soumettre la demande à la mémoire secondaire et d'essayer d'utiliser le CPU à d'autres choses pendant le temps d'attente qui en résulte. À la différence des solutions matérielles déjà présentées et uniquement réactives, Mutlu *et al.* utilise le temps de recouvrement pour analyser le flots d'exécution disponible dans le cache d'instructions. Cette analyse commence à l'instruction fautive et suit le graphe de flot de contrôle qui en part jusqu'à constater une instruction manquante en cache, dans toutes les directions possibles du graphe de flot de contrôle. Un algorithme de décision est ensuite utilisé pour choisir si un pré-chargement peut être effectué. Dans cette approche, si le défaut de cache qui n'a pu être évité est géré « réactivement » , le prochain s'il existe peut alors être soumis pro-activement, dès que la mémoire secondaire a fini son traitement en cours.

Synthèse sur le pré-chargement Le recouvrement d'opération par pré-chargement n'est réellement utile que lorsque le nombre de défaut de cache est important et les approches présentées se basent sur cette hypothèse de départ. Mais en dehors de ce cas de figure, qui n'est pas le meilleur comportement attendu d'un cache, le recouvrement d'opération par pré-chargement peut aller jusqu'à devenir un handicap. En effet, les solutions par injections d'instructions ou par analyses dynamiques augmentent significativement le coût en cycles CPU. Dans ces conditions, si l'*overhead* est supérieur au gain apporté par les recouvrements `effectifs`, alors le recouvrement d'opération par pré-chargement dégrade les performances.

Le recouvrement d'opération est donc une approche délicate à manipuler et obtenir d'elle

une valeur ajoutée est concrètement difficile sans support matériel ou sans circonstances adaptées.

3.3.3 Notre approche : regroupement d'accès au cache

Notre approche part du constat, que nous montrerons, que les principaux leviers de performance d'un cache dans notre contexte ne sont ni la politique de remplacement, ni la gestion anticipée des défauts de cache. De plus, nous montrerons également que le goulot d'étranglement se situe dans le nombre d'accès au cache et non sur une gestion optimale de l'alimentation du cache.

Néanmoins, un cache, dont la forme reste à définir pour notre contexte, est indéniablement un composant qui permet de résoudre une bonne partie de notre problématique. Les usages qui en sont fait dans des cas similaires et depuis des décennies le prouvent.

Malheureusement, une carte à puce n'a ni cache d'instructions, ni suffisamment de mémoire vive pour adopter pleinement les approches de la section 3.3.1, page 29. La conception d'une interface d'exécution en place dans une carte à puce doit donc être encore plus radicale, et doit prioritiser dans l'ordre : empreinte mémoire, efficacité, et accessibilité. Bien que cette dernière soit déjà facilement réalisable, comme cela a déjà été discuté.

Fort de nos constats sur l'état de l'art et sur les lacunes des caches, il nous est apparu que l'idée de grouper des requêtes pour effectuer un seul accès au cache permettrait de mutualiser des cycles CPU. Cette approche par regroupement d'accès pose de nouveaux problèmes que nous allons décrire ici. Nous donnons ensuite des pistes susceptibles de les résoudre.

3.3.3.1 Contraintes structurelles d'un cache

L'intuition est qu'un premier accès au cache pour une instruction de tête permet l'accès direct aux autres instructions du bloc de base sans autres interventions du gestionnaire de cache. Ce qui revient à accéder au contenu du cache en outre-passant son interface logicielle. Hélas, il n'est pas possible de rester cohérent dans la durée avec une telle approche.

Prenons comme point de départ un opérateur d'exécution - un processeur, ou une JVM - , ayant besoin d'accéder à une valeur K à l'adresse virtuelle $@_{virt}$. Imaginons ensuite que le cache, au lieu de renvoyer K, renvoie l'adresse physique $@_{phy}\{P_0+8\}$, un offset dans la page n°0 du cache, en RAM, où se trouve une copie de la valeur K. Si maintenant le pointeur d'instruction (IP) courant de l'opérateur d'exécution est redirigé vers $@_{phy}\{P_0+8\}$ plutôt que $@_{virt}$, on peut penser « naïvement » que le programme puisse continuer son exécution sans devoir à nouveau requêter le cache, simplement en incrémentant IP comme il se doit.

Or, ce principe de fonctionnement est invalide car, de fait, rien n'indique que IP continue à lire et exécuter du code valide au-delà de $@_{phy}\{P_0+8\}$. Tout simplement parce que cette adresse est une adresse dans une page de cache, et que $@_{virt}+1$ peut en réalité se trouver dans une autre page de cache, à l'adresse $@_{phy}\{P_5+0\}$ par exemple, et non à $@_{phy}\{P_0+9\}$. Ce phénomène est dû à la politique de remplacement et aux branchements dans le programme qui font que deux pages de cache adjacentes physiquement se retrouvent inévitablement[6] à contenir des copies de blocs de données non-contigües du binaire du programme.

Dans ce cas, l'opérateur d'exécution ne sait pas qu'il n'évolue plus sur une plage d'adresses physiques continues et son comportement devient alors indéterminé. À ce stade, la seule parade est donc d'effectuer un accès au cache par son interface logicielle pour chaque instruction ou opérande à récupérer.

[6] Le contraire serait le fruit du hasard.

3.3.3.2 Vers une découverte dynamique de groupes d'accès

La seule lacune de l'approche décrite ci-dessus est l'absence d'une information de terminaison claire et précise qui indiquerait à l'opérateur d'exécution que l'incrément de IP sort d'une plage d'adresses physiques continues. Cette information de terminaison est soumise à de deux cas de figures.

- Le premier cas de figure intervient lorsque IP pointe vers un rupteur de flot de contrôle sortant de la page de cache courante.
- Le second cas de figure est la détection d'une fin de page de cache. C'est-à-dire détecter que IP finit par pointer vers l'adresse physique du dernier octet de la page de cache courante, qui peut ne pas être un rupteur de flot de contrôle.

Ces deux informations de terminaison ne sont détectables que par l'analyse du graphe de flot de contrôle. En cherchant à identifier des plages d'adresses physiques continues à l'intérieur du cache, nous cherchons implicitement à former des groupes d'instructions qui pourrait être exécutés de manière séquentielle. La découverte de plages d'adresses continues peut être résolue en cherchant à reconstituer artificiellement un bloc de base par analyse du graphe de flot de contrôle, information perdue une fois que le binaire est compilé.

- Recherche de groupes hors-ligne - Réaliser cette analyse hors-ligne est envisageable simplement, car elle s'apparente aux travaux sur le pré-chargement et l'injection d'informations à la compilation. À ce stade de la vie d'un programme, bon nombre d'informations requises sont disponibles. Le premier inconvénient de cette approche est l'augmentation de l'empreinte mémoire dûe aux nouvelles données accompagnant le binaire exécutable. Le second, plus important, est que le cas de figure n°2 ne peut être entièrement satisfait sans une connaissance globale de la configuration et du type de cache, ainsi que les propriétés de la mémoire secondaire. Ce qui implique que dans la *post-issuance*, une application téléchargeable doit être compilée autant de fois que de couples cache/matériel possibles. Ce qui est une contrainte impossible à mettre en application, et qui n'est pas du tout l'approche et l'esprit de Java et JavaCard par exemple.

- Recherche de groupes en-ligne - Réaliser l'analyse en-ligne du graphe de flot de contrôle est donc une meilleure approche, car les informations sur le cache et les propriétés de la mémoire secondaire sont ici connues. Tout comme dans les travaux déjà présentés sur la sécurité, elle peut être de deux natures : soit réalisée statiquement à l'installation de l'application dans la carte à puce, soit dynamiquement pendant son exécution.

L'analyse statique à l'installation rejoint les propositions de l'état de l'art sur la problématique de sécurité posée par la *post-issuance* (section 2.1.5.1, page 9). Si le domaine d'application est différent, elles montrent que l'analyse de code en-ligne est une solution encartable. Ce qui permet de conclure qu'il n'y a donc *a priori* pas de frein à d'autres usages de l'analyse du graphe de flots de contrôle en-ligne, si ce n'est bien-sûr l'effort à fournir pour la rendre viable dans une carte.

Appliquer l'analyse statique dès l'installation à notre approche offre les même avantages mais aussi les même inconvénients que dans le cadre de la sécurité. Toutefois, à la différence de la vérification du code par exemple, nous avons par contre besoin de garder l'information produite par l'analyse, ce qui va à l'encontre d'une empreinte mémoire attendue faible. Sans connaître pour l'instant la taille unitaire d'une de ces informations à mémoriser, on peut déjà noter que leur somme sera proportionnelle au nombre de blocs de base, voire plus, pour pouvoir mémoriser également les blocs tronqués par des fin de pages. À titre d'information,

le nombre moyen de blocs de base dans une application JavaCard est de 54 (sans les fins de pages), qui est lui-même à multiplier par le nombre d'applications encartées. Ce qui peut facilement doubler l'empreinte mémoire d'un cache de 512 octets.

C'est pourquoi nous proposons plutôt une approche de recherche d'informations sans mémorisation par une analyse statique en-ligne du graphe de flot de contrôle mais effectuée dynamiquement pendant l'exécution des programmes.

3.3.3.3 Méthode de découverte dynamique des groupes

Notre approche pour combler les lacunes d'un cache repose donc sur une analyse du code, en-ligne. Le but est d'identifier des plages d'adresses physiques continues, exécutables séquentiellement, et sans accès intermédiaire à l'interface du cache. Comme discuté, cette identification correspond à la découverte de groupes d'instructions qui sont soit des blocs de base complets, soit des blocs de base tronqués lorsqu'ils sont répartis dans plusieurs pages de cache.

Le principe est donc de ne réaliser qu'un seul accès au cache, sur l'instruction de tête, et trouver par analyse de quoi garantir que le bloc de base courant peut être exécuté sans autres accès à l'interface du cache. De plus, dans cette approche, une fois la garantie trouvée, le bénéfice s'étend au fait que seules les instructions de tête sont susceptibles de déclencher des défauts de cache.

Principe de fonctionnement Dans notre approche, une analyse de code est déclenchée à chaque exécution d'une instruction de tête. Elle est utilisée pour rechercher les fins de blocs de base et donc les deux types de terminaisons de plages d'adresses mentionnée précédemment. Notre approche offre une autre optimisation en permettant de greffer facilement à l'analyse un mécanisme de pré-chargement de pages. Elle permet en effet de découvrir de manière précoce si une information est présente en cache ou non et donc d'anticiper sa récupération.

Guides de conception Une telle approche pose les mêmes problèmes que tout algorithme dynamique d'analyse, à savoir le rapport entre son coût et son gain. Ici, le gain espéré est la réduction du coût total des accès au cache en ne l'utilisant qu'avec plus de parcimonie. Tandis que le coût principal de notre approche est celui de l'analyse de code. Le rapport gain/coût s'articule donc autour du nombre d'analyses effectuées et du nombre d'accès au cache évités, rapport décrit sous forme d'inéquation dans la formule ci-dessous.

Inéquation 1. *Seuil de rentabilité de notre approche*

$$\begin{cases} Coût_{Analyses} & < & Coût_{Accès-évités} \\ Coût_{Analyses} & = & T_{Analyse} * N_{Analyse} + \Delta \\ Coût_{Accès} & = & N_{Accès} * (T_{Recherche} + T_{GestionHit}) \end{cases}$$

De manière générale, concevoir une méthode de découverte dynamique d'informations dépend fondamentalement du type d'informations à collecter. Elle dépend bien entendu également du type de données à analyser, et enfin et surtout, du contexte dans lequel évolue d'une part l'analyse elle-même et d'autre part les données analysées.

Le type d'informations à analyser et leurs contextes d'utilisation seront étudiés dans les chapitre 4 pour du code compilé, et chapitre 5 pour des méta-données Java, qui sont les deux types de données en jeu dans l'exécution de code dans une carte à puce. Ces deux

chapitre seront également l'occasion d'évaluer l'utilisation d'un cache logiciel, qui reste le socle de notre approche.

L'élaboration, la conception et l'implémentation de notre approche, ainsi que la mesure de ses bénéfices sont ensuite détaillées dans le chapitre 6.

3.4 Méthodologie

Notre méthode pour évaluer la capacité des approches présentées dans la section précédente à résoudre notre problématique est un étude phénoménologique de leur comportement lorsqu'elles sont confrontées aux contraintes de notre contexte. Dans cette section, nous présentons les critères qui permettent de les évaluer et juger de leur pertinence, leur efficacité à résoudre notre problématique, et enfin leur performance. Nous présentons ensuite une liste de programmes de tests à même de correspondre d'une part à des comportements typiques et d'autre part à des critères spécifiques au monde des cartes à puce.

3.4.1 Critères d'évaluation

Le principal critère de performance est fixé par la norme ISO-7816-3 dans laquelle est défini le délai maximal d'une transaction entre un lecteur et une carte à puce. Ce seuil est fixé à cinq secondes et est en fait basé sur une étude qui a montré qu'un utilisateur lambda toléré un temps d'attente compris entre trois et cinq secondes. Ce critère est donc celui par défaut d'une exécution en place depuis la Flash NOR. Cela signifie que quelque soit l'approche suivie pour exécuter du code stocké dans une mémoire non-adressable, elle ne doit pas avoir un facteur de ralentissement poussant une transaction au-delà de ces cinq secondes.

À ce critère pratique, s'ajoutent plusieurs critères techniques qui permettent de comparer les solutions technologiques et logicielles entre elles. Le premier critère est le débit en lecture qui permet de comparer un cache, par exemple, à la Flash NOR et la Flash NAND. Le débit de ces deux dernières sont des frontières physiques. En d'autres termes, aucune solution ne pourra jamais dépasser le débit en lecture de la Flash NOR exécutant du code en place, ce qui représente donc ce vers quoi une solution doit tendre. Pour la Flash NAND par contre, cette `frontière physique` est un point de repère de performance minimale. Cette limite, fixée par le débit crête de la NAND en lectures séquentielles, marque le seuil où une solution passe du status de peu pertinente à efficace en proposant une solution dépassant le verrou des temps de latences de cette Flash.

Pour aller au-delà, la performance ne se situe plus en terme de débit mais en terme de facteur de ralentissement. C'est-à-dire, comment une solution proposée, efficace dans sa gestion de la mémoire non-adressable, continue à entretenir un certain temps de latence du point de vue de l'opérateur d'exécution. Ce critère s'évalue en terme de coût en instructions (CoI), *i.e.* le nombre d'instructions processeurs à exécuter pour récupérer une donnée stockée dans la mémoire non-adressable. Ce critère est le critère technique déterminant qui fait qu'une solution puisse au final respecter le temps d'exécution de la norme ISO-7816-3. Il représente ainsi le deuxième verrou de notre problématique qui constitue cette fois une `frontière logicielle` entre l'opérateur d'exécution et la mémoire non-adressable.

Pour terminer cette liste de critère, une solution logicielle est contrainte par la quantité de mémoire interne disponible. Tout d'abord, une telle solution doit avoir une empreinte réduite pour l'espace de code. Elle doit ensuite maîtriser son besoin en mémoire vive. C'est pourquoi nous viserons un objectif d'occupation en RAM inférieur à 2048 octets[7], en

[7]Soit, la taille d'une page de Flash et/ou la taille d'un tampon équivalent

cherchant à le réduire encore si possible. Ainsi, si un cache, par exemple, peut s'avérer une solution efficace, nous devons l'aborder également selon ce critère pour clairement statuer sur sa pertinence. L'évaluation des différentes solutions est donc également confrontées à `frontière conceptuelle`, où l'efficacité se mesure cette fois sur la capacité de l'étape de conception à gérer efficacement les maigres ressources à disposition, notamment la mémoire vive.

3.4.2 Programmes de tests

Pour mettre en lumière les forces et les faiblesses des différents éléments de notre étude, il nous faut utiliser des programmes d'évaluation cohérents avec notre contexte et avec ce que nous voulons étudier. La première mise en cohérence concerne la taille de ces programmes. Par exemple, un programme dont la totalité du code pourrait être copiée dans l'espace de cache n'a pas beaucoup d'intérêt et n'apporterait pas de contributions significatives. Le deuxième aspect concerne la façon dont sont compilés ces programmes. Il doivent correspondre à ce qu'ils seraient dans une carte à puce ou un système du même genre, à savoir être compilés avec des options privilégiant d'abord la petitesse à la vitesse. Les programmes de tests que nous utilisons sont tous recompilés avec cette option et sans bibliothèques dynamiques, ce qui peut produire de légères différences avec d'autres papiers dans lesquels ils sont également utilisés. Le dernier élément dans le choix de programmes de tests repose sur la diversité des technologies applicative présente sur les cartes à puce actuelles. Nous devons ainsi évaluer à la fois des programmes compilés en code natif, mais aussi des applications écrites dans des langages de plus haut niveau comme le Java et produisant du code semi-compilé.

Évaluation de code compilé natif Les analyses qui suivent s'appuient sur des programmes de la suite de tests MiBench [Guthaus 2001], une suite de tests conçue pour les systèmes embarqués. Si certains programmes de cette suite sont trop complexes pour une carte à puce, le sous-ensemble nommé « sécurité » est lui complètement en adéquation fonctionnelle avec elles. Il comprend plusieurs programmes d'encodage/décodage répandus comme Rijndael (AES), Blowfish, SHA, ou GSM, voir [Eisenbarth 2007] pour une veille sur leurs implémentations pour l'informatique enfouie.

Notre document propose également une collecte de résultats pour une machine virtuelle Java. Celle utilisée est KVM[8] [Simon 1999], l'implémentation de référence proposée par Oracle pour la catégorie J2ME[9], destinée aux systèmes embarqués types téléphone mobile[10]. Il s'agit de la plus petite catégorie de spécifications Java possédant encore toutes les fonctionnalités du langage - à la différence de JavaCard. L'architecture de code d'une machine virtuelle est très particulière du fait de son interpréteur et mérite donc qu'on s'attarde sur ce genre de spécificité.

Évaluation de code interprétré Il existe de nombreux programmes d'évaluation et de tests pour les machines virtuelles Java [SPEC 1998, Bull 2000, Smith 2001, Pozo 2005, Blackburn 2006, Schoeberl 2010]. Malheureusement, ces programmes sont bien souvent conçus pour tester les aspects fondamentaux d'une JVM comme le nombre de *bytecodes* exécutés à la seconde, les comportements du ramasse-miette, la parallélisation, ou des tests unitaires comparés à d'autres langages. Ces suites de tests sont soit trop grosses pour

[8]Kilobyte Virtual Machine
[9]Java2 Micro Edition
[10]Bien qu'à l'origine elle ait été conçue pour Palm Pilot

nos cibles, soit rarement riches de fonctionnalités « objet » qui mettent l'accent sur les méta-données.

Des programmes de test beaucoup plus judicieux pour notre étude se trouvent dans la suite de Richards [Simon 2006]. Cette suite à l'avantage de proposer toute une série de programmes ayant exactement les même fonctionnalités mais écrits et développés avec des approches différentes et/ou avec des langages différents dont le C, C++ ou JAVA. Le cœur de ce programme simule le gestionnaire de tâches d'un noyau de système d'exploitation.

Sa version Java est très intéressante car elle offre 7 versions différentes d'une même tâche applicative, en balayant un large spectre de techniques de programmation orientée *objet* [Bloch 2008], des plus basiques aux plus riches (passant de la même façon du C vers le C++) :

- la version 1 est une traduction directe du benchmark d'origine dans le style procédural du C sans fonctionnalité *objet*, *i.e.* pas d'encapsulation, pas d'héritage, pas de surcharge de méthodes ;
- la version 2 ajoute le mot clé *final* à la version 1 quand c'est possible pour réduire les coûts d'appel de méthodes Java ; ce mot clé permet de concrétiser définitivement certaines méthodes virtuelles ;
- la version 3 remplace le *switch/case* central du gestionnaire noyau simulé par des variables d'état sous forme d'*objets* ;
- la version 4 réécrit par héritage et abstraction la notion de tâche mais en gardant tous les champs *publics* ;
- la version 5 « privatise » ces champs publics qui ne peuvent alors être atteints que par des accesseurs virtuels[11], pour reprendre les règles de l'encapsulation *objet* ;
- la version 6 ajoute à version 5 le mot clé *final* quand c'est possible comme dans la version 2 ;
- enfin, la version 7 est une version proche de la conception d'un *framework* en utilisant des interfaces pour le typage des variables et le passage de paramètres.

Nous avons légèrement modifié ces programmes pour qu'ils puissent être compatibles avec les restrictions JavaCard 2.2. Mais ces modifications n'enlèvent rien à la complexité intrinsèque de chaque programme, et porte sur la suppression des quelques chaînes de caractères d'affichage de résultats et l'utilisation d'entiers 16 bits (*short*) pour l'indexation de tableaux, obligatoire en JavaCard.

Application représentative JavaCard 2.2 La suite de tests Richards est très pratique pour notre évaluation des méta-données. Elle pousse en effet à l'extrême un comportement à étudier, et tel est l'objectif d'un programme de test. Cependant, ces applications artificielles ne sont pas réellement représentatives de vraies applications Java, et moins encore d'applications JavaCard. Les versions 5 à 7, les plus intéressantes sur le plan des méta-données et du style, ne mettent pas en évidence les propriétés d'une application JavaCard lorsqu'il faut l'observer dans son ensemble et dans son contexte, et non uniquement sur les méta-données. Ces programmes de test présentent en effet trois problèmes à cette échelle.

[11] *getter* et *setter* dans le jargon de la programmation orientée objet

1. Tout d'abord d'un point de vue du temps d'exécution. Dans nos tests finaux que nous avons réalisé sur du matériel de carte à puce, une seule itération de la version 7 dure environ 34 mn, à 17.5MHz, et 24 minutes pour la version 1. Ce qui est purement et simplement inadapté.

2. Le deuxième problème, expliquant en partie le premier, est l'utilisation intensive de la *heap* de la JCVM. En effet, ces programmes utilisent énormement d'écritures dans des champs d'objets Java, ce qui est courant dans un programme de test, mais qui est loin de l'usage standard constaté dans une application JavaCard. D'autant que la *heap* de la JCVM se situe en Flash NOR, qui est très lente en écriture. Néanmoins, JavaCard propose la possibilité de stocker explicitement des *objets* temporairement en RAM, mais l'architecture des programmes de tests n'a pas permis d'utiliser cette option.

3. Les applications JavaCard et les cartes à puce possèdent une autre particularité de fonctionnement qui repose sur une succession de transactions et de mise-hors-tensions. De plus ces transactions sont soumises au traitement d'évenements extérieurs difficiles à reproduire *in-vitro* et ont une durée maximale d'exécution.

Nous avons donc étudié un large panel d'applications JavaCard pour identifier et mesurer les éléments qui forment leur spécificité pour rapport à des applications plus classiques. Notre objectif était de produire une application représentative, fidèle au mode de conception et structures des applications JavaCard.

Ce programme, que nous appelons JCProfil, est construit pour s'approcher le plus possible du profil moyen JavaCard, en respectant par exemple, le nombre de classes, le nombre de méthodes par classe, le nombre de *bytecodes* par méthode, la taille des blocs de base, le nombre d'appels de méthodes, de branchements conditionnels, etc, et avoir ainsi un nombre de *bytecodes* et un nombre de méta-données par *bytecode* cohérent avec la plus grande partie des applications JavaCard qui nous avions à disposition pour analyse.

En terme d'applicatif, une itération de l'application JCProfil simule 32 transactions manipulant plus ou moins arbitrairement de données stockées dans des listes chaînées et des tables de hachage, sur le mode d'une mini base de données. Pour nos tests, cette application est configurée pour 3 itérations pour rendre l'éxécution un peu plus longue et permettre des mesures plus précises. À titre d'information, dans une carte à puce à 17.5Mhz, et une exécution en place depuis la Flash NOR, ces trois itérations durent 3 minutes 45 secondes, soit 2,34 secondes par transaction.

3.4.3 Protocoles expérimentaux

Mesure du débit Pour produire ce type de résultats, nous avons créé un environnement de simulation capable d'analyser les comportements d'un cache logiciel quelques soient ses paramètres de configuration. Dans ce cadre de départ, la simulation se déroule en relisant une trace, par exemple adresse d'instruction par adresse d'instruction s'il s'agit de code compilé. À chaque instruction relue, le simulateur évalue si cette dernière est présente en cache et au quel cas comptabilise un *Hit* ou un *Miss*.

Dans notre simulateur, les algorithmes implémentant les stratégies de placement et de renouvellement sont écrits comme décrits dans l'état de l'art et disponibles dans de nombreux outils issus du monde des sources-libres. Le simulateur n'est en réalité qu'une enveloppe autour d'un gestionnaire de cache conçu et écrit en C, et intégrable tel quel dans une carte à puce. Le simulateur fait ainsi office d'opérateur d'exécution fournissant l'adresse virtuelle d'une donnée à un vrai gestionnaire de cache, et attendant une réponse en retour.

Mesure du coût en instructions Pour les mesures du critères du coût d'exécution d'un cache logiciel, nous reprenons exactement les mêmes implémentations de caches logiciels. Cette fois, le simulateur est lui même tracé pour établir le coût en instructions (CoI) de chaque algorithme des gestion du cache logiciel. La nouvelle trace générée ne contient que l'historique de l'exécution du code des algorithmes de cache, et ne contient donc pas de trace des instruments du simulateur notamment le relecteur de trace.

L'extraction du CoI pest réalisée en deux étapes et sur deux versions d'un même programme de test : une version Intel x86 32 bits, et une version ARM Risc 32 bits, notre architecture cible. La trace est obtenue sur la première version à l'aide de l'outil Valgrind sous Linux et d'un module complémentaire que nous avons spécialement écrit. Ce module est conçu pour tracer l'exécution par lignes de code source plutôt que par instructions natives. Chaque ligne x86 tracée est ensuite associée à son code ARM équivalent. Enfin, les statistiques sont agrégées par blocs de base ARM et/ou fonctions d'un algorithme de cache. Ainsi, les résultats présentés dans ce document sont tous basés sur du code ARM.

Cette approche *in vitro* rend l'évaluation indépendante de tous phénomènes extérieurs qui pourrait se produire sur un système en condition réelle. Certes, ce CoI n'est qu'une estimation du temps d'exécution car une instruction peut prendre plus ou moins de cycles CPU qu'une autre. Cependant, le nombre d'instructions collectées par simulation se comptant en dizaines de millions, cette approche rend donc cette estimation relativement fiable, et ce procédé est ainsi couramment utilisé en pareil circonstance.

Dans la suite de ce document, nous distinguerons les instructions collectées par la trace Valgrind retraduite du simulateur de cache sous l'acronyme IOv (Instruction de l'Opérateur virtuel), par opposition aux instructions ou données recherchées dans le cache simulé (ICa). Nous exprimons donc le coût en instructions d'un cache logiciel en nombre d'IOv et le coût d'un accès au cache en IOv/ICa.

Chapitre 4

Caches logiciels pour cartes à puce

Un cache est la solution le plus communément utilisée pour lire des données stockées dans une mémoire secondaire dont la latence est significativement plus longue que celle de la mémoire principale. Cependant un cache a également son propre temps de latence, temps accentué lorsqu'il est conçu entièrement en logiciel. De facto, la taille de l'espace mémoire dédié à un cache est admis comme son principal paramètre de performance, comme le montre la surenchère actuelle sans la taille des caches de processeurs, mais aussi des bases de données, des serveurs d'applications, etc.

Notre démarche pour construire un cache est inverse car elle consiste à réduire au minimum son empreinte mémoire pour qu'il puisse entrer dans une carte à puce. Dans ce but, nous présentons dans ce chapitre une analyse exhaustive des caches logiciels en passant en revue les stratégies d'implémentation qui peuvent être suivies, et en les confrontant à des caches de plus en plus petits. Nous montrons alors vers quelles performances maximales tend un cache, sans jamais les atteindre, lorsqu'il est conçu entièrement en logiciel et lorsqu'il doit faire face aux contraintes mémoires d'une carte à puce. Enfin, nous déclinons notre démonstration sur une classe particulière de cache que sont les caches d'instructions. Notre objectif est ainsi d'évaluer une solution de cache d'instructions logiciel à faible empreinte mémoire pour tenter de résoudre notre problématique de temps de latence lors de l'exécution d'applications stockée dans la mémoire non-adressable d'une carte à puce.

4.1 Cache logiciel à faible empreinte mémoire

Une carte à puce n'est pas dotée de support matériel pouvant l'aider à accéder rapidement à des données stockées dans une mémoire non-adressable à forte latence. Notre état de l'art montre qu'en pareille circonstance, un cache, même logiciel, reste un outil efficace et est presque exclusivement la seule solution utilisée. Malheureusement, il est aussi reconnu que la taille de l'espace de stockage affecté au cache est un facteur de performance essentiel. De part cet aspect, les contraintes mémoires que fixe une carte à puce sont donc sujettes à dégrader l'efficacité supposée d'un cache. Il est par conséquent nécessaire de ré-évaluer un cache logiciel, pour l'aborder à la lumière d'une empreinte mémoire la plus faible possible et pour juger de son intérêt dans une carte à puce.

Un cache est donc un système d'échange de données entre deux espaces mémoires dont une est sujette à une forte latence. Il a ainsi pour objectif de mettre plus rapidement ces données à disposition d'un consommateur, qui peut être une application quelconque. Un cache contient donc une copie partielle, temporaire et généralement désordonnée d'un ensemble de données se trouvant dans la mémoire secondaire sous forme unifiées. De son côté, le consommateur de données accède au cache pour lire des blocs de données de 1 à N octets.

Bloc de données : *nous définissons de manière générique un bloc de données comme étant une suite d'octets consécutifs, d'une taille qui ne varie pas tant qu'il se trouve dans le cache. Un bloc peut alors prendre concrètement la forme d'une variable, d'un tableau, d'une structure de données ou encore d'un bloc de base.*

Pour l'exploitation d'un cache logiciel dans une carte à puce, il s'avère donc que plus nous réduisons l'espace de stockage temporaire, plus nous limitons la quantité de blocs de données que celui-ci sera capable de resservir rapidement sans déclencher de défauts de cache. Toute l'efficacité d'un cache réside donc dans sa capacité à gérer au mieux ces blocs de données dans le temps.

Nous avons vu en section 2.3, page 19, que plusieurs éléments participaient à l'efficacité d'un cache et que plusieurs stratégies pouvaient être mises-en-œuvre pour chacun de ces éléments.

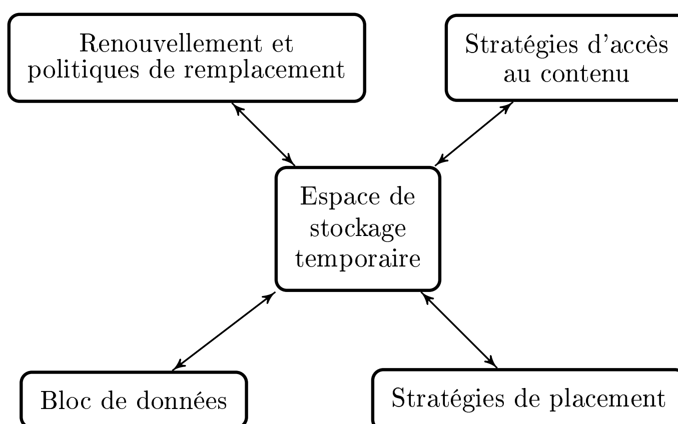


Figure 4.1: Interaction d'un cache avec son espace de stockage

Chacun de ces éléments interagit avec l'espace de stockage, qui est donc l'élément central d'un cache (figure 4.1, page 44).

1. Le bloc de données par des propriétés générales, comme sa taille ou sa fréquence d'utilisation, conditionne la quantité unitaire et la qualité du contenu dans l'espace de cache.

2. Les stratégies de placement dictent comment les copies de blocs de données issues de la mémoire secondaire sont organisées de l'espace de cache.

3. Les stratégies de renouvellement ont pour tâche de choisir quels blocs de données doivent rester dans l'espace de cache, et ceux qui peuvent en être supprimés.

4. Les stratégies d'accès au contenu sont quant à elle destinées à rechercher un tout ou partie d'un bloc de données à l'intérieur de l'espace de cache.

Par cette description, on constate clairement que la taille de l'espace de stockage a une influence sur chacune de ces stratégies et que réduire cet espace peut pousser ces stratégies à leur limite d'efficacité respective.

Dans la suite de cette section, nous passons en revue ces quatre éléments, dans l'ordre, en les confrontant à un espace de cache à faible empreinte mémoire. Par cette étude, nous mesurons l'impact d'une telle empreinte sur chacune de leurs stratégies pour identifier lesquelles supportent cette contrainte et jusqu'à quel point. Nous terminerons cette section

en observant comment une empreinte mémoire réduite conditionne le coût d'exécution d'un cache logiciel, coût qui constitue sa latence propre.

4.1.1 Propriétés généralisables des blocs de données

4.1.1.1 Taille

La première propriété généralisable d'un bloc de données est sa taille. Qu'il soit un entier 32 bits sur quatre octets ou une structure de données de 28 octets, un bloc de données n'a ainsi pas la même incidence sur l'espace de stockage. Le cache peut en effet stocké plus de types concrets du premier que du deuxième.

De plus, cette taille a un impact direct sur les stratégies de placement et une influence non-négligeable sur le renouvellement. Comme nous l'avons vu section 2.3.2, page 20, des blocs de tailles fixes ou de tailles variables génèrent plus ou moins de fragmentation et de pollution, qui sont deux phénomènes dégradant l'efficacité d'un cache. Au final, c'est l'espace de stockage utile qui s'amenuise, augmentant ainsi la probabilité de déclencher des défauts de cache plus fréquemment. La taille d'un bloc est donc un élément fondateur.

4.1.1.2 Degré de séquentialité

Un cache subit directement les propriétés physiques de la mémoire secondaire. Nous avons vu section 3.2.1, page 27, qu'une Flash NAND est peu performante pour les accès aléatoires. Ce manque d'efficacité se trouve propagé au cache si les demandes trop aléatoires du consommateur de données ne sont pas amorties par la capacité du cache à éviter les *Miss*.

La taille du bloc de données permet de cerner ce problème. En effet, on peut dire que si la taille d'un bloc de données est égale à N, cela signifie que la lecture de ce bloc est constituée d'une première lecture aléatoire suivie de $N-1$ lectures séquentielles.

Nous définissons alors le rapport $(N-1)/N$ comme étant le `degré de séquentialité` d'un bloc de données, compris entre 0 et 1, exclus.

Prenons par exemple, une taille moyenne de 10 octets pour des blocs de données demandés au cache. Ce qui représente seulement en moyenne une lecture aléatoire toutes les 10 lectures et un degré de séquentialité de $9/10 = 0,9$, soit en fait 90 % de lectures séquentielles.

Un `degré de séquentialité` élevé est une information fondamentale dans la résolution de notre problématique. Du point de vue d'un cache, il indique que dès lors qu'un bloc de données complet est présent en cache, alors aucune demande d'accès au cache pour une donnée qui le compose ne déclenchera de défaut de cache. Ceci bien évidemment, indépendamment de la façon dont le bloc de données a été rapatrié en entier dans le cache, problème que nous aborderons tout au long des sections suivantes de ce chapitre.

4.1.1.3 Points chauds

De manière générale, quelque soit leur type, des blocs de données n'ont pas tous le même niveau de sollicitation. Ces différences de niveaux amènent à distinguer plusieurs catégories connues sous des appellations de points de chaleur [Knuth 1971]. Ainsi, une donnée surutilisée est qualifiée de point chaud, à la différence d'une donnée peu fréquemment utilisée qui est dite froide.

Deux exemples de cartographie des points chauds sont illustrés dans la figure 4.2 (page 46). Chaque pixel représente un octet, ici dans des binaires exécutables, et une ligne correspond à 512 octets. Un point bleu matérialise un octet lu et exécuté moins de 100 fois,

un point vert moins de 1000 fois, un point orange moins de 10000, un point rouge moins de 100000 et un point noir plus de 100000 fois. Tous les points blancs correspondent à des instructions présentes dans le binaire mais qui ne sont jamais exécutées.

L'axe des ordonnées représente les adresses de chaque ligne de 512 octets, alignées sur le premier octet du binaire. Pour une analogie avec un stockage du binaire dans une mémoire paginée comme la Flash NAND, la grille mineure horizontale, en gris clair, découpe l'espace de code en blocs de 2048 octets, soit une page de Flash NAND.

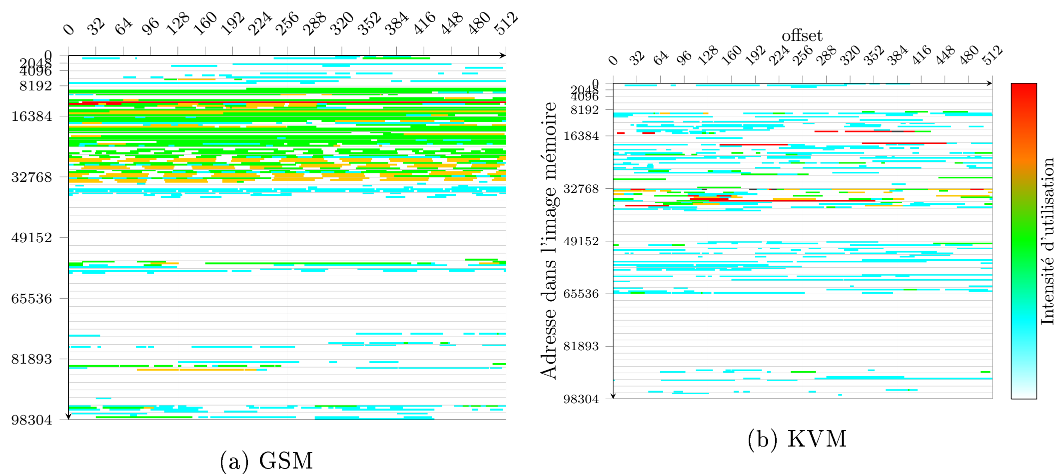


Figure 4.2: Deux exemples de cartographie des points de chaleur, pour deux binaires exécutables

Pour la conception d'un cache, ces points sont toutefois plus des informations observables que réellement exploitables. Sans outils de profilage, il est en effet très difficile de constater qu'*a priori* qu'un bloc de données est un point chaud ou non, et *a fortiori* sur un intervalle de temps donné. Néanmoins, ils constituent un angle d'analyse intéressant car ils forcent dynamiquement certains comportements du cache et notamment des comportements qu'il serait souhaitable d'éviter. Ils permettent donc de comprendre pourquoi un cache est bon dans certains cas mais moins efficace dans d'autres.

Vue leur intense utilisation, les points chauds sont donc des blocs de données ayant vocation à rester durablement dans le cache. Si ceux-ci, pour une raison ou une autre, ne sont pas présents en cache, ou si ce dernier peine à les y maintenir, alors le taux de *Miss* augmente rapidement. À l'inverse, les points plus froids, bien qu'étant peu utiles[1], devront à un moment ou un autre être exécutés, récupérés de la mémoire non-adressable et évinceront d'autres blocs du cache. De plus, et même si c'est pour un laps de temps très court, ils occuperont de l'espace dans le cache au détriment d'autres blocs, tant que la politique de remplacement n'aura pas jugé bon de les supprimer.

Par rapport à l'espace de stockage, la qualité d'une donnée en cache est donc associée à sa chaleur. Le meilleur espace de stockage est donc celui à même de pouvoir contenir le plus grand nombre possible de points chauds. Un cache sous-taillé par rapport au volume des points chauds est donc également gage de contre-performance. Car la politique de

[1] Note : Envisager de ne pas mettre un point froid en cache suppose de d'abord savoir qu'il s'agit d'un point froid. Le laisser dans un espace intermédiaire en attendant qu'il « chauffe » est une approche étudiée et suivie par plusieurs variantes de LRU [O'Neil 1993, Johnson 1994]. Ces approches impliquent que l'espace intermédiaire soit lui-même une sorte de cache pour permettre l'échange efficace de données, ce qui ne fausse donc pas notre constat.

remplacement qui a charge de les y maintenir ne peut en tout état de cause que se contenter de l'espace dont elle dispose.

4.1.2 Stratégies de placement et empreinte mémoire

Réduire l'espace de stockage du cache influence les stratégies de placement essentiellement sur la façon dont elles sont le mieux à même de gérer la fragmentation et/ou la pollution. Pour une telle stratégie, tout est question d'une ré-organisation efficace de l'espace lorsque la politique de remplacement choisit de remplacer un bloc par un autre. Selon, la stratégie choisie, l'effet n'est pas le même. Nous revenons dans cette sous-section sur chacune des stratégies présentées section 2.3.2, page 20, pour évaluer les enjeux à réduire l'empreinte mémoire du cache.

4.1.2.1 Impact sur l'allocation par taille variable

Comme déjà discuté, la stratégie par allocation de blocs de tailles variables souffre du problème de la fragmentation. Celle-ci est principalement causée par l'allocation/dés-allocation de blocs de tailles disparates.

Le principal vecteur d'efficacité de cette stratégie est de chercher parmi les blocs[2] , des tailles appropriées aux demandes d'allocation. Ces blocs libres ayant été formés par la dés-allocation de blocs d'autres tailles, ces variations peuvent donc avoir un effet boule de neige sur la fragmentation. Plus ces tailles varient, plus le taux de réutilisabilité s'amenuise et plus se forment ainsi ces « trous » , source de fragmentation.

De plus, ces variations sont souvent légères, de quelque octets. Ce qui augmentent donc la propension à perdre petit à petit de l'espace. Car des trous de 1 ou 2 octets sont difficiles à réutiliser. Il faudrait pour cela qu'ils puissent être fusionnés avec un bloc adjacent. Chose impossible tant que celui-ci n'ait pas été libéré à son tour. Si un trou est entouré de points chauds, il est encore moins probable que cela se produise.

En d'autres termes, l'allocation par blocs de tailles variables ne peut *a priori* jamais avoir un taux de remplissage optimal du cache, si ce n'est à quelques moment isolés dans le temps. De plus, réduire l'empreinte mémoire du cache réduit mécaniquement la profondeur des listes gérant les blocs libres et les blocs utilisés. D'une part, la politique de remplacement devra potentiellement supprimer plus de blocs, car ces chances de trouver un bloc assez large dès la première demande diminue. Et d'autre part, ceci réduit le nombre de choix possibles pour les algorithmes *First fit* ou *Best fit* pour trouver le candidat idéal à la ré-allocation (voir page 22).

4.1.2.2 Impact sur l'allocation par taille fixe

Comme nous l'avons vu, cette stratégie ne souffre pas de la fragmentation, mais de la pollution lié au bourrage. La quantité de ce bourrage dépend de la taille des blocs de données rapatriés en cache. Si tous ces blocs de données ont une taille unique, alors le bourrage est nul car dans ce cas, l'unité de stockage dans le cache peut être alignée sur cette taille. Dans les faits, ces cas sont très rares. Seul le cas d'une base de données peut correspondre, où chaque enregistrement d'une table à une taille fixe et invariable. Des

[2] *i.e.* libres, ceux libérés par la politique de remplacement. Dans le cas de cette stratégie, plusieurs blocs peuvent être supprimés tant qu'un bloc libre suffisamment large n'a pas été trouvé. Comme la politique de remplacement ne travaille généralement pas sur des critères de tailles, il n'est donc pas obligatoire que la stratégie de placement puisse à chaque fois produire un espace libre adéquate lors d'une première éviction. Dans ce cas, la stratégie de placement demande alors à la politique de remplacement qu'un autre bloc soit choisi pour être désalloué et ainsi de suite.

systèmes de base de données ont été portés dans ces cartes à puce (norme ISO-7816-7, PicoDBMS [Pucheral 2001], etc), et un cache par allocation de taille fixe est donc bien adapté.

Néanmoins, les blocs de données sont généralement plutôt de tailles variables, comme par exemple des blocs d'instructions, ou de multiples types de structures composant un modèle de données. Dans ces conditions, le bourrage prend plus d'importance.

En conséquence, le choix d'une unité de stockage n'est pas anodin et la pollution peut finir par être un facteur de contre-performance aussi préjudiciable que la fragmentation qu'elle évite. De plus, la pollution n'est pas proportionnelle à la taille de l'espace de stockage. Par exemple, si la pollution représente 5 % sur un espace de 4096 octets, soit 205 octets, alors un cache de 1024 octets qui aura au même instant la même pollution perdra quant à lui 20 % d'espace. Ceci est dû au fait que le volume de bourrage par bloc ne change jamais à un instant T du cycle de vie du consommateur de données, et ce quelque soit la taille du cache.

4.1.2.3 Impact sur la segmentation

La segmentation définit une organisation déstructurée de l'espace de stockage où un bloc de données n'est plus clairement identifié et peut se retrouver tronqué. Étant donné qu'elle est par définition indifférente au contenu, elle n'est donc pas influencée par la taille des blocs de données comme les autres stratégies. Au contraire, elle aura tendance à contenir plus de données, chaudes et/ou froides, mais également plusieurs bloc des données dont un seul est utile sur le moment. L'efficacité de la segmentation est donc tributaire des points chauds et de leur localisation dans la mémoire secondaire.

L'unité de stockage de la segmentation (*i.e.* une page de cache) est alors le facteur déterminant pour savoir si elle échange avec la mémoire secondaire beaucoup de points chauds ou beaucoup de pollution. Nous avions défini cette problématique, page 23, comme suit :

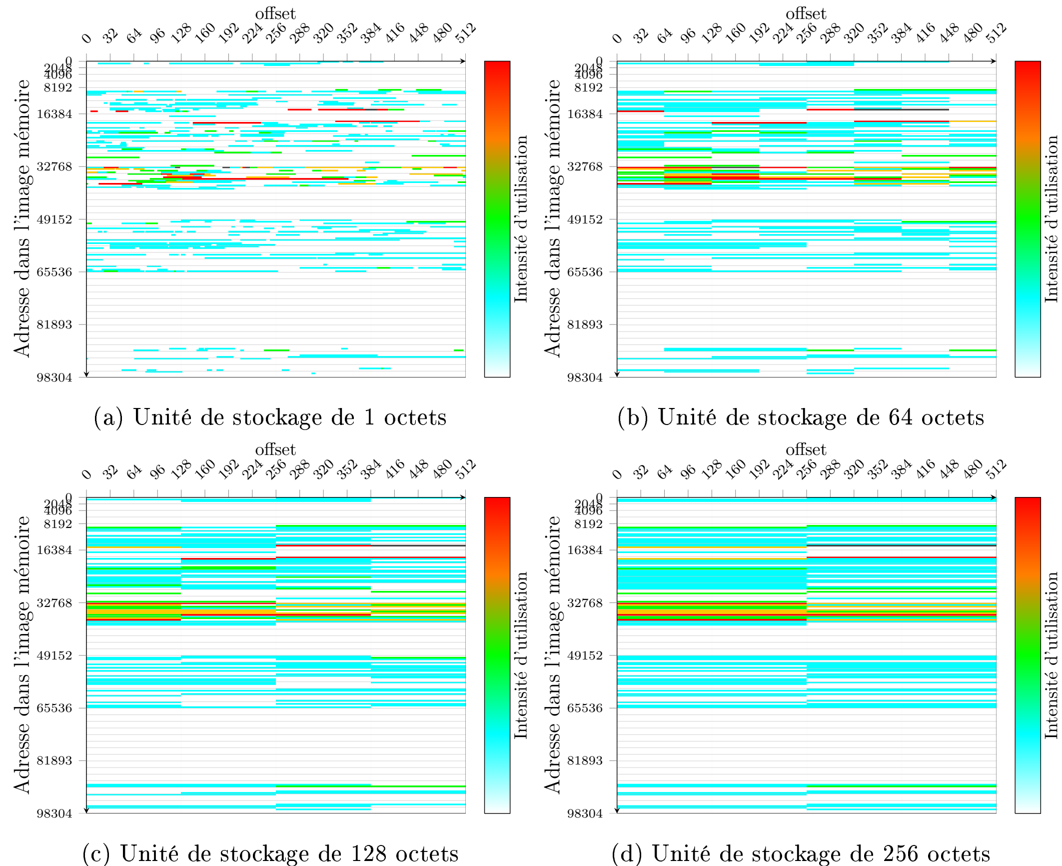
- plus une unité de stockage est large, plus la probabilité d'y trouver plus tard des données utiles est élevée, et ce sans le vouloir forcement au départ ;
- plus une unité de stockage est large, plus la probabilité qu'elle contienne de la pollution est élevée.

Par données utiles, nous pouvons entendre désormais « points chauds ». Par pollution, il ne faut plus seulement comprendre données froides, mais également données « blanches » , par nature complètement inutiles. En reformulant, le nouveau questionnement est donc celui-ci : la taille de l'unité de stockage du cache favorise-t-elle alors le maintien en cache des points chauds ou de la pollution ?

Nous avons vu dans nos exemples de cartographies, page 46, que les points chauds et froids sont le plus souvent petits, dispersés, et entourés par beaucoup de données « blanches ». Malheureusement, plus la taille d'une page de cache augmente, plus il est alors difficile de voir qui de la pollution ou des points chauds domine le contenu du cache. Pour évaluer comment une unité de stockage, donnée à l'avance, transforme la chaleur, en terme de pertinence et de durabilité dans le cache, nous avons présenté dans [Cogniaux 2011] ce que nous avons appelé la `dilution spatiale`.

Définition. *La* `dilution spatiale` *est la moyenne des chaleurs de tous les octets qui composent une page de cache de taille donnée. La* `dilution spatiale` *est donc un calcul permettant d'évaluer comment les points chauds et la pollution se mélangent, se dominent ou s'annulent pour une unité de stockage donnée.*

Une page de cache devient ainsi une page « chaude » , uniquement lorsque son taux de concentration en points chauds est élevé. À l'inverse, si ce taux est bas, la page est donc « froide » et riche en pollution.



(a) Unité de stockage de 1 octets

(b) Unité de stockage de 64 octets

(c) Unité de stockage de 128 octets

(d) Unité de stockage de 256 octets

Figure 4.3: Cartographies de la dilution spatiale sur le programme KVM.

À titre d'exemple, les figures 4.3 sont des cartographies de la dilution spatiale sur le programme KVM. La colorimétrie de ces cartes est la même que précédemment en terme de niveau d'utilisation. Cependant, pour comprendre les conséquences de la dilution spatiale, il faut noter que ces couleurs ne matérialisent plus la `localisation` des points chauds. Elles matérialisent désormais des `taux de concentration` d'une page en pollution ou en points chauds, qui vont du bleu au rouge, voire au noir.

En comparant 3 tailles différentes d'unité de stockage (*e.g.* 64, 128, 256 octets), on observe une augmentation constante de la pollution potentielle (en bleue) en terme de volume, proportionnelle à l'unité de stockage. À l'inverse, les pages les plus utilisées se « refroidissent » légèrement mais reste marquées. Leur volume totale collectable par le cache dans la mémoire secondaire augmente également, par assimilation des données adjacentes et des anciens points chauds et individuels.

Ce phénomène illustre donc la remarque que nous avions émise dans la section introduisant les points chauds. Dans le cas de la segmentation, la taille minimale idéale de l'empreinte mémoire du cache est contrainte par la taille de l'unité de stockage. En dessous de cette taille, chaque point chaud hors-cache déclenchera un nombre proportionnel à son

usage, donc par définition élevé, de défauts de cache.

Synthèse La réduction de la taille de l'espace de stockage du cache a un impact directement sur le potentiel de chaque stratégie de placement. Chacune a ses avantages, mais inévitablement, une empreinte mémoire faible accentue leurs inconvénients. Par les stratégies par allocation, une réduction d'empreinte augmente les risques et l'impact de fragmentation et/ou de pollution. Celles-ci couvrent même un paradoxe qui est que l'espace utile dans le cache se réduit encore plus vite que la réduction de la taille du cache lui-même.

Concernant la segmentation, c'est cette fois l'unité de stockage qui joue un rôle majeur en annonçant un seuil critique au-dessus duquel le cache dégrade subitement son efficacité lorsqu'il ne peut plus contenir suffisamment de points chauds.

4.1.3 Limite théorique du renouvellement

Nous avons présenté un état de l'art des politiques de remplacement section 2.3.3, page 23. Dans cette exposé, nous avons notamment présenté l'algorithme MIN qui fixe la limite théorique du nombre de défauts de cache. Par conséquent, mesurer les performances de cet algorithme fixe une limite au-delà de laquelle une politique de remplacement ne peut plus permettre à un cache de s'approcher de la limite physique de la Flash NOR.

La difficulté de cette évaluation tient dans les paramètres utilisés. Pour être efficace, une politique de remplacement a besoin d'un certain nombre d'entrées pour ne pas choisir trop brutalement des données à évincer. Par exemple et en simplifiant, LRU aura plus de faciliter à garder les 5 points chauds essentiels sur 8 entrées plutôt que 4. Pour augmenter le nombre d'entrées, deux leviers sont alors possibles : soit augmenter la taille du cache, soit réduire l'unité de stockage. Ce qui n'a pas du tout la même incidence, car dans le premier cas par exemple, un cache gagnera *a priori* plus en performance grâce à sa taille que par son renouvellement efficace.

Cependant, à notre échelle, seule l'empreinte mémoire domine. Il est donc plus intéressant d'évaluer les politiques de remplacement à taille de cache limitée en se focalisant principalement sur le nombre d'entrées.

Pour évaluer « le débit crête » de toutes politiques de remplacement (MIN), nous proposons de l'aborder le plus simplement possible en considérant dans un premier temps que tous les autres algorithmes du cache ont un coût nul. Ceci nous permet d'isoler MIN des interactions internes aux caches. Pour l'instant nous n'avons besoin que de la stratégie de placement. Nous n'évaluons toutefois que la segmentation car c'est la stratégie qui utilise le moins grand nombre d'entrées, qui est la seule sensible aux variations de l'unité de stockage.

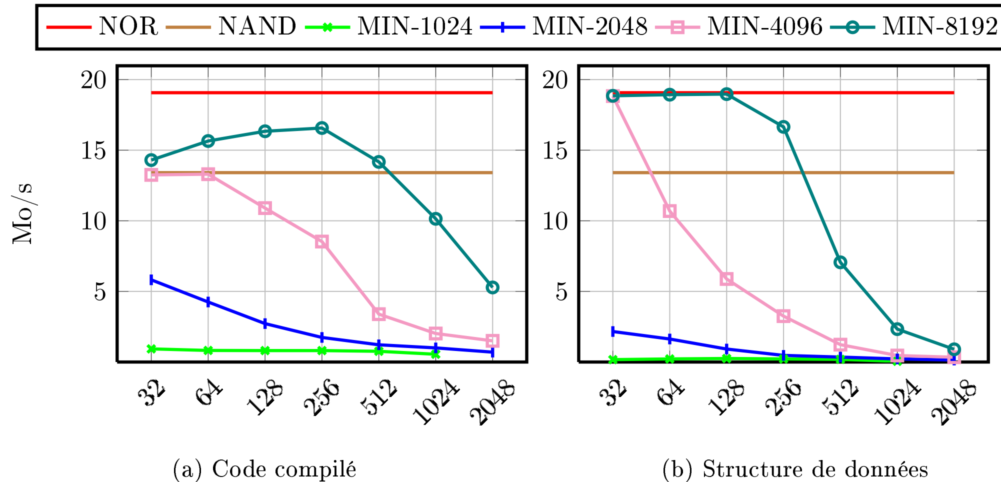
Nous mesurons donc les performances en terme de débit pour les comparer aux débits crêtes de la Flash NOR et de la Flash NAND. La formule est la suivante :

Équation 2. *Caractérisation du débit d'une politique de remplacement sur un système à 20MHz*

$$\begin{cases} D\acute{e}bit\,en\,Mo/s = (Mo_{Lu\,en\,cache})/(N_{Hit} * T_{RAM\,sec} + N_{Miss} * T_{NAND\,sec}) \\ T_{RAM} = 50ns\,\grave{a}\,20\,MHz\,et\,par\,octet \\ T_{NAND} = 145.6\mu s\,\grave{a}\,20\,MHz\,et\,par\,page \end{cases}$$

Les graphiques de la figure 4.4 illustrent l'évaluation de la politique MIN sur deux types de bloc de données : des blocs de base de code compilé, et des structures de données, en l'occurrence des méta-données Java.

Chaque courbe représente une taille d'espace de stockage particulière, *e.g.* 1024, 2048, 4096 et 8192 octets. Le débit crête de la Flash NOR est en rouge et le débit crête séquentiel



(a) Code compilé

(b) Structure de données

Figure 4.4: Évaluation de la politique de remplacement MIN

de la Flash NAND en brun. L'unité maximale de stockage est fixée à 2048 octets, soit la taille d'une page de Flash NAND.

Nous constatons que, de manière évidente, les performances s'améliorent avec la taille du cache. Cependant, il est encore plus intéressant de noter que celles-ci s'améliorent encore plus fortement lorsque le nombre d'entrées augmente, quelque soit la taille du cache. Ainsi, pour 4096 octets, le débit passe de 1,5 Mo/s avec 2 entrées de 2048 octets, à 13,3 Mo/s avec 128 entrées de 32 octets. Une autre remarque extrêmement intéressante est qu'un cache plus petit peut finir par rattraper les performances d'un cache plus gros, rien qu'en changeant la taille de l'unité de stockage.

Enfin, si l'on ne regarde que les très petits caches, avec une empreinte inférieure à 2048 octets, on constate que les débits de MIN sont très loin des débits de la Flash NOR et du potentiel de la Flash NAND. Ce point est certainement le plus intéressant pour notre problématique puisqu'il indique que même la meilleure politique de remplacement ne peut élever les performances de caches aussi petits. En réalité, ce phénomène est lié à ce que nous avons déjà présenté concernant les points chauds et la dilution spatiale. Si l'essentiel du programme ne peut être maintenu en cache et/ou si la pollution est trop forte, alors un cache ne peut être efficace.

4.1.4 Stratégies d'accès au contenu d'un cache

Cette sous-section évalue l'accès aux données du cache, en analysant plus particulièrement les algorithmes de recherche les plus utilisés dans l'état de l'art. L'algorithme de recherche est un facteur de performance important dans un cache, d'autant plus lorsque celui-ci est implémenté en logiciel. En effet, chaque recherche a un coût en cycles CPU, et donc en temps, qui constitue la latence du cache.

4.1.4.1 Algorithmes de recherche

Liste chaînée L'algorithme LRU, le plus utilisé comme politique de remplacement, est traditionnellement implémenté sous forme de liste chaînée, servant à la fois à la recherche et à l'éviction. Cette liste est continuellement triée pour garder un historique d'utilisation ordonné du plus récent accès au plus ancien. Pour ce faire, chaque accès au contenu du cache modifie le chaînage en déplaçant l'entrée nouvellement accédée vers la tête de liste

(soit, le temps $T_{GestionHit}$ dans l'équation 1, page 31). Le remplacement d'une entrée se fait alors simplement en modifiant la dernière entrée pointée dans la liste, puis en la déplaçant à son tour en première position (soit, le temps $T_{GestionMiss}$ dans l'équation 1). La complexité d'un défaut de cache est dans ce cas de $O(1)$, tandis qu'une recherche a une complexité de $O(n)$. L'implémentation de LRU sous forme de listes chaînées est donc optimisée pour la résolution d'un défaut de cache plutôt que pour une recherche.

Néanmoins avec un taux de *Hit* élévé, si 99,9 % des accès ne génèrent pas de défauts de cache, l'algorithme de recherche devient inévitablement un goulot d'étranglement. Une recherche itérative dans une liste n'est donc pas forcement le meilleur moyen de répondre au besoin d'un accès rapide. D'autant que d'autres algorithmes de recherche sont connus pour avoir une complexité moindre que la recherche itérative.

Il subsiste néanmoins une difficulté quant à l'ordre dans lesquelles sont triées les données. L'optimisation de l'algorithme LRU repose sur un triage temporel. Alors que les accès au cache se font par adresse et non par date. Il apparaît donc plus judicieux pour un algorithme de recherche que celui-ci maintienne un ordre sur ces adresses. Il semble également important qu'un algorithme de recherche n'ait pas à faire autre chose que de chercher, limitant si possible les temps de gestion des *Hit* et des *Miss*.

Deux candidats, parmi les plus connus, répondent à ces critères : la table de hachage et l'arbre binaire de recherche.

Arbre binaire de recherche Dans un arbre binaire de recherche, chaque nœud possède une clé et, au plus, deux nœuds appelés fils de tel sorte que le sous-arbre correspondant au fils gauche ne possède que des clés de valeurs strictement inférieures à la clé du père, et le sous-arbre de droite des valeurs strictement supérieures. La recherche dans un arbre binaire est comparable à la recherche par dichotomie et à une complexité de $O(log(n))$. Son parcours se fait récursivement par division de sous-ensembles de clés triées, vers la droite ou vers la gauche selon la valeur cherchée, en l'occurrence une adresse. Il existe plusieurs types d'arbre binaire de recherche aux complexités différentes pour les opérations de parcours, d'insertion, et de suppression. Un parcours efficace est assujetti à un arbre équilibré, mais cet équilibrage ralentit les deux autres opérations. Toutefois, ces deux opérations ne sont pas la priorité dans le cas d'une implémentation de recherche efficace dans un cache. Le meilleur compromis est offert par l'arbre dit « rouge et noir » , dont les propriétés d'équilibrage en temps constant, en font un arbre binaire de recherche intéressant, d'où son utilisation dans de nombreux systèmes temps réels.

Table de hachage Typiquement, une table de hachage est un tableau dans lequel on accède aux données par une clé. Cette clé est calculée par une fonction de hachage qui retourne toujours la même clé pour une même valeur qui lui est passée en paramètre. L'univers des clés possibles étant généralement plus grand que la taille du tableau, plusieurs clés correspondant à un même index dans le tableau peuvent entrer en *collision*. Pour garder les valeurs ayant les mêmes clés, le tableau est souvent un tableau de listes chaînées (où liste de seau, *bucket* en anglais), où les valeurs en collision sont ajoutées à la liste correspondant à la clé. La difficulté d'implémentation d'une table de hachage consiste donc à réduire le nombre de collision pour obtenir des listes de seaux courtes pour chaque entrée de la table. D'un point de vue complexité en temps, une recherche dans une table de hachage est la meilleur, en étant de $O(1)$.

Algorithmes	Nombre d'entrées				
	128	64	32	16	8
Liste chaînée	12	12	12	12	13
Table de hachage	13	15	18	24	36
Arbre rouge et noir	24	24	24	25	26

Tableau 4.1: Surcoût mémoire en octets par entrées.

Algorithmes	Caches	Nombre d'entrées				
		128	64	32	16	8
Liste chaînée	1024	+151,17%	+76,17%	+38,67%	+19,92%	+10,55%
	2048	+75,59%	+38,09%	+19,34%	+9,96%	+5,27%
	4096	+37,79%	+19,04%	+9,67%	+4,98%	+2,64%
Table de hachage	1024	+169,14%	+94,14%	+56,64%	+37,89%	+28,52%
	2048	+84,57%	+47,07%	+28,32%	+18,95%	+14,26%
	4096	+42,29%	+23,54%	+14,16%	+9,47%	+7,13%
Arbre rouge-noir	1024	+301,56%	+151,56%	+76,56%	+39,06%	+20,31%
	2048	+150,78%	+75,78%	+38,28%	+19,53%	+10,16%
	4096	+75,39%	+37,89%	+19,14%	+9,77%	+5,08%

Tableau 4.2: Surcoût mémoire en pourcentage par rapport à l'empreinte mémoire de l'espace de stockage du cache.

Algorithmes	Taille de Cache	Taille de pages				
		32	64	128	256	512
Aucun	1024	32	16	8	4	2
	2048	64	32	16	8	4
	4096	128	64	32	16	8
Liste chaînée	1024	23	13	7	3	1
	2048	46	26	14	7	3
	4096	92	53	29	15	7
Table de hachage	1024	18	10	5	3	1
	2048	42	24	13	6	3
	4096	88	51	27	14	7
Arbre rouge-noir	1024	18	11	6	3	1
	2048	36	23	13	7	3
	4096	72	46	26	14	7

Tableau 4.3: Nombre d'entrées disponibles en incorporant les structures de données des algorithmes de recherche dans l'espace de stockage du cache.

4.1.4.2 Empreinte mémoire des algorithmes de recherche

Les structures de données utilisées par les algorithmes de recherche et de renouvellement, introduisent un surcoût mémoire s'ajoutant à celui de l'espace de stockage. Cette occupation mémoire supplémentaire est fonction de chaque algorithme et dépend du nombre d'entrées que ceux-ci doivent gérer. Les *delta* de chacun sont rassemblés dans les tableaux 4.1, 4.2 et 4.3, page 53. Chaque colonne correspond à un nombre d'entrées gérées, pour les trois

algorithmes de recherche, donnés en lignes.

Dans le tableau 4.1 sont listés les coûts mémoires moyens d'une entrée de cache. Les structures de données correspondantes sont jointes en détail en Annexe C.

De manière mécanique, le surcoût mémoire total décroit avec le nombre d'entrée en cache. Le vrai problème intervient lorsque qu'il faut comparer ce coût supplémentaire avec des espaces de stockage réduit. En prenant plusieurs exemples de tailles d'espace de stockage (*e.g.* 1024, 2048 et 4096 octets), le tableau 4.2 détaille l'inflation que subit l'empreinte mémoire totale du cache. Plus le cache est petit, plus l'inflation est proportionnellement marquée.

Le problème que pose également ce surcoût mémoire est le biais qu'il introduit dans l'analyse globale d'un cache et qui est rarement adressé dans la littérature. Le plus important, en effet, dans le contexte d'une carte à puce reste la quantité de mémoire allouée au système de cache complet. Dans ce contexte, un système nécessitant un volume de 2048+1548 octets, bien que plus efficace, ne peut pas être comparé raisonnablement à un système ne requérant que 2048+108 octets (*i.e.* un cache de 2048 octets avec une liste chaînée de 128 ou 8 entrées). Pour annuler ce biais, les structures de données des algorithmes doivent être naturellement incorporées à l'espace de stockage. Même si cela doit réduire l'espace utilisable pour les données.

Dans ces conditions, il faut alors comparer un cache de 4096 octets sur par exemple une liste chaînée à 92 entrées ou un arbre à 72 entrées et non 128. Toutes ces réductions sont listées dans le tableau 4.3. Les formules de calcul sont jointes avec le détail des structures de données en Annexe C.

Ces remarques sont valables quelque soit la stratégie de placement. Cependant elles provoquent un effet de bord contraignant sur les stratégies par allocation. Pour celles-ci, s'il faut limiter le nombre d'entrée, alors l'espace de stockage utile peut énormément diminuer. Prenons par exemple, une liste à 26 entrées pour un espace de stockage de 2048 octets. L'espace pour les données descend alors à 1664 octets. Si maintenant, à un instant T, l'espace alloué est occupé par des blocs de données dont le plus gros est 32 octets, alors l'espace minimum perdu est de $(1664 - 32 * 26) = 832$, soit 41 % de l'empreinte mémoire totale.

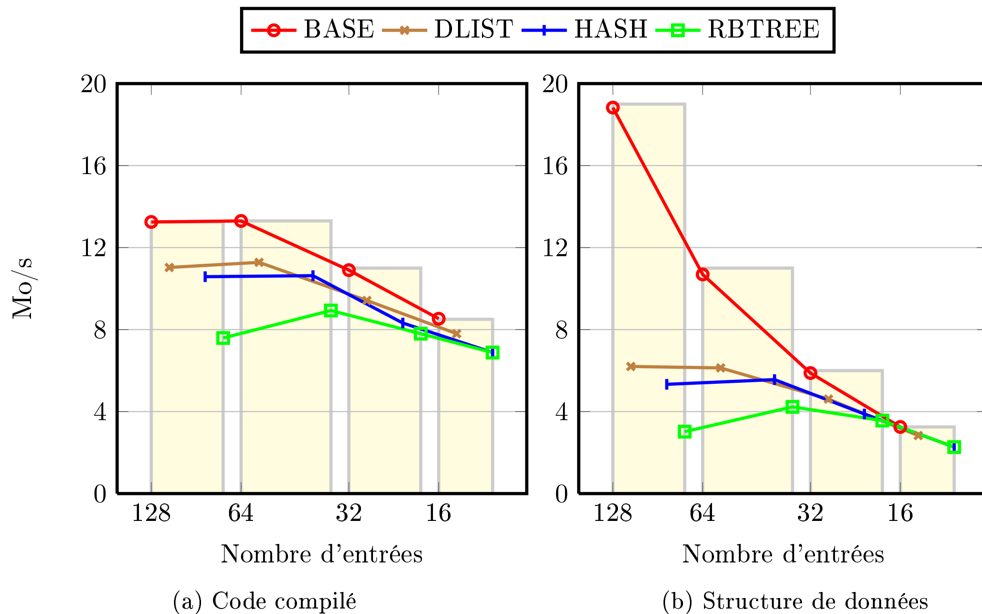
4.1.4.3 Débit pour des empreintes mémoires comparables

Un ré-évaluation des exemples de la section 4.1.3, page 50, est illustrée par les graphiques de la figure 4.5.

Car comme nous l'avons vu, le nombre de *Miss* dépend du nombre d'entrées dont la politique de remplacement de page a à disposition. Pour la segmentation, ce nombre d'entrée diminue mécaniquement la taille d'une unité de stockage, influençant alors les effets du phénomène de dilution spatiale.

La politique MIN est ré-évaluée sous les trois implémentations d'algorithmes de recherche standard. La différence avec les graphiques 4.10, page 66 tient uniquement dans le nombre de pages disponibles. Les bandes verticales orangée matérialisent la configuration précédente avec respectivement de la gauche vers la droites des unités de stockage de 32, 64, 128 et 256 octets. A l'intérieur de ces bandes se trouvent les débits de chaque algorithme avec leur nouveau nombre d'entrées à gérer, *i.e.* ceux énoncés dans le tableau 4.3. BASE représente la valeur MIN de référence des graphiques 4.10.

Les graphiques 4.5 montrent que le recadrage mémoire influence surtout la compétitivité de l'algorithme à arbre rouge et noir, le plus gourmand en mémoire. Nous notons également, que les complexités théoriques de chaque algorithme ne sont plus de si bons indicateurs lorsqu'on leur ajoute le paramètre mémoire comme nous le faisons ici. Ainsi, une liste



(a) Code compilé

(b) Structure de données

Figure 4.5: Ré-évaluation de l'algorithme MIN sur le nouvel alignement mémoire dans un cache de 4Ko.

chainée, moins gourmande en mémoire, permet alors d'avoir plus d'entrées à disposition et a donc une meilleure performance globale. Cependant, ce constat est pour l'instant hors coût d'exécution, et la complexité algorithmique abordée dans la section suivant y sera *a priori* plus prépondérante.

Ce qu'il faut retenir est que la compétitivité générale pâtit pour chaque cas, par rapport à BASE. De plus, le graphique 4.5b laisse apparaitre que la chute de performance peut être très importante dans certaines configurations. Ce cas démontre que quelques entrées supplémentaires peuvent parfois suffir pour emmagasiner le volume critique des points chauds, mais que l'inverse est malheureusement tout aussi vrai.

Ces graphiques montrent également que 3 implémentations possibles d'algorithmes de recherche ont des différences significatives, lorsque l'on met en parallèle le nombre de *Miss* évités (*i.e.* le débit) par rapport aux nombres d'octets mémoire dépensés, justement pour les éviter (*i.e.* le nombre d'entrées et/ou le type d'algorithme).

4.1.5 Coût d'exécution d'un cache logiciel

Après avoir abordé le coût mémoire, nous abordons maintenant le coût d'exécution d'un cache logiciel. Ce coût est prépondérant car il marque le temps de latence de ce type de cache du point de vue du consommateur de données et donc du système complet. Ce coût est essentiellement porté par la recherche de données en cache, la gestion des *Hit*, et la gestion des *Miss*, si la donnée cherchée n'a pas été trouvée.

Nous abordons tout d'abord brièvement le coût des stratégies de placement, puis nous analysons en détail les coût généraux d'un cache logiciel.

4.1.5.1 Coût des stratégies de placement

Les coûts de chaque stratégie de placement sont radicalement opposés. En effet, les stratégies par allocation ont un coût lié aux recherches d'une part dans les blocs libres pour

l'allocation, et dans les blocs utilisés pour la dés-allocation. Ce coût s'ajoute donc au coût de la gestion des *Miss*. Tandis que de son côté, la segmentation a un coût nul, car ce coût est pris directement en charge durant la désignation d'une victime par la politique de remplacement.

4.1.5.2 Coût en instructions

Les graphiques de la figure 4.6 présentent le CoI moyen des trois phases évaluées : recherche (4.6a), gestion des *Hit* (4.6b), et gestion des *Miss* (4.6c), page 57. Ces CoI sont des exemples obtenus sur le programme KVM.

Les tests évaluent l'algorithme LRU par une implémentation des trois algorithmes de recherche basés sur les structures présentées plus haut : liste doublement chaînée, table de hachage et arbre rouge et noir. Ces trois algorithmes seront également comparés avec l'algorithme FIFO basé sur une liste chaînée, la principale alternative à LRU hors académique. Les IOv sont ensuite agrégées suivant les trois étapes discutées précédemment : recherche, gestion des *Hit*, et gestion des *Miss*. L'algorithme MIN n'est ici pas évalué car il n'est pas implémentable concrètement.

Chaque graphique donne en ordonnées le nombre moyen d'IOv pour une des trois phases et par ICa. L'axe des abscisses est une variation du nombre d'entrées dans chaque structure de données évaluée (*e.g.* liste chaînée, table de hachage et arbre rouge-noir).

Ce paramètre est le paramètre fondamental car il détermine la profondeur de recherche à laquelle est confrontée l'algorithme. Ce paramètre agit donc directement sur la performance moyenne de l'algorithme, ce que n'exprime pas les complexités théoriques. De plus, une politique comme LRU modifie artificiellement la position relative des ICa entre deux accès en changeant l'ordre dans sa liste temporelle. Ce qui modifie donc potentiellement la profondeur de recherche de la même ICa entre un instant T et un instant $T + i$.

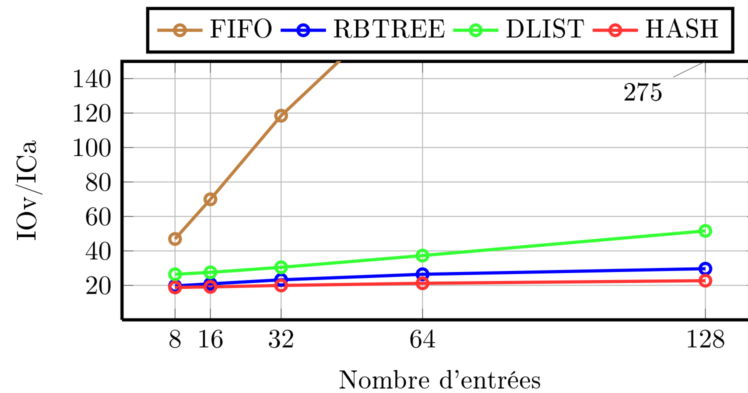
Bien qu'en utilisant le même algorithme de recherche, le CoI moyen d'une recherche peut donc légèrement varier d'un programme à un autre. Nous avons constaté des écarts de seulement 5 à 8 % pour rapport à la moyenne constatée pour le programme KVM que nous allons analysé maintenant, qui donne donc une bonne estimation à lui seul. Notons toutefois, d'abord, que les gestions des *Hit* et des *Miss* sont quant à elles constantes pour tout autre test car ces CoI ne varient que sur le nombre d'entrées et non sur le type de données mises-en-cache.

4.1.5.3 Analyse des coûts en instructions

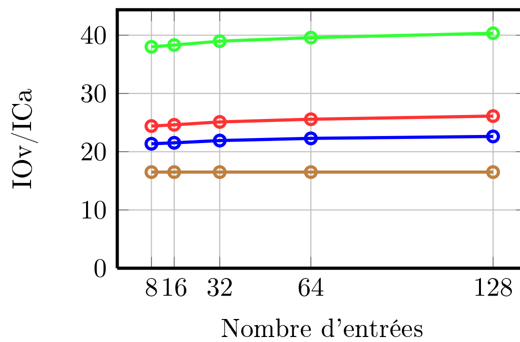
Recherche Comme nous pouvions nous y attendre, plus le nombre d'entrées est élevée, plus le CoI d'une recherche devient important (Graphique 4.6a). L'algorithme FIFO est de bien loin le moins compétitif car sa liste chaînée n'est jamais triée et la profondeur de recherche reste $O(N)$. À la différence d'une liste chaînée LRU triée, qui profite de la localité temporelle et où 90 % des données cherchées sont finalement trouvées en tête de liste. Contrairement à la complexité théorique, une liste chaînée triée par LRU a donc 9 fois sur 10 une complexité en $O(1)$, et non $O(N)$.

Les performances des arbres et des tables de hachage sont un peu meilleures. Leur CoI moyen croît moins vite que celui de la liste avec un nombre d'entrées plus élevé. On remarquera cependant que la liste chaînée est déjà deux fois moins performante que les deux autres, arrivée à 128 entrées.

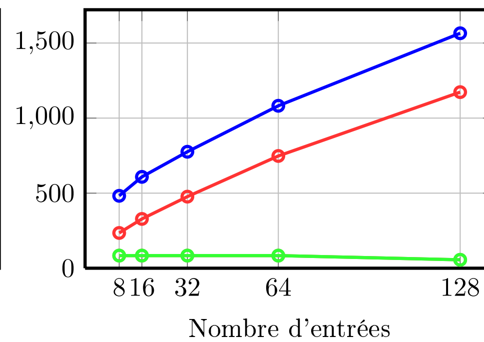
Gestion d'un *Hit* Pour la gestion d'un *Hit*, les CoI sont relativement stables par rapport au nombre d'entrées, pour tous les algorithmes. FIFO n'a rien d'autre à gérer que le renvoie



(a) Nombre d'IOv pour la recherche d'une ICa.



(b) Nombre d'IOv pour la gestion d'un Hit et pour une ICa.



(c) Nombre d'IOv pour la gestion d'un Miss et pour une ICa.

Figure 4.6: Exemple de CoI pour des blocs de données de type code compilé

d'une valeur trouvée ; son CoI est donc le plus faible. Pour LRU, le hachage et l'arbre doivent mettre à jour une estampille pour respecter la temporalité de ce dernier[3], tandis que la liste chainées doit rester triée. Ces obligations augmentent mécaniquement leur CoI par rapport à FIFO.

Gestion d'un *Miss* Le principal coût de gestion d'un *Miss* est la recherche de la victime à évincer. Le CoI de FIFO et d'une liste chaînée LRU sont identiques, constants et faibles car l'entrée à mettre à jour est déjà connue puisqu'il s'agit de la dernière.

La mise à jour d'une table de hachage ou d'un arbre est plus complexe car l'éviction et le remplacement requièrent au préalable de chercher le plus ancien élément parmi toutes les entrées (*i.e.* que nous réalisons par recherche itérative sur l'horodatage[4]). Leur CoI suit donc le nombre d'entrées.

[3] **Notes d'implémentation :**
Une table de hachage et un arbre trient des adresses et non des fréquences. Fort du constat que 90 % des accès se font dans la page courante, nos algorithmes de hachage et d'arbre contiennent une mémorisation du numéro de cette page courante pour leur permettre de profiter de cette localité temporelle.
Ensuite, bien que triés par adresses, c'est deux algorithmes maintiennent donc la notion de récence, propre à LRU, par horodatage sur chaque entrée.

[4] Nous aurions pu utiliser une autre table de hachage, par exemple, pour accélérer ce processus. Toutefois, l'empreinte mémoire totale aurait été trop importante, comme nous le discutons section 4.1.4.2

Synthèse De manière générale, on constate que l'utilisation d'un cache logiciel a un coût significatif. Dans le cas d'un accès à une donnée effectivement présente en cache, les coûts de recherches et de gestion des *Hit* s'additionnent, ce qui produit un CoI moyen de 45 à 65 instructions par accès au cache. Dans ces conditions, la latence du cache repose principalement sur ce coût. Celui-ci est toutefois moindre que l'exécution en place stricte ou avec un tampon, ce qui fait donc d'un cache logiciel un outil efficace pour réduire l'impact à utiliser une mémoire non-adressable. Cependant, cette latence met en évidence que la marge de manœuvre est encore grande et justifie la recherche d'alternatives au cache logiciel.

4.2 Cache d'instructions logiciel pour cartes à puce

Dans cette section, nous présentons des résultats mettant en évidence les forces et faiblesses d'un cache logiciel lorsque les blocs de données qu'il contient sont issus de programme compilé en code natif. L'objectif est d'évaluer un cache logiciel face à une classe de mise-en-application possible que représente un cache d'instructions, toujours avec comme challenge d'avoir une empreinte mémoire la plus faible possible.

4.2.1 Placement de blocs de base

Nous commençons par évaluer les propriétés des blocs de données qui influencent les stratégies de placement, à savoir leur taille et la fréquence de réapparition de ces tailles dans le temps. Les blocs de données étudiés ici sont les blocs de base d'un programme compilé. Ce que nous en dégageons est bien entendu équivalent pour des blocs de code semi-compilé de langages de haut-niveaux interprétés comme le Java ou JavaCard.

4.2.1.1 Principe de localité

Une étude approfondie sur le code et l'exécution d'un programme a été réalisée par P.J Denning [Denning 2005]. Dans cette étude, il définit ce qu'il a appelé le principe de localité. Ce principe se base sur le constat qu'un programme, quel qu'il soit, n'est jamais une suite arbitraire d'octets. Sur cette hypothèse, il a démontré qu'un programme expose plusieurs formes de localité et son résultat est le suivant:

- Un programme a majoritairement tendance à ré-exécuter du code ou ré-accéder à des données qu'il a déjà utilisé récemment : c'est le principe de localité temporelle.
- Un programme a également majoritairement tendance à exécuter du code stocké à un endroit plus ou moins proche de l'instruction qu'il vient tout juste d'exécuter : c'est le principe de localité spatiale.

Par exemple, les deux blocs de base composant un motif *if/else* sont très proches spatialement du bloc qui vient juste avant mais aussi celui qui vient juste derrière. Pour ce qui est de la localité temporelle, le meilleur exemple est la boucle où un ou plusieurs blocs de base sont ré-exécutés un certains nombre de fois dans un même enchaînement.

Le principe de localité nous apprend donc que l'exécution d'instructions n'induit pas nécessairement que des lectures aléatoires au sein de l'espace de code, mais répond plutôt à un schéma alternant une lecture aléatoire suivie de une ou plusieurs lectures séquentielles. Il s'agit donc du degré de séquentialité d'un bloc de donnée de type code compilé.

De plus, le principe de localité temporelle confirme que l'alternance entre blocs de base n'est pas complètement imprévisible et répond à des motifs d'exécution regroupant les blocs de base en grappes et dont l'exécution se répète dans le temps.

Programme	Couverture de code en octets	Taille moyenne des blocs de base	Taille max d'un bloc de base	Degré de séquentialité
Rijndael	12273	25	267	0,96
GSM	20726	27	302	0,96
SHA	9473	18	128	0,94
KVM	19154	19	184	0,95
Richards V7 C++	9790	16	117	0,94

Tableau 4.4: Couverture de code, blocs de base et `degré de séquentialité`

4.2.1.2 Tailles des blocs de base

Le support de lecture est fournit par les histogrammes de la figure 4.7, page 60. Ces histogrammes donnent la distribution des tailles de blocs de base pour quelques programmes d'exemple. Il s'agit ici des blocs de base du binaire exécutable ayant été exécutés au moins une fois, ce qui correspond donc à la couverture de code à l'exécution (voir deuxième colonne du tableau 4.4). Chaque programme montre une distribution plus ou moins similaire et fortement asymétrique vers la gauche. Ces histogrammes rappellent donc qu'un programme tend vers des blocs majoritairement de petite taille.

L'étendue de la distribution, c'est-à-dire l'écart entre le plus petit et le plus gros des blocs de base est très importante. Ce qui indique que les degré de séquentialité à l'exécution peut varier sur une grande amplitude pendant la durée de vie d'une application. Un programme peut ainsi également alterner de longues phases séquentielles avec d'autres beaucoup plus courtes.

4.2.1.3 Fréquences de réapparition de tailles de bloc

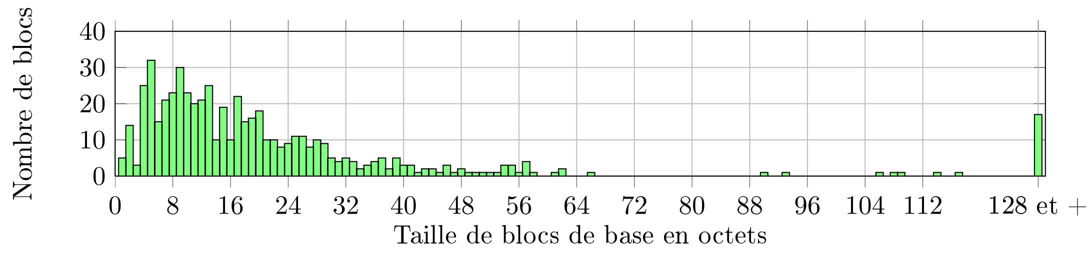
Le principe de localité repose explicitement sur le niveau de sollicitation de chaque bloc de base. La variation du degré de séquentialité au cours de la vie d'un programme est donc lié à ce principe, et plus particulièrement sur le principe temporel.

Une large boucle dans le graphe de flot de contrôle enchaînant de nombreux sauts vers de nombreux petits blocs de base, à l'intérieur ou à l'extérieur de la fonction en cours, produit pendant un certain temps un degré de séquentialité faible. Tandis qu'une autre boucle sur une longue séquence calculatoire produit l'effet inverse.

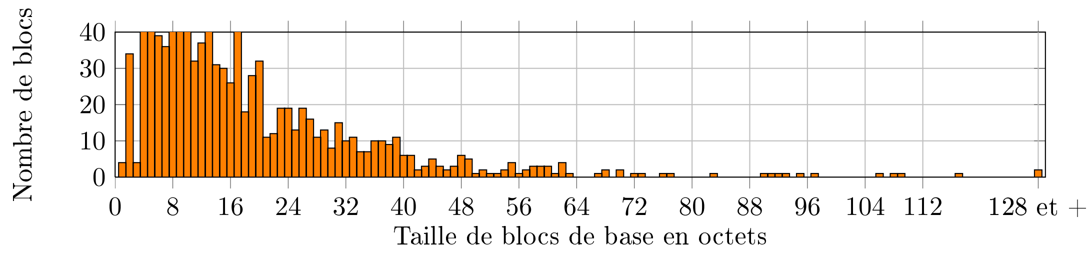
L'interpréteur d'une machine virtuelle est l'exemple du premier cas. Un calcul cryptographique est souvent un exemple du second. Bien-sûr, l'un comme l'autre peuvent alterner les deux.

Ces phénomènes sont mesurables en observant la fréquence de réapparition de tailles de blocs de base lors de l'exécution. Cette analyse permet d'évaluer les proportions de chacun des cas pouvant survenir durant la durée de vie du programme.

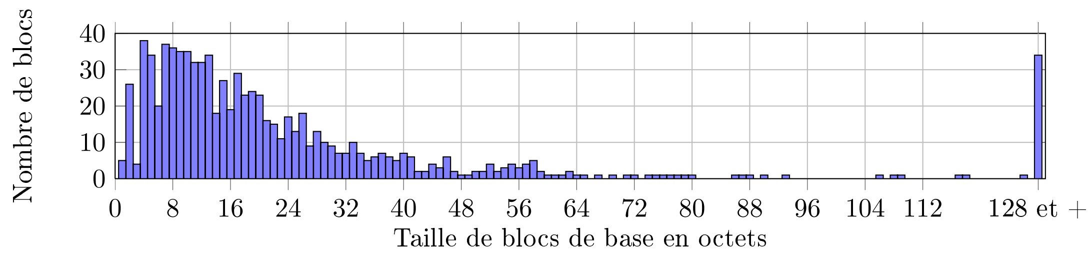
Les histogrammes de la figure 4.8, page 62, présentent à nouveau la taille des blocs de base mais cette fois en fonction de la fréquence de réapparition. L'axe des ordonnées représente cette fréquence, exprimée en pourcentage, pour tous les blocs de taille N, taille reprise sur l'axe des abscisses. La lecture qu'il faut avoir de ces graphiques est la suivante : prenons par exemple le programme SHA (figure 4.8d) où l'utilisation de blocs de base dont la taille est de 78 octets représente une fréquence de 29 %. Cela signifie que 29 % des blocs de base exécutés dans le programme SHA ont un degrès de séquentialité de 0,987. Si l'on regarde encore un peu plus près, on constate que le programme SHA exécute 72 % de blocs de base ayant une taille comprise entre 78 et 109 octets. À l'inverse, le programme KVM (figure 4.8b) alterne énormément de petits blocs. Dans son cas, 95 % des blocs de



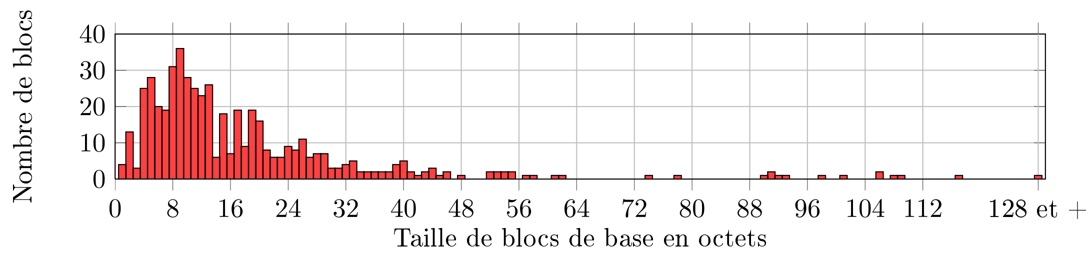
(a) Rijndael



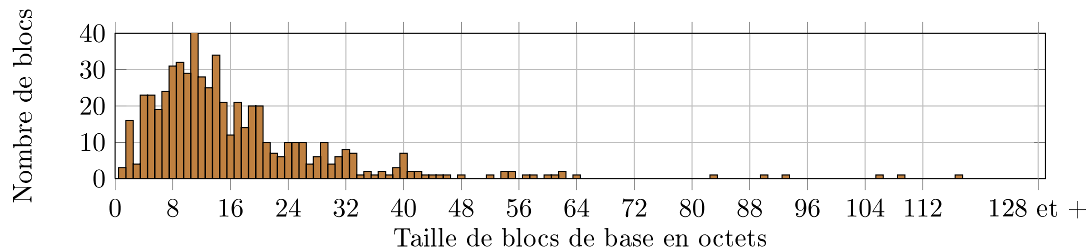
(b) KVM



(c) GSM



(d) SHA



(e) Richards V7 C++

Figure 4.7: Distribution des tailles de blocs de base en octets.

base exécutés ont une taille moyenne de 13 octets, ce qui équivaut à un degré séquentialité plus faible que celui vu dans le tableau 4.4, page 59, *i.e.* 0,92 pour 100 % de blocs utilisés.

Les distributions de ces fréquences étant très asymétrique, la moyenne est donc décalée, ici vers la gauche, ce qui la rend moins fiable. Pour les cinq programmes de test utilisés jusqu'à présent, le tableau 4.5, page 61, rapporte les fréquences de réapparition à des intervalles de population. Ces intervalles sont des recentrages de la moyenne à l'aide de fenêtres glissantes rattrapant le décalage constaté.

Par exemple, pour le programme Rinjdael, 95 % des blocs les plus utilisés ont une taille moyenne de 31 octets, mais 50 % de ces blocs ont une taille moyenne encore plus élevée, de 42 octets. Cela signifie qu'un bloc sur deux à un degré de séquentialité plus important que la moyenne. À l'inverse, pour le programme KVM, la taille moyenne diminue par intervalles de population, indiquant une tendance à exécuter des blocs plus petits.

Ces disparités entre intervalles permettent de mettre en évidence les tendances de chaque programme à se concentrer ou non sur beaucoup de séquentialité, et/ou la possibilité de présenter les deux profils. Les programmes KVM et Richards-V7-C++ ont ainsi un profil peu séquentiel, le programme SHA un profil très séquentiel, tandis que le programme Rinjdael montre une alternance des deux.

Pour conclure cette sous-section, l'analyse de l'évolution du degré de séquentialité durant le cycle de vie d'un programme met en évidence trois types de programmes :

1. des programmes très séquentiels ;
2. des programmes peu séquentiels ;
3. et des programmes alternant les deux premier types.

4.2.1.4 Dispersion des points chauds

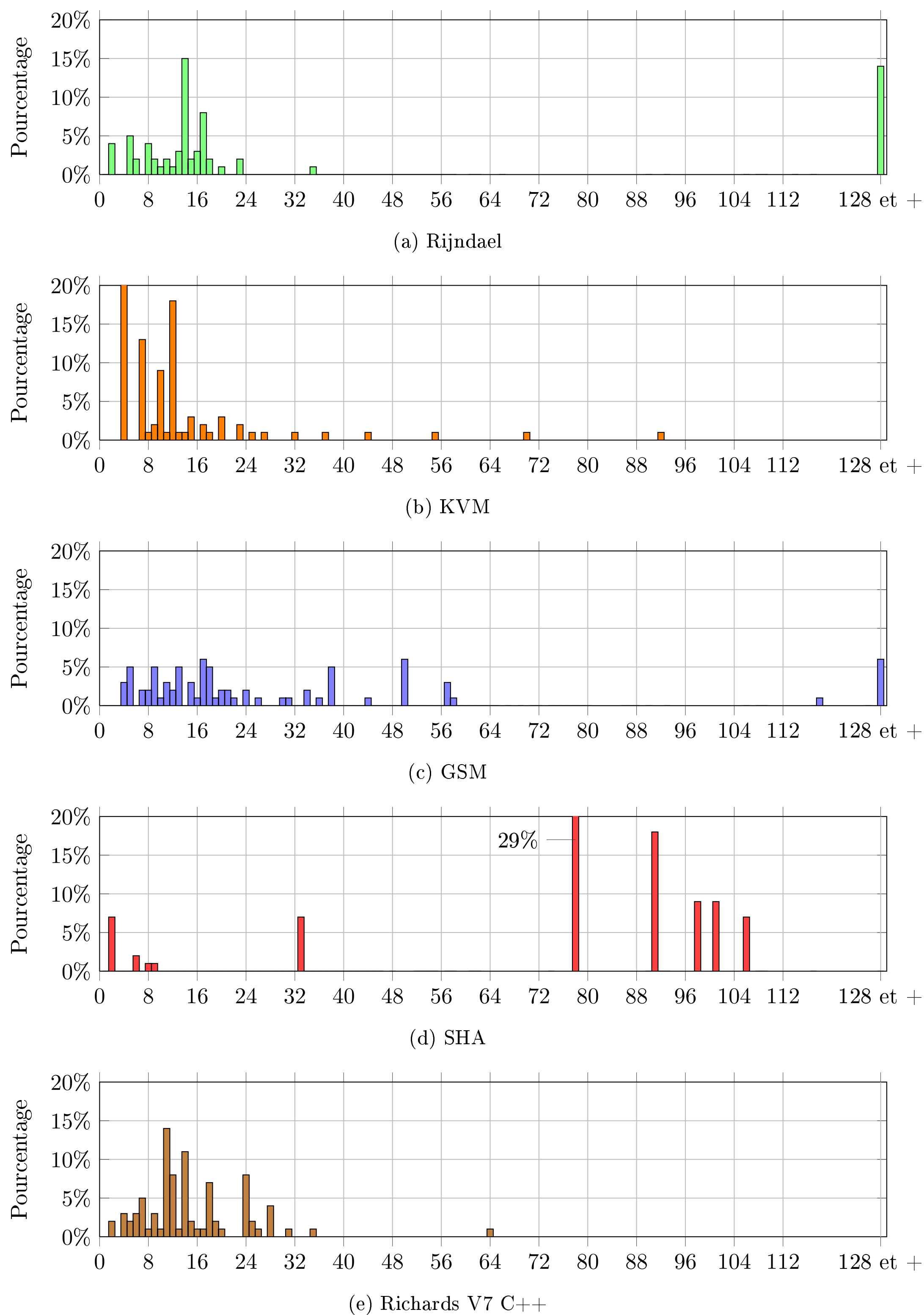
Dans le binaire, les points chauds ne sont par répartis de manière uniforme et sont très localisés du fait de leur petit nombre. De plus, ils ne sont pas forcement liés entre eux par des cycles dans le graphe de flot de contrôle. Par exemple, un bloc de base chaud peut déboucher sur deux directions à probabilité égale, divisant la chaleur des blocs d'arrivée par deux, et ainsi de suite.

Deux exemples de cartographie des points chauds sont illustrés dans la figure 4.9 (page 63, d'autres exemples sont disponibles en Annexe A).

On notera dans un premier temps que le nombre de points blancs est très importants. Ceci met en évidence que des processus de développement amenés à gérer tous les cas de figures possibles de l'exécution, ne correspondent pas à la réalité de cette exécution [Knuth 1971]. Le deuxième élément remarquable est la différence prononcée entre les deux exemples de cartographie reproduits dans cette figure 4.9.

Programme	moyenne 95%	moyenne 90%	moyenne 68%	moyenne 50%
Rijndael	31	31	36	42
GSM	30	29	30	37
SHA	71	74	78	85
KVM	13	11	8	7
Richards V7 C++	10	10	9	10

Tableau 4.5: Tailles moyennes des blocs de base, classées par intervalles de population.



(a) Rijndael

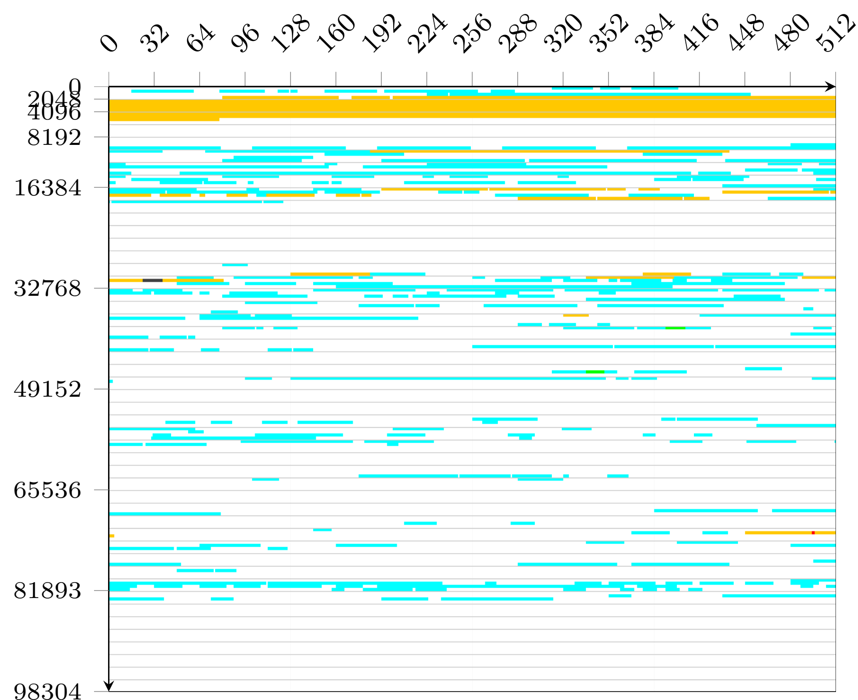
(b) KVM

(c) GSM

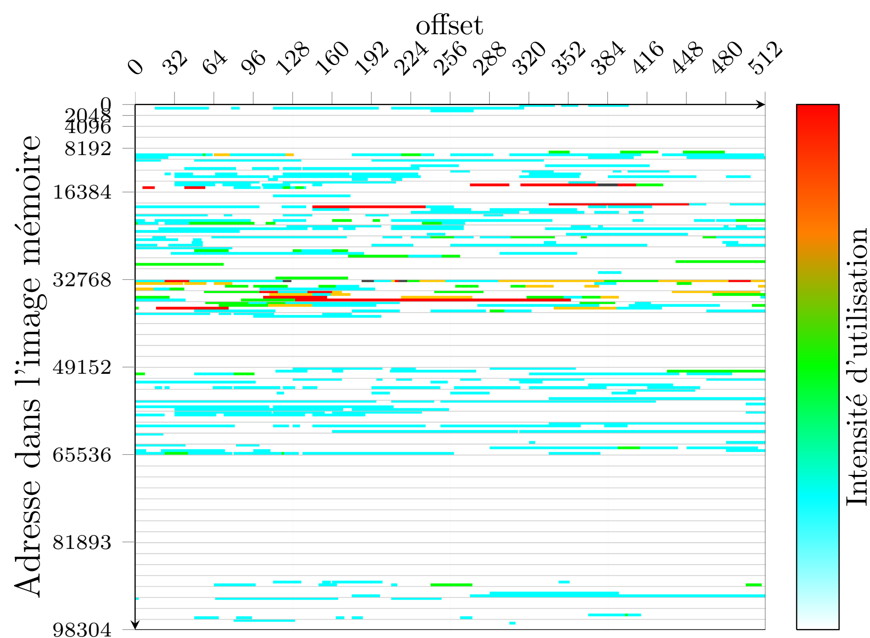
(d) SHA

(e) Richards V7 C++

Figure 4.8: Distribution de la fréquence de réapparition des tailles de blocs de base.



(a) Rijndael



(b) KVM

Figure 4.9: Cartographie des points chauds.

De la cartographie du programme Rijndael ressortent plusieurs zones. Une première zone fortement teintée d'orange,« tiède » , de l'offset 0 à 4096, et deux autres zones où l'on trouve à la fois des blocs tièdes et des blocs très froids. La première zone se concentre sur

seulement deux pages et plus précisément sur 3072 octets contigües. Alors que les deux autres zones sont très éparses. On y découvre une multitude de points froids répartis de manière très éclatée et disséminés sur plus d'une vingtaine de pages, ainsi que quelques points critiques très isolés.

La cartographie du programme KVM présente des similitudes au niveau de la dissémination des points froids, là aussi répartis sur une large plage. Mais une différence majeure est à noter. Il n'y a pas, en effet, cette large zone contigüe. Au contraire les points chauds du programme KVM sont plutôt isolés et dispersés sur un plus grand nombre de pages. Un rapide survol des cartes en Annexe A, montre la prévalence de ces deux tendances malgré quelques spécificités propres à chaque programme, correspondant bien évidemment aux spécificités de leur propre code.

Programme	bleu, <100	vert, <1000	orange, <10000	rouge, <100000	noir, >100000
Rijndael	11 589	41	4 677	2	14
GSM	15 391	13 801	4 785	549	0
SHA	8 240	27	374	522	78
KVM	21 193	1 433	620	754	40
Richards V7 C++	10 274	65	1 227	808	92

Tableau 4.6: Couverture de code par tranche de chaleur.

Programme	bleu, <100	vert, <1000	orange, <10000	rouge, <100000	noir, >100000
Rijndael	0.139%	0.027%	95.093%	0.203%	4.537%
GSM	0.337%	6.545%	56.367%	36.751%	0%
SHA	0.062%	0.01%	2.447%	64.814%	32.667%
KVM	0.317%	0.78%	6.01%	59.927%	32.966%
Richards V7 C++	0.108%	0.03%	22.432%	56.073%	21.358%

Tableau 4.7: Pourcentage de lecture dans chaque tranche de chaleur.

Les tableaux 4.6 et 4.7, page 64, complètent les informations issues des cartographies. Chaque colonne de ces tableaux reprend les seuils utilisés dans celles-ci. Le premier tableau donne le volume d'instructions par tranche de chaleur. Le second tableau reprend l'approche de Knuth montrant à quel niveau les points chauds forment l'essentiel des instructions exécutées.

On y voit que la tranche de blocs de base réellement froids (<1000) représente en général 90 % de la couverture de code mais au final très peu en utilisation. À l'inverse, les points chauds sont peu nombreux en volume, mais représentent un fort pourcentage de sollicitation comme l'a démontré Knuth. Toutefois, les niveaux ne sont pas réellement ceux attendus. Le programme Rijndael est même un contre-exemple, puisqu'il se concentre sur un plus large volume de données, diminuant ainsi la chaleur relative de chaque instruction.

Ces deux constats montrent les difficultés à pouvoir utiliser efficacement des caches à empreinte mémoire réduite, en mettant en évidence l'effet de seuil constaté dans la section précédente et concernant la taille minimale idéale d'un cache.

Celle-ci est toujours ici tributaire de la présence ou l'absence caractérisée de points chauds, ainsi que leur volume.

4.2.1.5 Synthèse

En résumé, des blocs de données de type code compilé sont largement de petites tailles et de tailles très variables. Ceci a alors un impact fort sur les stratégies de placement par allocation. De plus, ces variations sont souvent légères, de quelque octets. Ce qui augmentent donc la propension à perdre petit à petit de l'espace. Car des trous de 1 ou 2 octets sont difficiles à réutiliser. Il faudrait pour cela qu'ils puissent être fusionnés avec un bloc adjacent. Chose impossible tant que celui-ci n'est pas été libéré à son tour. Si un trou est entouré de points chauds, il est encore moins probable que cela se produise.

Nous constatons également que la dispersion des points chauds est très inégale. Ce constat a quand à lui un impact significatif sur la stratégie par segmentation, mais aussi sur les politiques de remplacement dans leur capacité à maintenir ces points chauds en cache.

4.2.2 Évaluation croisée du placement et du renouvellement

Avant d'évaluer le coût plus global d'un cache d'instructions logiciel, nous proposons une vue intermédiaire présentant l'intérêt d'une stratégie de placement ou de renouvellement par rapport à une autre dans notre contexte.

Les résultats d'une l'évaluation croisée des stratégies de placement et de renouvellement sont présentés dans les graphiques de la figure 4.10, page 66. D'autres exemples sont proposés en Annexe B.

Les graphiques de la colonne de droite sont des résultats basés sur un placement par segmentation tandis que ceux de la colonne de gauche sont des résultats issus de placement par allocations dynamiques. Une ligne de graphiques permet de comparer ces deux stratégies de placement pour un même programme. Pour chaque graphique, l'axe des ordonnées donne le résultat de chaque simulation en terme de débit, en Mo/s. L'axe des abscisses est quant à lui une variation de l'unité de stockage du cache. Pour l'allocation dynamique, un bloc de 1 octet représente l'allocation par tailles variables et les autres valeurs correspondent à des bloc de tailles fixes.

En résumé, chaque point de chaque graphique est le résultat d'une seule simulation[5] pour :

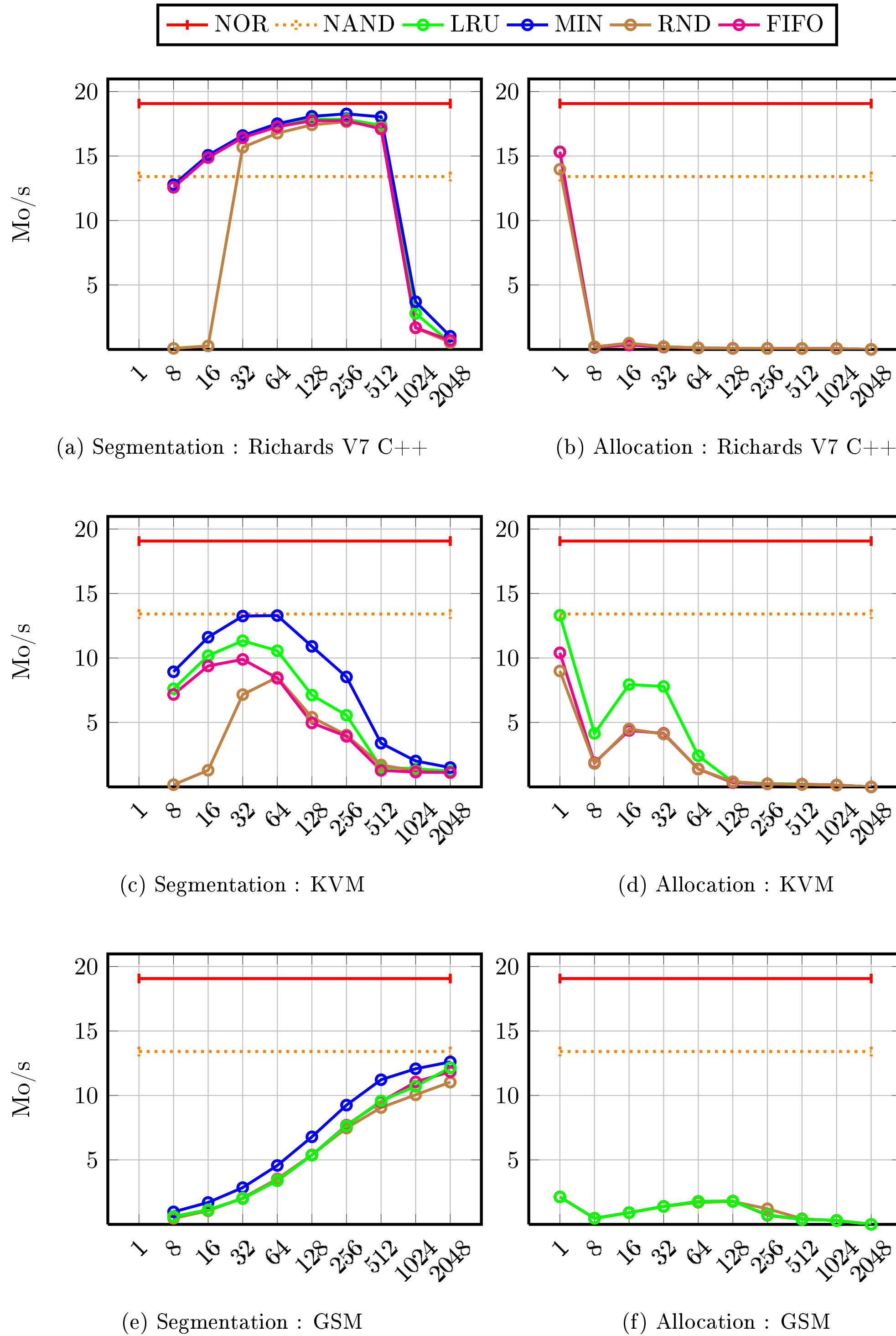
- un programme ;
- une unité de stockage ;
- une stratégie de placement - segmentation ou allocations ;
- une politique de remplacement (MIN, LRU, RaNDom ou FIFO).

Enfin, à titre indicatif, la borne supérieure formée par la NOR en terme de débit est représentée par une ligne rouge. Elle permet de visualiser et mesurer l'écart de débit entre une mémoire adressable et un cache d'instructions logiciel.

Le panel de la figure 4.10, page 66, fait état de comportements bien différents, en situation de taille de cache pourtant équivalente. Ces différences sont la conséquence des multiples combinaisons possibles de configurations de cache mais également des impacts sur le placement listé en début de section et dans la section précédente.

L'allocation dynamique n'est globalement par satisfaisante pour des petits caches, alors que montrée comme la plus intéressante mais à une autre échelle par Miller *et al.* dans

[5]Pour la politique aléatoire, il s'agit de la moyenne de 15 tirages.



(a) Segmentation : Richards V7 C++

(b) Allocation : Richards V7 C++

(c) Segmentation : KVM

(d) Allocation : KVM

(e) Segmentation : GSM

(f) Allocation : GSM

Figure 4.10: Évaluation des stratégies de placement et de renouvellement avec un cache logiciel de 4096 octets.

[Miller 2006]. Ceci est dû, à la fragmentation d'un coté et au bourrage de l'autre, qui prennent une part trop importante dans des espaces de stockage plus réduits. Leur impact n'est pas démontrable sur un seul graphique car il fluctue dans le temps et selon des circonstances principalement liées à la structure de chaque programme. Cependant, nous avons constaté que la fragmentation générée variait entre 10 et 40 %, que le bourrage occupé de l'espace à raison de 15 à 45 %.

Une autre vision d'ensemble des graphiques montre que les politiques de remplacement se valent quasiment toutes à notre échelle. On peut ajouter que contre-intuitivement, une politique basée sur le hasard rivalise, à quelques exceptions près, avec la politique MIN pourtant optimale. Certes, la différence relative entre les deux varie encore de 10 et 25 %, en dehors des exceptions visibles. Pourtant cet écart est globalement insignifiant car il est souvent plus petit de plusieurs ordres de grandeur que les différences amenées dans notre contexte par le changement de taille des unités de stockage. La politique de remplacement, seule, n'est donc clairement pas le levier d'action permettant de résoudre notre problématique.

Concernant la segmentation, colonne de gauche, on constate que celle-ci est dans la plupart des cas plus efficace que l'allocation dynamique, à défaut d'être toujours performante comparée à la NOR. L'approche classique de la segmentation dit que comme la lecture d'une page de Flash NAND est longue et donc coûteuse, la garder entière en cache est *a priori* une bonne idée comme le présentent Park *et al.* dans [Park 2004]. Malheureusement, cette approche ne passe pas l'échelle de petits caches.

– Premièrement parce qu'une page de Flash est une unité de stockage trop grosse, ce qui limite le nombre d'entrée dans le cache. Ce qui a pour conséquence réduire le champs d'action de la politique de remplacement lors de la recherche de victimes. Elle en vient donc à évincer brutalement du cache des pans entiers du programme.

– Deuxièmement parce que la dilution spatiale est trop forte dans de trop grosses pages et favorise ainsi plus souvent la pollution.

L'exemple du programme GSM, figure 4.10e, est un contre-exemple de ces deux poins en terme de résultats, mais non en terme de diagnostique. Dans son cas, de part la structure de son code source, l'essentiel de son code compilé n'est ni prononcé vers la pollution ni vers un fort taux de concentration en points chauds[6]. De plus, ce programme reste plus longtemps sur une même page de Flash, avant de passer à une autre, ce qui a pour effet de réduire le nombre de *Miss* en évitant un va-et-vient incessant entre des pages manquantes.

De manière générale, la segmentation est meilleure sur des unités de stockage plus petites qu'une page de Flash, mais aussi plus grosses que la taille moyenne d'un bloc de base. Ceci vient du fait que la segmentation profite du principe de localité car ce principe s'applique très bien à l'intérieur d'une page même plutôt petite, lorsque la dilution spatiale joue en faveur des points chauds.

À l'inverse, une stratégie par allocation est très mal adaptée pour un cache d'instruction logiciel lorsque son empreinte doit être faible.

4.2.3 Coût d'exécution d'un cache d'instructions logiciel

4.2.3.1 Unification des paramètres

Cette section porte désormais sur l'analyse du coût global qui est donne $T_{Accès\,moyen}$ de l'équation 1 (voir section 3.3.1.2, page 30) pour le contexte d'un cache logiciel pour instructions. Le coût complet d'un Miss est maintenant pris en compte, avec le coût total de sa

[6] Le lecteur peut se reportait aux cartographies associées figure A.1, Annexe A, page 137

pénalité incluant l'accès à la Flash NAND, la FTL et la copie de blocs de code dans l'espace de stockage temporaire.

L'unification de tous les paramètres est l'occasion de confronter les résultats des différents composantes d'un cache étudiées jusqu'ici, ainsi que les constats qui ont pu en être dégagés. Nous garderons ainsi la segmentation comme stratégie de placement car elle est la mieux adaptée à notre contexte. De la même manière que dans la section précédente, nous présentons des résultats sur FIFO et LRU, les politiques de remplacement éprouvées et de référence. D'autant que nous avons montré section 4.2.2, page 65, que le gain d'une nouvelle politique n'est pas suffisamment significatif dans notre contexte pour en proposer de nouvelles.

Enfin, nous avons également montré que, en dehors de la taille du cache, le paramètre tirant le plus vers le haut ou le bas l'efficacité était l'unité de stockage. En résumé des sections précédentes, modifier la taille de l'unité de stockage entraîne plusieurs phénomènes qui peuvent se cumuler ou s'annuler :

1. la variation du nombre d'entrées dans le cache ;

2. la dilution spatiale ;

3. la possibilité pour la politique de remplacement de manipuler plus ou moins d'entrées ;

4. l'augmentation ou la réduction de la profondeur de recherche d'une ICa.

4.2.3.2 Résultats expérimentaux

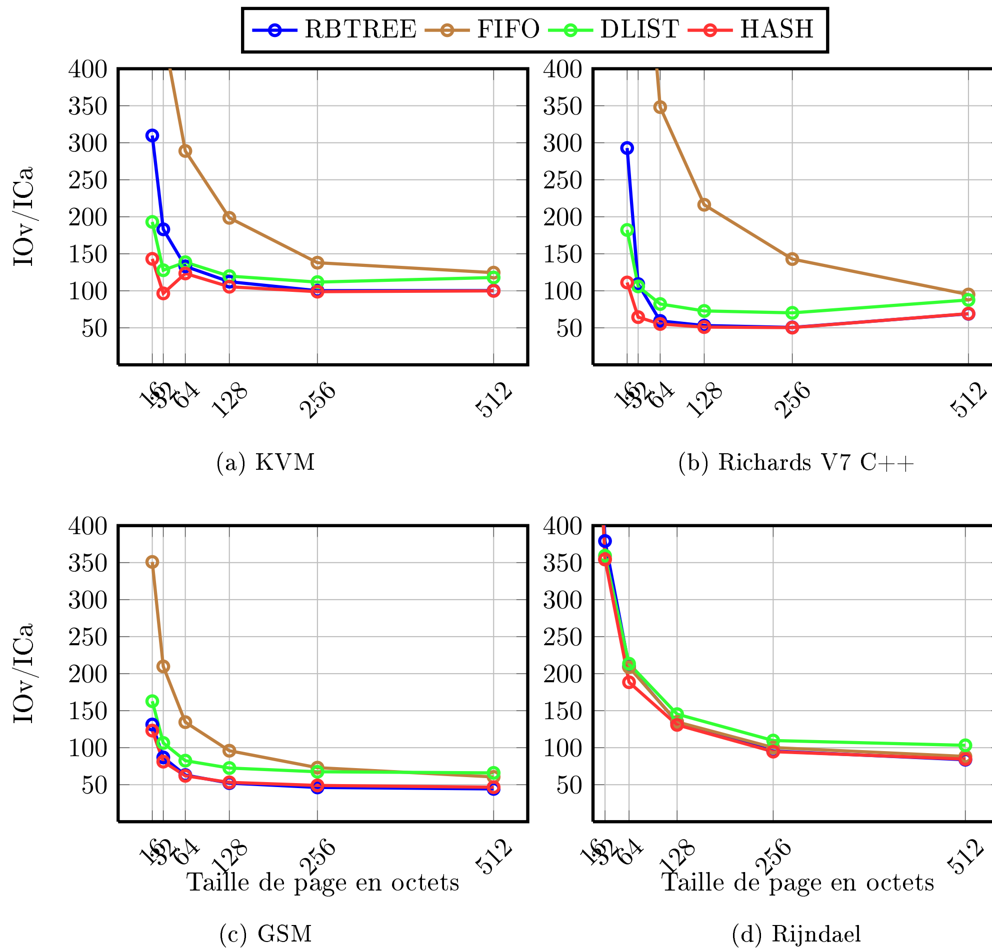
Les résultats sont rassemblés dans la figure 4.11, page 69, pour nos programmes de tests. Le protocole expérimental est même que pour la section 3.4.3. L'axe des ordonnées reste la mesure du CoI exprimée en IOv/ICa, et un point sur une courbe représente un CoI moyen caractérisant le temps $T_{Accès\,moyen}$. L'axe des abscisses est une variation du nombre d'entrées dans chaque structure de données évaluée (*e.g.* liste chaînée, table de hachage et arbre rouge-noir). Cependant, les nombres d'entrées données sur les graphiques sont par clarté ceux de « références ». Le lecteur peut se reporter au tableau 4.3, page 53, pour trouver la correspondance avec le nombre d'entrées du cas de figure mesuré.

Enfin, nous noterons que la taille de l'empreinte est ici de 4096 octets. Sous cette barre, nos programmes de tests sont beaucoup moins performants et les courbes moins explicites. Dans des caches plus petits, les tendances constatées restent les mêmes que celle présentées, mais il est clair que les programmes utilisés sont trop gros pour nos cibles à 1 ou 2 Ko de cache. Pour des programmes natifs de cette taille, il est donc recommandé d'avoir des systèmes matériels un peu moins restreints, comme par exemple des cartes à puce existantes avec 32 Ko de mémoire vive.

4.2.3.3 Évaluation et analyse du coût global

L'analyse globale est rendue difficile à la base par l'interaction de tous les paramètres de cache entre eux, et de la structure de chaque programme. Grâce à l'analyse individuelle de chacun d'eux menée dans les sections précédentes, il est maintenant plus aisé de dépasser l'effet boîte noire que représente un cache en terme d'ingénierie.

On constate ainsi que le CoI moyen d'accès au cache suit les courbes de débits de la section 4.2.2, page 65, mesurant principalement l'impact des défauts de cache. Au final, le CoI tend à se stabiliser autour d'une unité de stockage optimale comprise entre 128 et 256 octets.



(a) KVM

(b) Richards V7 C++

(c) GSM

(d) Rijndael

Figure 4.11: Coût en instructions de l'interface logicielle d'un cache de 4096 octets.

Ces points sont légèrement décalés par rapport aux résultats sur le débit du fait cette fois de l'interaction des algorithmes de recherche et du temps complet de la pénalité d'un *Miss*. Il est intéressant de noter que nos programmes de test tendent tous vers ce même point de stabilité malgré leurs structures, propriétés et différences respectives. Certes, tous n'atteignent pas le même niveau d'efficacité, mais leurs meilleurs niveaux respectifs tournent autour de ce point.

Ces résultats montrent également que les stratégies de recherche et de renouvellement LRU présentent désormais des différences mineures car leur coût est absorbé par le coût total.

Mais la principale information que ces résultats apportent est l'écart qui subsiste entre un cache et une mémoire adressable. Dans celle-ci, le processeur accède à une instruction en un cycle, alors qu'il lui en faut encore au moins 50 en moyenne avec un cache de notre échelle. Certes, un cache comble l'écart avec la NOR, puisque celui-ci se réduit par rapport au $2 * 2048$ octets de tampons comparables (eux-même 73 fois plus lent que la Flash NOR en moyenne).

Pour clore ce chapitre, nous terminons sur un dernier constat en lien avec le précédent qui sera fondamentale dans notre approche présentée chapitre 6. Le coût principal d'un cache implémenté en logiciel n'est pas la latence de la mémoire secondaire et les défauts de

cache, lorsque ce dernier est relativement bon dans cet exercice. Le goulot d'étranglement dans ce type de cache est le nombre d'accès.

La figure 4.12 présente pour chaque algorithme d'accès et pour chaque unité de stockage la part que représente dans le CoI d'accès moyen :

- une recherche avec succès et la gestion d'un *Hit*, en vert ;
- une recherche avec échec et la gestion d'un *Miss*, en brun.

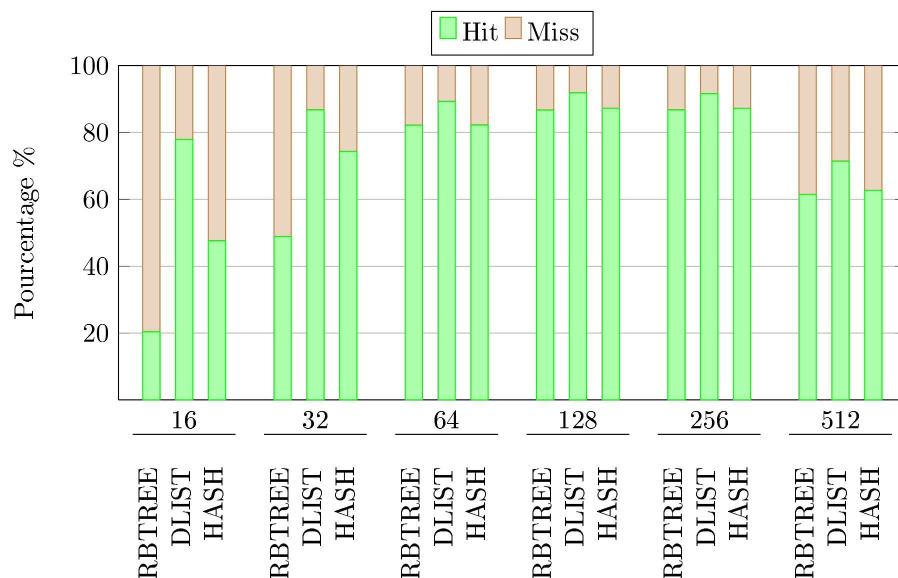


Figure 4.12: Répartition des coûts en instructions d'un succès et d'un défaut de cache.

Ces résultats proviennent cette fois du programme de test Richards-V7-C++, qui est le plus proche en terme de structure de code de ce que sont les applications JavaCard. On constate que le coût d'accès moyen pour les meilleures unités de stockage est largement dominé par l'accès à des données déjà présentes en cache.

Cela signifie qu'il passe donc le plus clair de son temps à servir et resservir des données présentes désormais dans la mémoire vive, elle-même adressable et accessible en temps constant. Ce qui est paradoxal, puisque c'est exactement l'endroit et les conditions les plus favorables pour la meilleure performance d'un opérateur d'exécution. Dans ces conditions, le cache forme donc bel et bien une barrière entre la lecture de ces données et leur présence effective en mémoire vive.

4.3 Synthèse et conclusions

Nous avons montré dans ce chapitre vers quelles performances tend un cache, sans jamais les atteindre, lorsqu'il est conçu entièrement en logiciel et lorsqu'il doit faire face aux contraintes mémoires d'une carte à puce. Notre revue de stratégies d'implémentations d'un tel cache souligne les difficultés à faire converger empreinte mémoire et coût d'exécution des algorithmes de cache vers de bonnes performances globales. Cependant, nous montrons qu'une telle approche permet de réduire de manière effective l'écart qui existe entre l'exécution en place depuis une mémoire adressable comme la NOR et l'exécution en place depuis une mémoire non-adressable comme la NAND.

Nous avons décliné notre démonstration sur une classe particulière de cache que sont les caches d'instructions. Notre objectif était d'apporter une réponse à notre problématique d'exécution d'applications stockée dans la mémoire non-adressable d'une carte à puce. Nous avons ainsi vu qu'il existe encore une marge de manœuvre importante entre ce que peut déjà apporter un cache, même réduit, et les performances d'une mémoire adressable.

Dans ce chapitre, nous avons également introduit la notion de degré de séquentialité d'un bloc de données dont le cache ne profite pas directement. Macroscopiquement, un cache contient, à un instant donné, une copie partielle, morcelée et désordonnée d'une image binaire comme par exemple un exécutable. L'information microscopique et utile, mesurable par le degré de séquentialité, et que plusieurs séries d'instructions contigües existent à des endroits précis de l'espace de cache à un instant T. À notre sens, ce décalage macro/micro constitue une piste importante d'amélioration des performances d'un cache.

En effet, si l'on regarde un cache en dehors de ces attributions (*e.g.* récupération, placement, renouvellement, etc), on constate que tout ce dont a besoin un opérateur d'exécution pour regagner en performance est devenue disponible en mémoire principale : un ou plusieurs blocs de données `adressables`. Hélas ces données sont inatteignables directement car le gestionnaire de cache masque cette propriété et modifie de plus le positionnement et la disponibilité de ces séries de données, du fait du renouvellement nécessaire et périodique du contenu du cache.

Il nous semble donc extrêmement intéressant d'exploiter cette piste qui semble prometteuse en essayant d'outre-passer la frontière hermétique que le cache implique entre le consommateur de données et les données déjà présentes en cache.

Chapitre 5

Cache logiciel de méta-données Java et JavaCard

Les langages interprétés comme le langage Java introduisent un type de données d'exécution particulier qui n'est pas du code. Ce type de données, ou `méta-données`*, décrit les données applicatives du programme Java, ou* `objets`*. Le méta-données sont regroupées dans un modèle relationnel structurant la façon dont elles sont organisées entre-elles. L'accès à ces méta-données est immergé dans le code sous forme d'opérandes complexes nécessitant un processus de résolution qui n'existe pas dans le code compilé. La conjonction entre code et méta-données modifie donc en profondeur les structures et motifs d'accès en cache identifiés dans le chapitre précédent.*

Si le code compilé a beaucoup été étudié pour des problématiques de caches d'instructions, les méta-données ne sont pas aujourd'hui documentées sur cet aspect, alors qu'elles représentent des données exécutables à part entière. Nous proposons dans ce chapitre une première approche possible pour combler ce manque.

Dans ce chapitre, nous nous attachons dans un premier temps à marquer ces différences en étudiant les méta-données dans leur modélisation et en confrontant plusieurs modèles de méta-données. Nous évaluons ensuite ces modèles de méta-données à la lumière d'une exécution depuis un mémoire secondaire non-adressable en reprenant les protocoles du chapitre précédent. Enfin, nous vérifions si le lien intime qui existe entre code et méta-données peut modifier les conclusions établies sur le code seul.

5.1 Introduction au modèle de méta-donnée Java

5.1.1 Désambigüisation

Dans ce chapitre, de nombreux concepts reposent sur l'utilisation du terme *donnée.* Afin d'éviter toutes confusions et pour lever dès maintenant toutes ambigüités, nous proposons d'abord une liste de définitions et d'acronymes que nous utiliserons dans la suite de ce chapitre.

Une donnée est une représentation conventionnelle d'une information en vue de son traitement informatique[1].

Une méta-donnée est une donnée définissant ou décrivant une autre donnée.

Une structure de données, est une structure logique destinée à contenir des données en les présentant sous une forme organisée destinée à simplifier leur traitement informatique. Cette organisation est un regroupement de `champs` qui peuvent être, soit des données, soit d'autres structures.

---

[1] Dictionnaire français Larousse, révision 1990

Une structure de méta-données, SMD, est une structure de données destinée à contenir des méta-données.

Un modèle de données définit de façon abstraite comment des structures de données, appelées entités du modèle, sont organisées entre-elles.

Un modèle de méta-données, MMD, définit de façon abstraite comment des structures de méta-données sont organisées entre-elles. Il ne doit pas être confondu avec un méta-modèle de données qui lui est un modèle qui décrit un autre modèle.

Le MMD Java est donc un ensemble organisé de structures de méta-données qui décrivent les propriétés et capacités de la structure de données Java de base qu'est l'*objet*, dans sa représentation en mémoire.

La spécification du fichier binaire *.class* Java est un MMD, structurant la représentation compilée d'une classe Java tel qu'elle est définie dans son code source.

5.1.2 Objectif et méthode

L'objectif de ce chapitre est d'identifier et étudier les inconvénients à stocker des méta-données Java dans une mémoire non-adressable, et d'envisager des pistes pour corriger ces inconvénients. Du point de vue de l'opérateur d'exécution qu'est la machine virtuelle, ces méta-données ne sont pas des données applicatives mais sont des données d'exécution autres que du code. À la différence du code compilé natif où seul le corps de la fonction contient des données exécutables, un langage interprété oscille entre code intermédiaire (*bytecode* Java) et représentation symbolique de certaines opérandes. Ces symboles sont des références parmi les méta-données que la JVM doit décoder pour continuer à exécuter une application. Pour découvrir et traduire correctement ces symboles et ces références, une JVM utilise donc un ou plusieurs modèles de méta-données.

Le MMD du fichier binaire Java est le seul modèle réellement imposé par les spécifications de la JVM. Une JVM est donc libre d'utiliser son propre MDD pour avoir sa propre organisation de méta-données, et donc ses propres méthodes de résolution de symboles ou d'architecture de création d'objets en mémoire.

Étudier les méta-données Java par une approche générale est rendue difficile par la multiplication des MMD. En effet, étudier comment une JVM utilise ses méta-données en étudiant les chemins d'accès empreintés dans le MMD dépend bien évidemment du MMD lui-même. Alors que ce MMD, dépend lui de la JVM.

Notre objectif n'est donc pas de proposer un MMD universel supportant le stockage dans n'importe quel type de mémoire. Notre objectif est plutôt de montrer comment un MMD impacte directement les performances d'une JVM, en bien ou en mal, en le confrontant par exemple à un autre contexte, plus ou moins contraignant que son contexte d'origine. Dans cette démonstration, l'accent est alors mis sur les forces et faiblesses d'un modèle conçu pour un type de mémoire en l'observant dans un autre contexte mémoire et enfin en tirer des conclusions.

Pour réaliser notre étude, nous avons choisi deux JVM embarquées aux MMD radicalement opposés pour ainsi mesurer l'impact de ceux-ci. La première est la JCVM de JavaCard 2.2 puisqu'elle est la VM Java conçue pour les cartes à puce. La seconde est KVM car cette JVM est disponible publiquement, respecte nombre de contraintes liées à l'embarqué et possède un MDD très semblable au modèle classique des spécifications Java standard.

KVM est l'implémentation de référence proposée par Oracle pour la catégorie J2ME, destinée aux systèmes embarqués types téléphone mobile. J2ME est la plus petite catégorie

de spécifications Java possédant encore toutes les fonctionnalités du langage Java standard, des chaînes de caractères aux flottants, en passant par les processus légers.

Pour présenter une comparaison de KVM et de JavaCard 2.2, nous commençons par introduire le MMD Java standard issu des spécifications Java. Ce MMD est le modèle de données pris en entrée par le processus de chargement de classe que ce dernier transforme vers le MMD propre à la JVM. Pour rappel, ce MMD de départ est le même pour KVM et JavaCard. En effet, le chargement de classe en JavaCard commence dans l'étape de conversion (voir section 2.2.2.2, page 17), qui transforme bel et bien le MMD de départ (*i.e.* des fichiers *.class* Java) vers le MMD propre à JavaCard 2.2, le fichier CAP.

5.1.3 Fichier de classe : méta-données brutes

L'unité de compilation en Java est la classe, abstraction supérieure des langages orientés *objet*. Une fois la classe compilée, son binaire contient alors le code de chaque méthode, au format intermédiaire propre au Java, et un ensemble de données symboliques, descriptives et non-compilées que nous avons défini comme étant les méta-données du langage en section 2.2.1.3, page 16, de notre état de l'art.

Le format du fichier de classe Java est normalisé et décrit dans les spécifications de la machine virtuelle Java [Lindholm 1999]. Il est structuré selon le MMD décrit par la figure 5.1, page 76.

Pour donner des exemples au vocabulaire défini précédemment, cette figure est donc un modèle de méta-données, composé de sept entités. À titre d'exemple, l'entité *MethodInfo* modélise la structure de méta-données correspondant à la définition d'une *Method* dans le fichier binaire Java. Chaque champs de cette structure est donc une méta-donnée. Dans l'entité *MethodInfo*, elles sont au nombre de cinq : quatre de type *unsigned short*, et une de type *AttributeInfo*.

Chaque fichier binaire contient donc une représentation structurée des attributs d'une classe définis dans son code source, et qui se répartissent en cinq catégories principales :

1. les propriétés de la classe elle-même ;
2. une table de symbole générique (ou *ConstantPool*) ;
3. les définitions symboliques de chaque champ ;
4. les définitions symboliques de chaque méthode ;
5. et le code de chacune de ces méthodes.

Cette figure 5.1 formalise les relations qui existent entre les éléments des cinq catégories de méta-données. Ils correspondent aux éléments constitutifs de la programmation orientée *objet* : la classe qui est le modèle d'instanciation d'un *objet*, les champs qui décrivent les propriétés d'un *objet*, et les méthodes regroupant les fonctionnalités et comportements d'un *objet* ou encore les modes d'interactions des objets entre eux et/ou avec le système.

La figure 5.1, page 76, est donc une représentation des méta-données « brutes » du binaire Java. La figure 5.2, page 76, propose une modélisation de leurs descriptions formelles établies dans les spécifications par [Lindholm 1999]. Ce MMD découle de la description de chaque structure de méta-données que ce document introduit et représente ainsi le MDD de base dont s'inspirent toutes les JVM.

Le MMD Java utilise des liens et chemins complexes entre méta-données et oblige souvent plusieurs indirections pour atteindre une propriété à partir d'une donnée de départ. De prime abord, et dans l'optique d'un stockage en Flash NAND, on peut noter que chaque

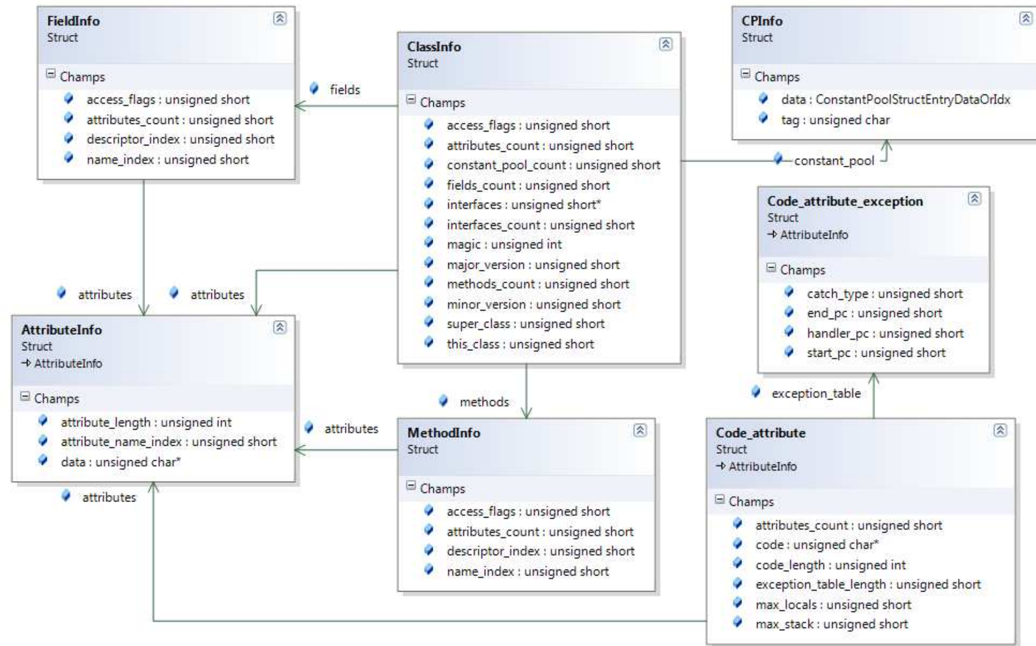


Figure 5.1: Structure du fichier de classe Java

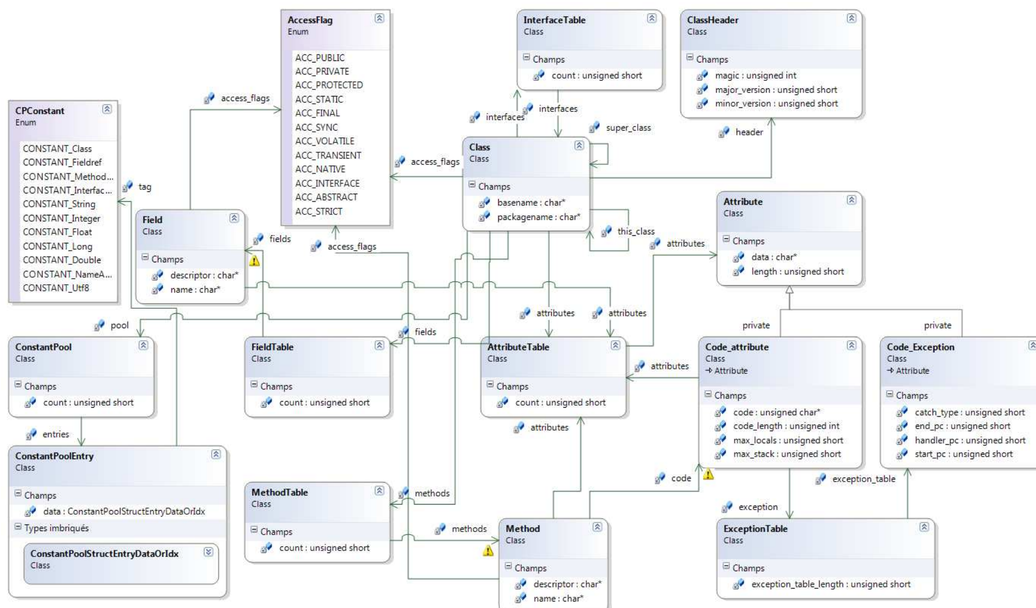


Figure 5.2: Modélisation UML des méta-données Java

indirection correspond à une ou plusieurs lectures et donc correspond à une succession de sauts aléatoires. Le nombre de sauts dépendra bien évidemment de la manière dont est implémentée la JVM, et dont est structurée le MMD.

Ces chemins et indirections ont comme point de départ, soit l'instruction en cours d'exécution, soit l'*objet* Java lui-même.

Accès par le code Le caractère semi-compilé d'un langage interprété comme le Java réside dans la particularité des opérandes de certaines instructions. Contrairement au code compilé où l'opérande contient le paramètre complet de l'instruction, dans le code semi-compilé l'opérande est souvent un index dans une table de symboles, ce qui nécessite une résolution parfois complexe d'informations.

Prenons un exemple. Dans le format du fichier de classe, l'opérande d'un appel de méthode est un index dans la table des symboles permettant de retrouver le nom de la méthode à appeler, disons M. Au sein de la classe et au sein du code, tous les appels à la méthode M pointent vers cet index. La contre-partie se situe derrière l'utilisation de l'index pour déclencher réellement l'appel à la méthode M.

La résolution du nom d'une méthode, sous la forme de nom de classe, nom de méthode, et signature (exemple: *"java/util/List"*,*"get"*,*"(I)Ljava/lang/Object;"*) est une recherche dans une arborescence d'index où les chaînes de caractères sont des feuilles et sont uniques. Ce parcours et la profondeur de cet arbre constitue donc un nombre important d'indirections mémoire.

Accès par l'object À l'exécution, la création d'un *objet* en mémoire dépend de la classe qu'il instancie. Toutefois, un *objet* en mémoire n'est pas une duplication de la structure de la classe et n'intègre d'elle que peu d'informations. Pour réduire l'empreinte mémoire d'un *objet* créé, celui-ci ne possède généralement qu'une référence vers la définition de la classe qu'il instancie. L'invocation d'une méthode sur un *objet* se fait donc d'abord par cette référence, ajoutant une indirection supplémentaire.

Nous pouvons donc formuler une remarque essentielle :

> *Par rapport au code compilé, où un bloc de base est strictement une suite de* **lectures séquentielles**, *un bloc de base semi-compilé perd cette propriété par le besoin de lire des données à des adresses différentes* **avant** *l'exécution du rupteur de flot flot de contrôle marquant la fin d'un bloc*[2].

5.2 Chargement des méta-données Java

Une machine virtuelle comme une JVM ne contient pas d'application et n'est qu'un opérateur d'exécution. Une application Java est une classe ou un ensemble de classes qui se présente sous la forme d'un composant enfichable dans la JVM qui en lancera l'exécution. Une étape préalable de chargement de l'application est donc nécessaire. Notons que cette décorrélation avec le reste de la JVM permet également un chargement dynamique et différé d'applications.

Le chargement d'une classe a principalement pour cible l'acquisition en toute sécurité des méta-données. La JVM évoluant dans un monde ouvert où des applications ou des classes peuvent venir de partout, elle ne peut faire confiance aux compilateurs, qui pourraient

[2]Seul l'appel de méthodes n'entre pas dans cette remarque. Les autres *bytecodes* utilisant des méta-données ne sont pas des rupteurs de flot de contrôle, au sens strict défini section 2.1.5.2, page 10

facilement produire du code malveillant. Le chargement d'une classe est donc codifié en trois étapes qu'une JVM fiable et de confiance se doit de respecter. Ces étapes sont la acquisition, l'édition des liens et l'initialisation.

5.2.1 Chargement de classes

Le chargement de classes peut avoir lieu à plusieurs moments durant l'exécution de la JVM. Elle est déclenchée en premier lieu lors du démarrage de la machine virtuelle pour charger les classes de base. Puis elle intervient au lancement d'une application pour charger les classes de cette application mais aussi leurs dépendances. Enfin elle peut se produire lors d'appels explicites au chargement dynamique de classes pendant l'exécution.

5.2.1.1 Acquisition des méta-données

Cette étape consiste principalement à réserver un espace mémoire pour les méta-données lues depuis le fichier de classe. Cette phase du chargement commence par une pré-vérification analysant la structure et l'intégrité du fichier de classe conformément au modéle détaillé figure 5.1. Si le fichier est correct, son contenu est transformé dans le modèle de méta-données propres à la JVM.

5.2.1.2 Édition des liens

L'édition des liens est une étape de post-compilation réalisée en-ligne, alors qu'en natif cette étape est réalisée dès la compilation. Cette étape finalise la compilation hors-ligne qui a crée le fichier de classe pour lier les classes entre elles et résoudre les dépendances trouvées en parcourant la table des symboles.

Comme signalé, une JVM n'a aucune garantie que la classe en cours de chargement ait été produite par un compilateur respectant les standards du langage. Dans un autre cas de figure, une classe peut avoir été produite correctement, mais une de ses dépendances, correcte à la compilation, peut avoir été modifiée entre temps, brisant ainsi la compatibilité entre classes. L'édition des liens est donc une obligation et une condition *sine qua none* à toutes utilisations de nouvelles classes par la JVM.[3]

L'édition des liens comportes trois étapes, qui doivent impérativement être exécutées dans l'ordre :

Vérification L'étape de vérification s'intéresse principalement à l'intégrité du code et son innocuité. Cette étape reprend toutes les problématiques et techniques déjà présentées section 2.1.5.2, page 10.

Préparation La préparation est une étape de pré-initialisation allouant l'espace mémoire pour les champs statiques et affectant les valeurs constantes. À la différence de l'initialisation, qui intervient plus tard, aucun code Java n'est pour l'instant exécuté.

Résolution L'étape de résolution, plus ou moins optionnelle, consiste à déterminer les valeurs propres à la JVM de tout ou partie des références symboliques du *ConstantPool*. Cette étape peut ainsi être l'occasion d'une nouvelle transformation appliquée au modèle de méta-données interne à la JVM. Par exemple un symbole résolu peut ainsi remplacer

[3] À noter, que chaque étape de l'édition des liens peut alors être initiatrice du chargement d'autres classes, correspondant à des dépendances n'ayant pas encore été chargées. Le succès du chargement de la classe en cours est alors conditionné par le succès du chargement de ses dépendances.

un ou plusieurs champs du modèle, voire créer de nouvelles structures comme par exemple une table de méthodes virtuelles, une nouvelle version réarrangée du *ConstantPool* (i.e. jugée mieux adaptée à l'exécution par le concepteur), un *ConstantPool* commun à tout le système, etc.

Notons enfin que la résolution peut très bien intervenir tardivement, c'est-à-dire uniquement au moment où cela est nécessaire, voire même sans aucune mémorisation de la résolution et donc avec une résolution rééditée à chaque fois que cela est nécessaire.

5.2.1.3 Initialisation

L'initialisation consiste à exécuter l'initialiseur statique Java de la classe, si il est présent. L'initialisation statique se fait au moment opportun, c'est-à-dire au libre de choix de la JVM, mais une seule fois, et sur toute la hiérarchie de classes dont la classe courante est la plus basse et en commençant par l'ancêtre non-initialisé le plus éloigné.

Un champ statique d'une classe étant un *objet* Java unique, l'initialisation est une condition préalable à toutes utilisations de la classe en cours de chargement, qu'il s'agisse de création d'une nouvelle instance, d'un accès à un de ses champs statiques ou une de ses méthodes statiques. La classe n'est considérée chargée et instanciable que lorsque cette dernière étape a eu lieu.

5.2.2 Pré-chargement de classes

Dans les architectures de machines virtuelles scindées (SVM), classiques dans les systèmes embarqués de petites tailles, le chargement de classes est effectué sur une station de travail, en dehors de l'environnement d'exécution de la JVM (*i.e.* une carte à puce, un système embarqué communiquant,...) et donc en dehors de la VM même malgré l'appellation de SVM. Cette approche est celle utilisée par JavaCard, et est une des options suivies par plusieurs JVM embarquées comme KVM [Simon 1999], Squawk [Simon 2006], Darjeeling [Brouwers 2009] ou JITS [Courbot 2010] pour ne citer qu'elles.

Ces technologies utilisent deux approches possibles offertes par une SVM. Soit l'utilisation d'un format de fichier dédié et pré-chargé comme le fichier CAP JavaCard, ou alors une approche dite par *Romisation* ou chargement précoce.

5.2.2.1 Chargement précoce

La *Romisation* est une technique qui consiste à charger à l'avance des méta-données directement sous la forme qu'elles prendront à l'exécution dans la JVM cible. Cette image, ou « Rom » , est une super-structure C générée hors-ligne par un outil spécialement conçu pour et fonctionnant comme un chargeur de classe. Une instance de cette super-structure C alimentée statiquement par les données des fichiers de classes est ensuite compilée directement dans le binaire de la JVM.

La *Romisation* est donc un pré-chargement global de classes où toutes les méta-données du futur système sont rassemblées dans un seul conteneur. Elle offre ainsi l'avantage de déporter un processus coûteux hors du système embarqué cible. La *Romisation* n'empêche par l'ajout de nouvelles classes sur le système cible post-déploiement mais bien souvent l'image a pour but d'être placée dans une mémoire non-inscriptible. Si tel est le cas, la mécanique de chargement de classes peut alors elle aussi être supprimée de la JVM cible et libérer ainsi beaucoup d'espace. Car dans ces conditions, il n'est plus besoin de revérifier de code à chaque re-démarrage puisqu'il n'est plus jamais modifiable.

La *Romisation* est aussi souvent l'occasion de certaines optimisations dont la principale est le compactage du *ConstantPool*. L'idée directrice est de le rendre commun à toutes

les classes de la Rom, éliminant ainsi les redondances de chaines de caractère et libérant énormément de place dans le système cible.

De plus, dans une image romisée, puisque toutes les classes du système sont présentes, la résolution des symboles et l'éditions des liens entre classes peuvent être poussées à une phase très avancée. Dès sa fabrication, il devient aussi possible de spécialiser la Rom par la suppression des classes, méthodes ou chaînes de caractères qui ne seront jamais utilisées [Courbot 2010].

5.2.2.2 Format pré-chargé

Un format de fichier dit pré-chargé quant à lui annule et remplace le fichier de classe au format standard. Généralement en prenant plusieurs classes à la fois, mais jamais l'ensemble complet des classes du système comme avec la *Romisation*. Le fichier pré-chargé est donc un morceau de Rom avec les mêmes propriétés et possibilités que cette dernière. Le fichier CAP de JavaCard par exemple est la *Romisation* d'un paquetage Java complet.

Les classes de ce paquetage sont donc chargées hors-ligne et leurs méta-données agencées dans un format permettant leur exécution directe par la JVM JavaCard. Le mode de génération de ce type de fichier est la conversion, mode présenté section 2.2.2.2, page 17.

L'avantage du format pré-chargé par rapport à la *Romisation* est qu'il permet l'ajout d'applications post-déploiement de la JVM et du système cible sans réel chargeur de classe. Le fichier étant déjà pré-chargé, la JVM n'a pas besoin d'effectuer toutes les étapes décrites précédemment. La conversion prend en effet en charge la résolution symbolique à l'intérieur d'un paquetage Java et prépare déjà l'initialisation des champs statiques.

Toutefois, un certain nombre d'étapes doivent quand même être réalisées en ligne comme la re-vérification du code et les liaisons inter-paquetages. Bien que ces dernières soient simplifiées par la création du fichier d'export (voir une nouvelle fois section 2.2.2.2).

5.2.3 Synthèse

Contrairement au code compilé, le problème des méta-données n'est plus un problème d'adressabilité. Néanmoins, stocker des méta-données comme celles de Java dans une mémoire non-adressable et imaginer les aborder comme du code ou des données classiques est impossible.

- d'abord, simplement parce que leurs formats sont différents ;
- ensuite parce que leurs motifs d'accès, fait de plusieurs indirections, tout en étant immergées dans le flot d'exécution, forcent la VM vers beaucoup plus de lectures aléatoires et surtout brise la propriété de séquentialité d'un bloc de base ;
- et enfin parce que le modèle de données guidant ces motifs d'accès est dépendant de constructions propres à chaque implémentation de JVM.

Pour évaluer ces différences, nous proposons maintenant une présentation de deux MMD, en commençant par celui de la Rom KVM, qui sera suivi d'une présentation du MMD pré-chargé JavaCard, le fichier CAP.

5.3 Modéle KVM

Le modèle de méta-donnée de KVM décrit dans la figure 5.3 est relativement proche de celui-ci édicté par les spécifications de la JVM standard, page 76. Il utilise par exemple le même schéma par tableaux pour associer méthodes et champs à une classe.

Cependant, le modèle de KVM utilise d'autres champs de méta-données qui lui sont propres. Un exemple est le champ *instSize* de *instanceClassStruct* qui résulte du pré-calcul du nombre de champs[4] qu'un *objet* instanciant une classe possédera. Ce pré-calcul permet d'allouer rapidement l'espace nécessaire au stockage de cet *objet* dans le tas de la JVM. Le même mécanisme est utilisé dans *methodStruct* pour le champ *frameSize* qui permet d'allouer rapidement sur la pile l'espace nécessaire pour l'exécution d'une méthode.

KVM supportant les processus légers et la synchronisation, le modèle de donnée le supporte donc aussi. La méta-donnée *classStruct* de KVM est ainsi également une instance de la classe *java.lang.Class* synchronisable, ce qui alourdit le modèle.

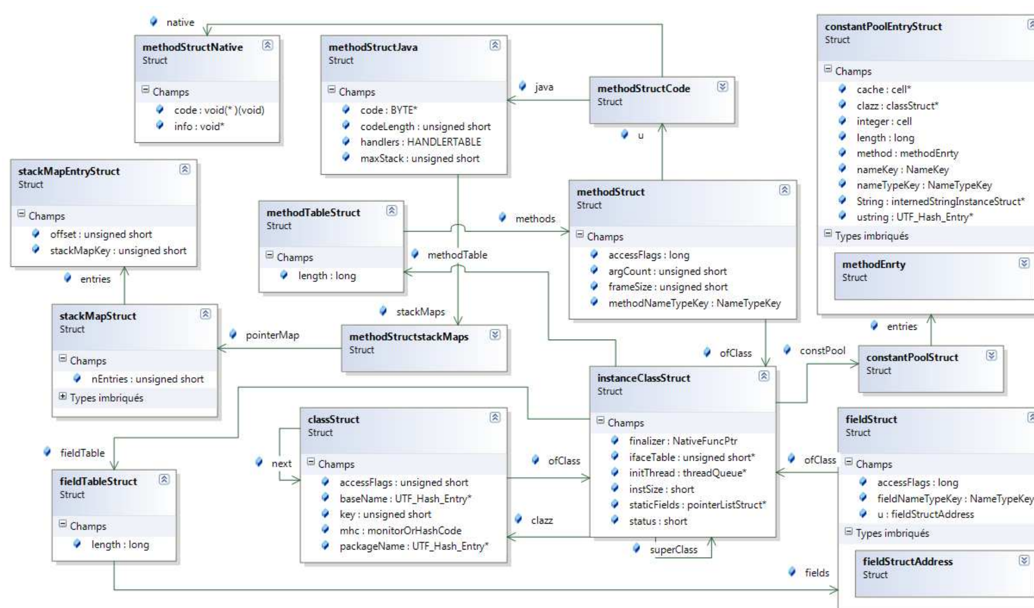


Figure 5.3: Modèle des méta-données de KVM

5.3.1 Motifs d'accès aux méta-données

Parmi les quelques 200 *bytecodes* définis dans le jeu d'instructions du Java, peu d'entre eux finalement utilisent des méta-données. On compte parmi eux quatre types d'appels de méthodes[5], quatre accesseurs aux champs statiques et d'instances[6], trois *bytecodes* de lecture de constante dans le *ConstantPool*[7], le *bytecode* de création d'*objets*[8] et celui de tableaux d'*objets*, et deux *bytecodes* de contrôle dynamique de typage[9]. Les instructions de retour de méthode n'ont pas d'opérandes mais font également appel indirectement à des méta-données pour rétablir le contexte de la fonction appelante.

Les différents types de méta-données que ces *bytecodes* utilisent sont résumés dans le tableau 5.1.

[4] Y compris les champs hérités

[5] INVOKEVIRTUAL, INVOKESPECIAL, INVOKESTATIC, INVOKEINTERFACE

[6] GETSTATIC, PUTSTATIC, GETFIELD, PUTFIELD

[7] LDC, LDC_W, LDC2_W

[8] NEW, ANEWARRAY

[9] CHECKCAST, INSTANCEOF

	Classe	Methode	Champs	*ConstantPool*
INVOKEXXX	X	X		X
GET/PUTXXX	X		X	X
LDCXX				X
RETURN	X	X		
NEW, LDC,...	X			X

Tableau 5.1: Type de méta-données par type de *bytecodes*.

5.3.2 Cas d'étude : INVOKEVIRTUAL

Le meilleur exemple de motifs d'accès aux méta-données Java dans KVM mais aussi un des plus complexes est la résolution de méthodes virtuelles. Les motifs d'accès dessinés par les autres *bytecodes* cités précédemment sont très ressemblants et reprennent au moins la partie résolution de nom.

La résolution d'une méthode virtuelle intervient lorsque qu'une méthode M_1 d'une classe C_1 en vient à appeler une méthode M_{virt} en exécutant un *bytecode* INVOKEVIRTUAL, comme dans le code d'exemple Listing 5.1. Ce dernier est écrit en Java, tandis que le Listing 5.2 est la version compilée de la même méthode.

Dans cet exemple, la méthode virtuelle *toString()* est appelée sur un objet de type déclaré *java.lang.Object* passé en paramètre à la méthode *demo*. À l'exécution, le type concret de *obj* peut être *a priori* de n'importe quel type chargé dans la JVM car nous ne connaissons pas ici les cas d'usage de *demo*. La probabilité pour que la bonne méthode *toString()* ne soit pas celle définie dans la classe *java.lang.Object* est donc très élevée. C'est à ce moment qu'intervient la résolution de méthode virtuelle, qui est un processus obligatoirement dynamique car le type concret de *obj* est inconnu à la compilation.

Listing 5.1: Exemple de méthode virtuelle sans type concret

```
public String demo(Object obj){
   return obj.toString();
}
```

Listing 5.2: Version compilée

```
public java.lang.String demo(java.lang.Object);
  Code:
   Stack=1, Locals=2, Args_size=2
   0: aload_1
   1: invokevirtual  #26; //Method java/lang/Object.toString:()
       Ljava/lang/String;
   4: areturn
```

Le processus complet de résolution de méthode virtuelle est schématisé dans le diagramme de séquence figure 5.4, page 83, et se déroule comme suit :

Phase 1 - Résolution de la méthode de référence :

1. La première étape consiste à d'abord résoudre la méthode de référence M_{ref} indépendamment du type de *obj*. À partir de l'entrée n°26 dans le *ConstantPool* de C_1, l'aborescence des symboles mène à découvrir que M_{ref} appartient à la classe

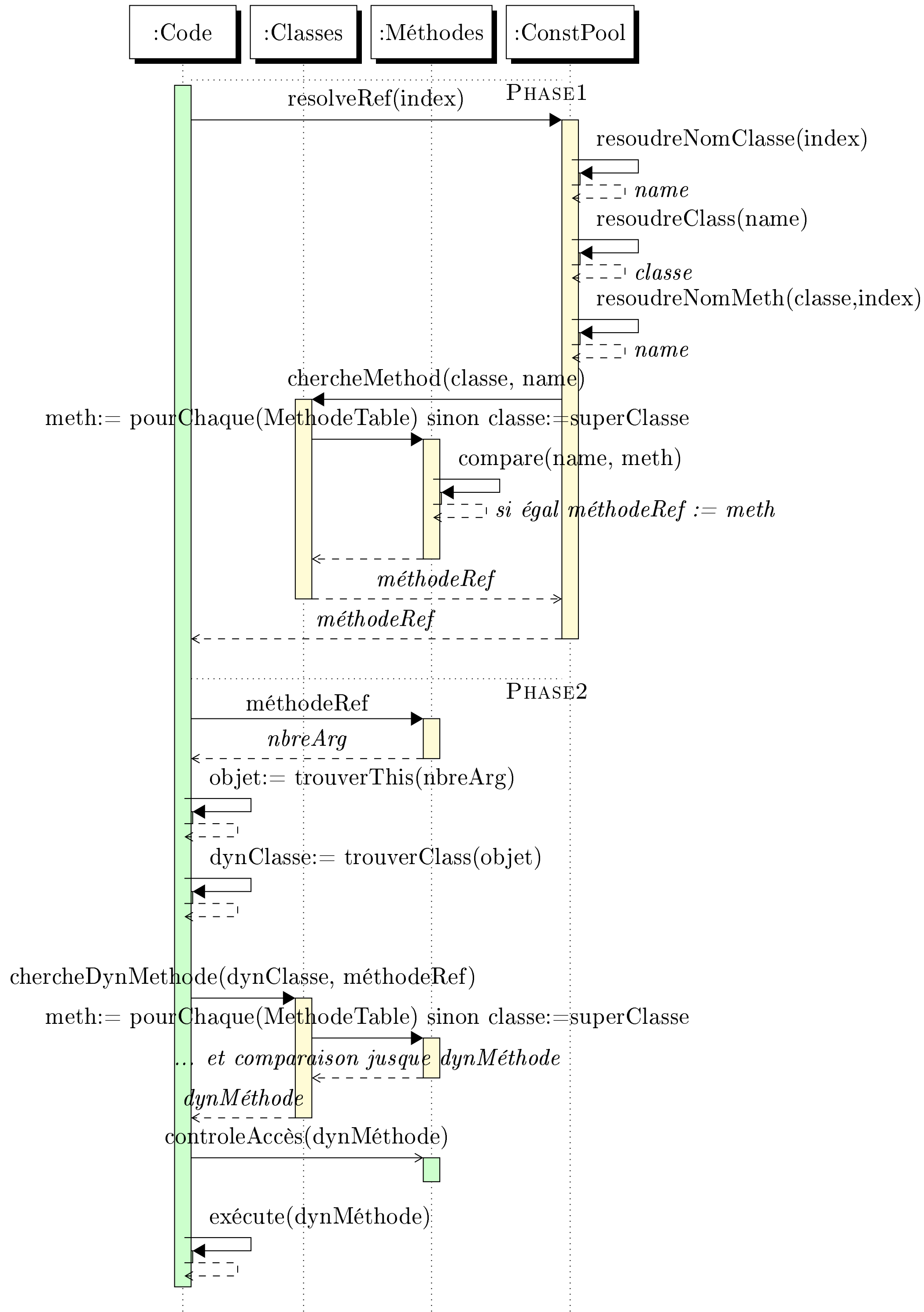


Figure 5.4: Diagramme de séquence de résolution de méthodes virtuelles.

java.lang.Object (C_{obj}) et que son nom est *toString* et *()Ljava/lang/String;* sa signature.

2. Grâce à ce nom, le processus cherche alors la définition de cette méthode dans la table des méthodes de C_{obj}. Si C_{obj} ne possède pas de méthodes de ce nom, la recherche continue dans la classe parent de C_{obj} - si elle existe. À la fin de cette étape, et en cas de succès, la première phase du diagramme se termine avec la connaissance de l'adresse de la structure de méta-données de type *methodStruct* correspondant à M_{ref}

Phase 2 - Résolution de la méthode virtuelle :

1. Rechercher M_{ref} est l'unique moyen de retrouver dynamiquement la position de *obj* dans la pile Java, en récupérant le nombre d'arguments de M_{ref}. Le but est de pouvoir déterminer le type concret de *obj* (*i.e. this*) qu'est la classe C_{dyn}.
2. En utilisant C_{dyn} et sa table de méthodes, la résolution se termine par la découverte de M_{virt}, l'implémentation de *toString* surchargée dans le source d'origine de C_{dyn}.java.
3. Après une dernière étape de contrôle d'accès (*i.e.* vérifier que M_{virt} est accessible en terme de droits depuis M_1), l'exécution de M_{virt} peut alors commencer.

5.3.3 Spécificités de KVM

À ces *bytecodes* standards s'ajoutent d'autres instructions propres à KVM, inspirés de ceux décrits en tant que pistes d'optimisation dans le chapitre 9 de [Lindholm 1999]. Ces *bytecodes* sont liés à l'édition des liens, et notamment l'étape de résolution de noms symboliques vers des valeurs concrètes. Par exemple, dans le fichier de classe d'origine, l'opérande du *bytecode* NEW est un index dans le *ConstantPool* pointant vers le nom de la classe à instancier. Une fois trouvé le nom, puis l'adresse mémoire de la structure contenant la classe correspondante, le *bytecode* NEW et son opérande sont remplacés directement dans le code de la méthode en cours d'exécution par un nouveau *bytecode* NEW_FAST et une nouvelle opérande permettant d'accéder directement à la classe sans passer par son nom.

KVM étant une JVM 32 bits et l'opérande de NEW un entier 16 bits, il n'est pas possible de stocker dans le code le pointeur 32 bits vers la stucture *classStruct*. Pour contourner le problème KVM utilise un cache monomorphique en ligne [Hölzle 1991]. L'opérande devient alors un index dans ce cache dont l'entrée pointe alors vers la structure *classStruct* correspondante à celle fraîchement résolue, et en mémorisant l'ancien contenu.

Cette technique est appliquée aux autres *bytecodes* consommant des méta-données mais n'est pas toujours possible. D'abord parce que le cache a une taille finie et le remplacement d'une entrée entraîne un retour arrière d'un *bytecode* « résolu » vers sa version « non-résolue » , avec potentiellement un recommencement de la procédure si il est à nouveau exécuté. La deuxième impossibilité porte sur le cas des méthodes virtuelles et d'interfaces. Le cache en ligne de KVM mémorise la dernière résolution de méthode virtuelle, comme vue précédemment, à un endroit précis du code, et surtout pour un type donné. Si à la prochaine exécution de ce *bytecode*, le type concret de l'*objet* est différent, alors la recherche itérative par le nom recommence à travers les tables de méthodes virtuelles comme dans décrit la phase 1 du diagramme 5.4.

La principale optimisation de la Romisation KVM consiste ainsi à identifier un maximum de mutations pouvant être définitives, comme les créations d'objets, les appels aux méthodes statiques, les constructeurs ou certaines méthodes virtuelles M_{dyn} qui ne seront jamais surchargées d'un fait du caractère clos de la Rom, etc. Notre évaluation des accès

aux méta-données dans KVM se basera donc implicitement sur ces mutations allégeant considérablement les accès aux méta-données du *ConstantPool* (et donc les chaînes de caractères).

5.4 Modèle JavaCard

5.4.1 Un modèle condensé

Le modèle de méta-données JavaCard (figure 5.5, page 86), qui est aussi celui du fichier CAP, est beaucoup plus complexe que celui de KVM. Le modèle JavaCard est un emboîtement de petites structures basées sur des entiers d'au maximum 16bits. Ce modèle est découpé en 12 composants qui regroupent les méta-données par catégories d'usage, plus que par types. On y trouve des classiques comme par exemple le composant *Class*, *Method*,*ConstantPool*, etc (La liste complète est disponible dans [JCVM.2.2.1 2003]). Toutefois, toutes les informations constituant habituellement les méta-données Classe ou Méthode ne sont pas stockées dans un composant unique. Une partie est par exemple déportée dans un autre composant appelé *Descriptor*. Ce composant contient des méta-données descriptives par opposition aux méta-données exécutives, celles utilisées dans le flot d'exécution des *bytecodes* listées section 5.3.1, page 81, et contenues dans le deux premiers composants.

La grosse différence avec le modèle standard tient dans l'absence des chaînes de caractères dans les spécifications Java pour JavaCard. Ceci a pour conséquence de les exclure également des méta-données. La recherche symbolique d'une classe ou d'une méthode ne peut donc se faire sur son nom. Au lieu de cela, l'étape de conversion affecte aux classes, interfaces, méthodes et champs, un jeton sous forme d'un entier 16 bits qui sert à les identifier dans le modèle de données. Chaque jeton est unique à l'intérieur d'un paquetage JavaCard (*i.e* un ficher CAP) et réduit donc considérablement l'empreinte nécessaire pour lier les méta-données entre elles dans le modèle.

Néanmoins, le modèle JavaCard utilise toujours un *ConstantPool*, même si celui-ci est compacté à la conversion sur le modèle de la *Romisation*. Ce *ConstantPool* entretient un minimum de symbolisme pour les *bytecodes* utilisant des méta-données. Ce qui permet de lier les fichiers de CAP entre eux, lors de leur installation sur la carte à puce.

5.4.2 Motifs d'accès revisités

Du fait de modèles différents, les chemins d'accès aux méta-données ne sont pas exactement les mêmes entre JavaCard et KVM. Nous reprenons notre exemple le plus complexe, à savoir la résolution de méthodes virtuelles, pour mettre en évidence les similitudes et différences entre ces deux modèles.

Le diagramme 5.4, page 83 peignant la résolution de méthodes virtuelles reste valable dans ces grandes lignes pour JavaCard. Il y a toujours deux phases ; une résolution de liens dans le *ConstantPool*, et une recherche de la bonne méthode dans la hiérarchie des classes. Toutefois, chaque étape est grandement simplifié en terme de données accédées. JavaCard distingue ainsi méthodes virtuelles internes au CAP de celles externes. Une classe JavaCard possède donc deux tables de méthodes virtuelles. Le diagramme 5.4 est remplacé pour JavaCard par le Listing 5.3, page 87, en pseudo-code, pour plus de lisibilité de cette distinction.

Dans chaque table d'une classe, un index est strictement égal à un jeton de méthode, et ce, relativement à la première méthode virtuelle définie dans la classe, appelée base. L'entrée à cet index contient l'offset de cette méthode dans le composant *Method*. De plus, la hiérarchie des index des tables suit la hiérarchie des classes. Ainsi, le premier index dans

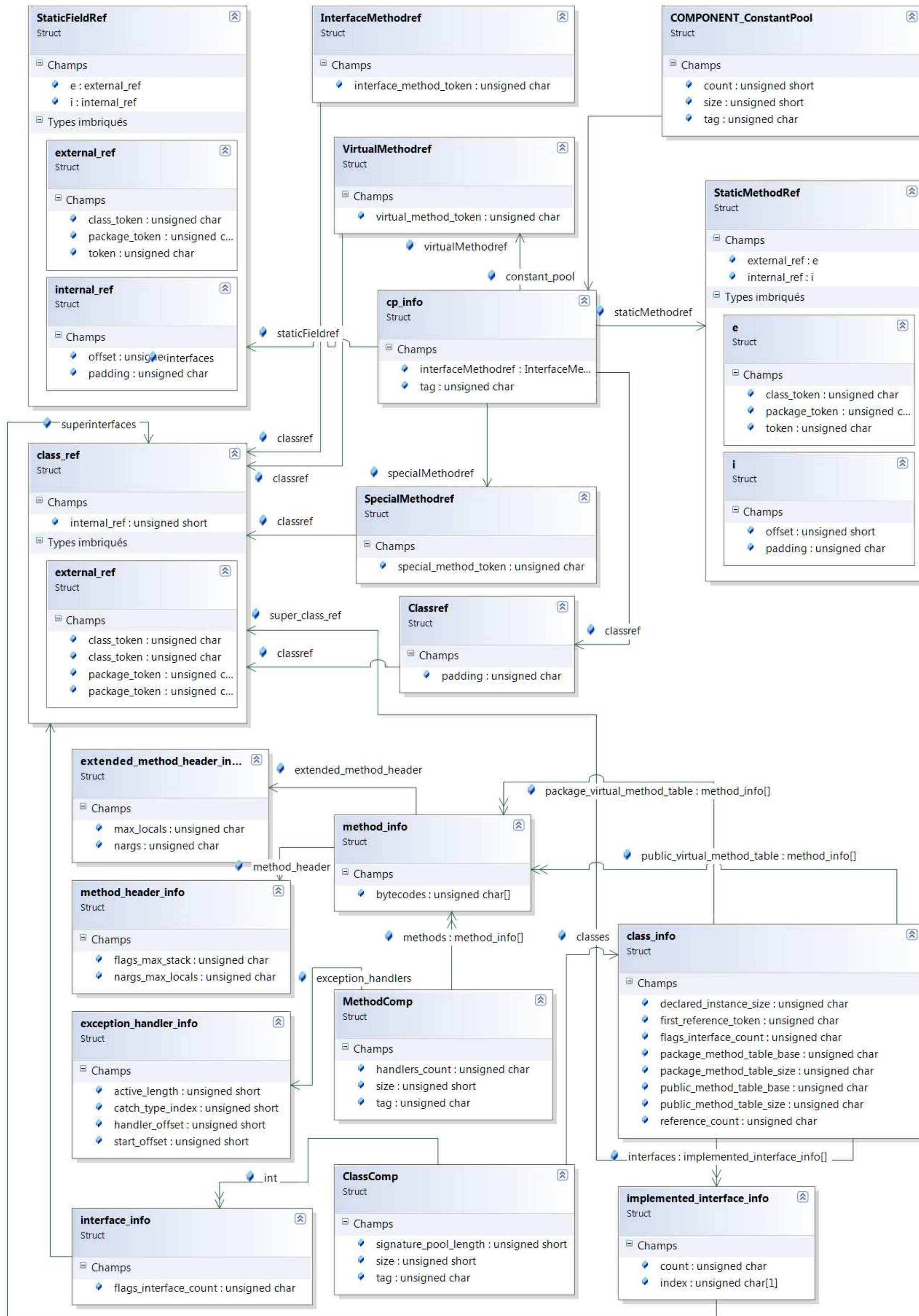


Figure 5.5: Modèle des méta-données de JavaCard (Classes, Méthodes et *ConstantPool*).

Listing 5.3: Résolution de méthode virtuelle JavaCard

```
entrée : bytecode= INVOKEVIRTUAL , operande = XX

//PHASE 1
methodToken := pool[operande].virtual_method_ref.virt_method_token
classToken  := pool[operande].virtual_method_ref.class
si classToken estInterneAuCAP alors
   classInfo  := class_component.class_info[refClass.internal_ref]
sinon
   capToken   := refClass.external_ref.package_token
   classToken := refClass.external_ref.class_token
   classInfo  := chercheClassExportée(capToken,classToken)
fin si

si methodToken & 0x80 alors
   base = classInfo.package_methode_table_base
   tant que methodToken < base
      classInfo= classInfo.superclass
      base = classInfo.package_methode_table_base
   fin tant que
   methodInfo := classInfo.package_methode_table[methodToken-base]
sinon
   base = classInfo.public_methode_table_base
   tant que methodToken < base
      classInfo= classInfo.superclass
      base = classInfo.public_methode_table_base
   fin tant que
   methodInfo := classInfo.public_methode_table[methodToken-base]
fin si

//PHASE 2
this := sommetPile - methodInfo.argcount
dynClass := this.class

tant que dynClass != java.lang.Object
   si methodToken & 0x80 alors
      index := methodToken - dynClass.package_methode_table_base
      si index < dynClass.package_methode_table_size alors
         dynMethod= dynClass.package_methode_table[index]
      fin si
   sinon
      index := methodToken - dynClass.public_methode_table_base
      si index < dynClass.public_methode_table_size alors
         dynMethod= dynClass.public_methode_table[index]
      fin si
   fin si
   dynClass := dynClass.superclass
fin tant que

retourner dynMethod
```

la table d'une classe fille suit le dernier index de la table de la classe mère (Phase 2 du Listing 5.3), grâce à un calcul simple sur la « base ». Ce qui permet de remonter quasiment directement dans la hiérarchie des classes parentes sans recherche itérative par classes. Par rapport au modèle de KVM, l'appel d'une méthode virtuelle est donc fortement simplifiée à ce niveau.

JavaCard permet également de simplifier encore cette étape grâce à un composant spécial

appelé *ReferenceLocation*. Ce composant liste tous les offsets dans le composant *Method* où se situe une référence dans le *ConstantPool*. Grâce à ce composant, il est alors facile de résoudre partiellement presque toutes les références symboliques restantes à l'installation. Cette résolution « partielle » est la suivante.

La Phase 1 sert uniquement à trouver le nombre d'argument de la méthode à appeler pour localiser l'*objet this* sur la pile. Quelque soit sa localisation dans la hiérarchie des classes, cette méthode virtuelle aura toujours le même nombre d'argument. Grâce au composant *ReferenceLocation*, il est alors facile de stocker une information permettant de retrouver rapidement ce nombre d'argument. Avec cette résolution précoce, qui peut être réalisée à l'installation, la Phase 1 peut alors être évitée dans la plupart des cas.

5.5 Vers la mise-en-cache des méta-données

5.5.1 Préparation des expérimentations

5.5.1.1 Modification de KVM

La *Romisation* dans KVM est à l'origine conçue pour être stockée dans une mémoire adressable et les liens entre structures de données à l'intérieur de la Rom se font donc par pointeurs. Cela ne fonctionne pas si cette Rom est stockée dans une mémoire non-adressable. Nous avons donc dû modifier l'outil de *Romisation* et la JVM en conséquence.

En premier lieu, pour rendre la Rom indépendante de son adresse de base dans la Flash, qui est inconnue à la romisation, nous lions les structures de données de la Rom désormais par offsets relatifs à l'intérieur de celle-ci. Nous avons également ajouté une entête à cette Rom donnant les offsets de la table de classes et de la table des chaines de caractères pour que la JVM puisse s'initialiser en accédant facilement ses classes de base (Object, System, String, Thread et Class). Ces tables de hachage sont reconstituées au démarrage de KVM mais les données pointées restent bien évidemment dans la Rom. Enfin, nous avons modifié le source de la JVM pour remplacer tous les accès par pointeurs aux méta-données par des adresses virtuelles en Flash série et en enlevant dans le code toutes notions de structures C. Au final, cette Rom peut être stockée dans un fichier indépendant du binaire de KVM pour réaliser des tests à l'identique d'une mémoire Flash série.

Le placement des données au sein de la Rom reste identique à la version d'origine. L'approche suivie par KVM est de regrouper les structures de méta-données par types plutôt que par classes. Cette Rom est donc constituée de plusieurs plages distinctes et clairement délimitées. Par exemple, les structures de méta-données *methodStruct* et leurs *bytecodes* correspondants ne sont ainsi pas mélangés et très éloignés.

5.5.1.2 Modification de la JCVM

Pour ce qui concerne JavaCard, nous avons utilisé une implémentation de JCVM propriétaire car aucune implémentation publique n'est disponible. Toutefois, un gros avantage est d'avoir pu étudier une JCVM utilisée à grande échelle, fiable et éprouvée.

Les modifications qui lui ont été apportées reprennent la même logique que pour KVM en cloisonnant et instrumentant les accès aux méta-données et en enlevant toutes notions de structures C. À une différence près, propre à la sécurité des plate-forme JavaCard : pour des raisons évidentes de protection de code critiques, certaines méta-données et certains bouts de code doivent être placés dans une mémoire non-inscriptible. Puisque la cible doit donc pouvoir indifféremment accéder à des données dans deux mémoires différentes, l'instrumentation en tient compte elle-aussi.

À la différence de la Rom KVM, le placement au sein de la mémoire se fait par fichiers CAP. L'organisation de leur contenu n'est pas modifiée. Ainsi, par exemple, la structure de méta-données *method_info* se trouve donc directement placée à côté de son tableau de *bytecodes*. Ce qui constitue une autre différence fondamentale entre la JCVM JavaCard et KVM.

5.5.2 Répartition des méta-données

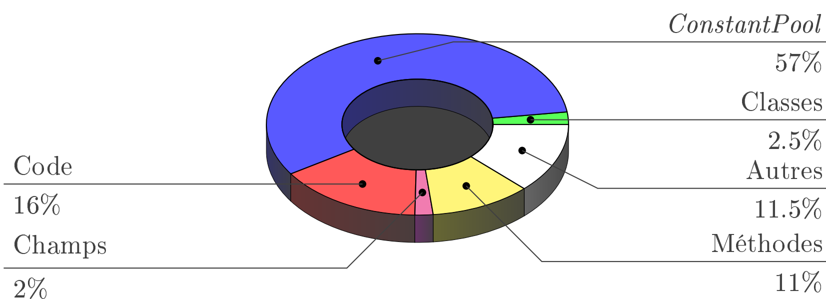


Figure 5.6: Proportion des méta-données J2ME.

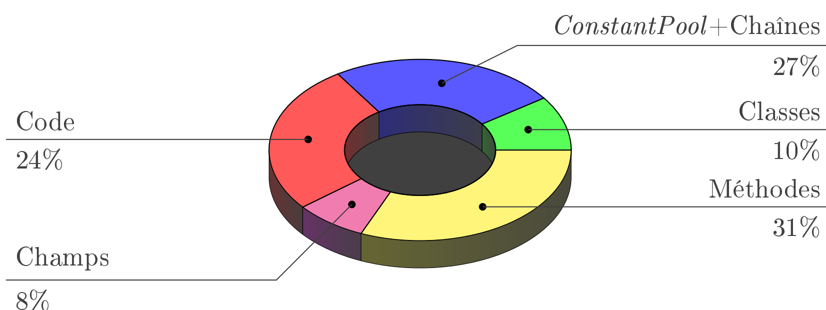


Figure 5.7: Proportion des méta-données J2ME romisées pour KVM.

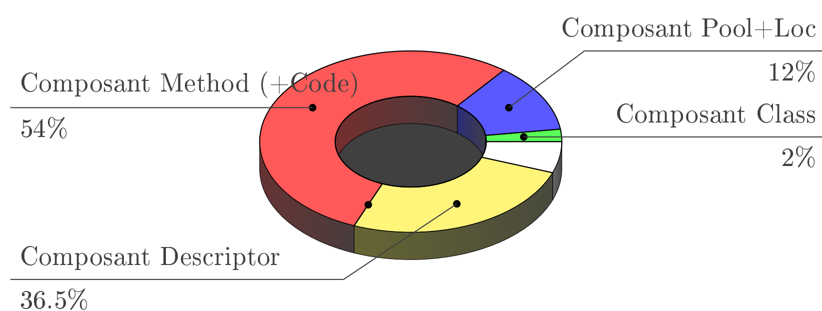


Figure 5.8: Proportion des méta-données de l'API JavaCard
et de nos programmes de tests.

5.5.2.1 Répartition des méta-données brutes

Le camembert figure 5.6 fournit la proportion de chaque type de méta-données comptabilisées dans l'ensemble des fichiers de classes formant la bibliothèque J2ME ainsi que

l'ensemble des applications de tests qui sont utilisées dans la suite de ce chapitre. Le *ConstantPool*, en bleu, forme de loin le contingent le plus important avec plus des deux tiers du volume total. Le code, en rouge, ne représente au final que 16 % du volume total de tous les fichiers de classes. Alors que dans un binaire natif, le code représente la quasi-totalité du volume (hors débogage si inclus). Les principaux types de méta-données que sont les classes, les champs et les méthodes représentent un peu moins de 15 %. Le segment blanc regroupe des méta-données descriptives qui n'interviennent pas ou indirectement lors de l'exécution mais qui servent au chargement de la classe comme la table des variables locales de chaque méthode, des informations sur l'utilisation de la pile Java, etc[10].

5.5.2.2 Répartition des méta-données dans KVM

La figure 5.7 fournit la proportion de chaque type de méta-données dans une *Rom* KVM. Pour notre évaluation, cette *Rom* contient la bibliothèque J2ME ainsi que l'ensemble des applications de tests qui sont utilisées dans ce chapitre.

La première remarque concerne la part du *ConstantPool* et son compactage discuté plus tôt. On remarque que celui-ci se réduit de manière spectaculaire par rapport aux données brutes, montrant tout l'intérêt d'une telle approche pour les systèmes embarqués contraints en espace mémoire. La deuxième remarque porte sur la part prise par le type *Méthode* qui dépasse le volume du *Code*. Cette proportion est une donnée fort intéressante, car en regardant de plus près, il s'avère que 1341 méthodes, soit 75 % de celles présentes dans la *Rom*, ont moins de 32 octets de code, c'est à dire moins que la taille de la structure *methodStruct* et ses types imbriqués par des unions, vue dans le modèle 5.3, page 81. Cela signifie qu'appeler une méthode dans KVM va potentiellement provoquer plus de lectures de méta-données que lire le code lui-même, et ce, sans compter bien-sûr les autres méta-données utilisées par ce code.

Dernier constat enfin, de manière générale, la *Romisation* réduit l'empreinte mémoire des données Java (code + méta-données), de 32 % par rapport aux fichiers de classes. Mais si l'on fait abstraction du *ConstantPool*, le volume des méta-données de la *Rom* augmente de 20 % par rapport aux mêmes fichiers de classes. En considérant que ce *ConstantPool* peut être écarté par différentes techniques connues (voir [Lindholm 1999] chap. 9), on constate donc au final une inflation notable entre le volume de méta-données brutes et les méta-données KVM, c'est-à-dire au cours de la transformation du MMD.

5.5.2.3 Répartition des méta-données dans JavaCard

La répartition des méta-données par type dans un fichier CAP est plus difficile à comparer avec le modèle classique ou même celui de KVM du fait de l'existence de méta-données très spécifiques à la plateforme JavaCard. De plus certaines informations sur les classes ou les méthodes sont réparties sur plusieurs composants. Par exemple, l'information d'appartenance d'une méthode à une classe ne se trouve ni dans le composant *Method* ni le composant *Class* mais dans le composant *Descriptor*. Le *ConstantPool* est lui aussi, par exemple, livré sous deux formes, le composant du même nom, et une indexation dans le composant *ReferenceLocation*, vu précédemment.

Le composant *Method* contient des informations basiques sur les méthodes du CAP, uniquement de quoi les placer sur la pile. Il inclue surtout le code de toutes ces méthodes, qui représente souvent plus de 80 % du composant. Le composant *Class*, quant à lui, contient les tables de méthodes virtuelles qui, dans le modèle classique, ainsi que celui de

[10] La liste complète se trouve dans [Lindholm 1999], Chap 4.7

KVM, sont des méta-données dissociées de la classe et des méthodes, par être des méta-données à part entière (*i.e. methodTableStruct*).

La figure 5.8 montre la répartition en volume des principaux composants d'un fichier CAP. Ces chiffres sont issues d'une moyenne calculée sur l'API JavaCard et un panel de 38 fichiers CAP d'applications utilisées dans de vraies cartes JavaCard du marché. On constate clairement une position dominante en volume, du code par rapport aux méta-données. Ce qui marque à nouveau une autre différence nette entre le MMD JavaCard et celui de KVM.

	Richards Version 1 Java	Richards Version 7 Java
Taille Cap en octets	2205	4854
Dont volume Code en octets	1002	1728
Taille .class en octets	7368	16053

Tableau 5.2

Enfin, un dernier constat, un fichier CAP ayant en moyenne une taille de 4,8 Ko, il ne représente alors plus que 30 % du volume des fichiers de classe correspondant (exemples dans le tableau 5.2), soit une compression extrêmement performante et bien plus poussée que KVM.

5.5.3 Utilisation des méta-données

La suite Richards permet d'étudier finement les méta-données puisqu'elle va en manipuler à différents degré selon les fonctionnalités *objet* utilisées. Par exemple, remplacer des accès directs à des champs d'objets par des méthodes change le type de méta-données utilisées pour réaliser la même opération de flot d'informations. Cependant, ces modifications entraînent dans le même temps une augmentation du nombre de *bytecodes* exécutés. Dans l'exemple simple précédent, ce nombre passe ainsi de 1 à 3, dans le cas le plus simple d'une lecture de champs (*getter*). Dans le passage de la version 1 à la version 7 de la suite Richards, le nombre de *bytecodes* exécutés augmente très fortement (+ 112 %) et c'est en toute logique que la version 7 sera la plus lente, bien que produisant le même résultat applicatif.

Cependant, cette augmentation du nombre de *bytecodes* n'est pas le seul facteur de ralentissement au gré des différentes versions, et n'est surtout pas le facteur plus prononcé. En effet, le volume de méta-données utilisées augmente par exemple de son coté de + 807 % dans KVM entre la version 1 et la version 7.

La suite Richards est présentée comme un ajout progressif de fonctionnalités *objet* à un même programme Java. Or, cette vision n'est pas juste car il est indéniable que les librairies, *framework* et applications Java sont quasiment toutes développées dans le style de la version 7 ou au pire dans le style de la version 5. La suite Richards est donc plutôt une dégradation progressive, de la version 7 à 1, des fonctionnalités *objet*. Dégradation qui laisserait donc à penser que pour améliorer les performances globales d'une application Java, l'idéal est de supprimer les avantages et bénéfices des langages orientés *objet* en terme de conception d'applications. Or, ce discours n'est évidemment pas tenable. Néanmoins, la suite Richards met en évidence par ce biais, la capacité ou l'incapacité d'un MDD d'une JVM donnée à subir la montée en charge de la programmation orientée *objet* conventionnelle et actuelle.

Pour que l'impact du nombre de *bytecodes* ne viennent pas parasiter l'analyse de méta-données, il faut également revenir à des nombres de *bytecodes* exécutés comparables entre toutes les versions de la suite. Toutes nos analyses comparant des programmes entre eux se baseront donc sur leur premier million de *bytecodes* exécutés. Ce critère étant suffisamment

long pour laisser chaque programme développer son comportement particulier face à la consommation de méta-données.

	Richards					
	V1	V2	V3	V4	V5	V7
KVM	1,95	1,25	2,94	6,15	17,68	17,71
JavaCard	0,20	0,22	0,30	0,62	1,24	1,26

Tableau 5.3: Nombre moyen de méta-données par *bytecode* dans KVM et JavaCard.

Le tableau 5.3 livre nos résultats sur les mesures du volume de méta-données consommées par les programmes de tests, indépendamment donc, du nombre de *bytecodes* exécutés. La première ligne donne le nombre moyen de méta-données lues pour un *bytecode* exécuté, et mesuré dans KVM. La seconde ligne rapporte le même exercice mais dans notre JCVM JavaCard.

Dans ce tableau, un nombre moyen supérieur à 1 signifie que la JVM consomme plus de méta-données que de code. En ne regardant ainsi que la version 7, on constate de manière claire que la programmation orientée *objet* implique dans nos deux MMD étudiés une consommation de méta-données plus importante que de *bytecodes*.

Pour KVM, l'écart de volume est très important, soit les déjà-cités + 807 %. Cet écart indique que KVM de part son MMD subit la programmation orientée *objet* de manière assez violente. Ceci est d'ailleurs exacerbé par le fait que le MMD JavaCard n'implique pas du tout un tel volume de méta-données pour exécuter exactement le même code Java. La JCVM JavaCard utilise 14 fois moins de méta-données que KVM.

Les chemins d'accès qui mènent d'un *bytecode* à une méta-données sont donc beaucoup plus longs dans le MMD KVM que dans le MMD JavaCard.

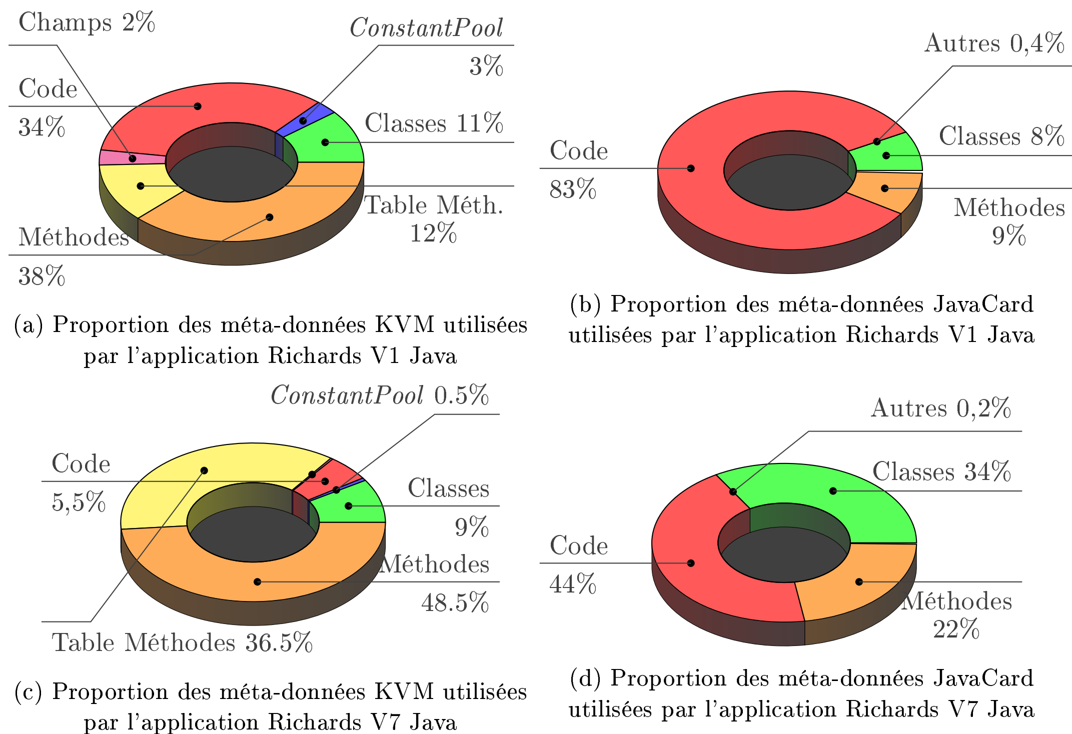
Les figures 5.9, page 93, montrent la proportion des types de chemins, et donc des types de méta-données, qui influencent ces différents écarts de volume. Ils comparent toujours les versions extrêmes 1 et 7 (en ligne), et KVM et JavaCard (en colonne). Bien que comme nous l'avons dit, la version 1 n'a pas beaucoup de sens, il reste néanmoins intéressante de s'attarder sur ces graphiques pour voir et comprendre pourquoi la version 7 JavaCard est également moins consommatrice de méta-données que la version 1 KVM, comme noté dans le tableau 5.3.

Pour KVM, le chemin critique se situe dans l'accès aux structures de méta-données *MethodStruct et MethodTableStruct* (voir MMD de KVM Fig. 5.3, page 81). C'est-à-dire aux tables de méthodes virtuelles. Dans le MMD JavaCard, ces tables constituent le même chemin critique[11]. Toutefois, l'impact est moindre que dans le MMD de KVM. La gestion hiérarchique des méthodes virtuelles du MMD JavaCard standard est donc globalement plus efficace que la gestion à plat par niveaux et par tableaux de KVM, issue d'une lecture « naïve » des descriptions provenant des spécifications.

Synthèse Il faut retenir trois points essentiels de cette première partie de l'analyse. Ces premiers résultats montrent d'abord à quel point un MMD impacte directement les performances. Ensuite, nous avons également identifié que les points critiques des MMD étaient les structures définissant les méthodes et tables de méthodes. Enfin, nous avons mis en évidence qu'une JVM comme KVM peut même consommer plus d'octets de méta-données que d'octets de code.

Nous continuons maintenant notre comparaison en nous orientant désormais vers des problématiques de mise en cache.

[11] notons que dans ce MMD, les tables de méthodes virtuelles font parties de l'entité *class_ info*



(a) Proportion des méta-données KVM utilisées par l'application Richards V1 Java

(b) Proportion des méta-données JavaCard utilisées par l'application Richards V1 Java

(c) Proportion des méta-données KVM utilisées par l'application Richards V7 Java

(d) Proportion des méta-données JavaCard utilisées par l'application Richards V7 Java

Figure 5.9: Comparaison de l'usage de méta-données dans KVM et JavaCard

5.6 Cache de méta-données logiciel

5.6.1 Cartographies

De la même manière que dans le chapitre précédent, il est également possible pour les méta-données de parler de dissémination ainsi que de points chauds, ou de dilution spatiale au sein de la *Rom* ou dans un CAP, et donc dans la mémoire non-adressable. Pour évaluer la dissémination, nous reprenons un support cartographique avec la même nomenclature que précédemment. La cartographie inclue également les *bytecodes* à titre indicatif et pour montrer une vision globale de l'usage de la *Rom* ou du CAP.

5.6.1.1 Cartographie de KVM

Comparé au code compilé, la figure 5.10 montre que la répartition des méta-données de KVM a une grande amplitude, la *Rom* étant par construction plus grosse que les binaires testés chapitre 4. Néanmoins, il apparaît toujours une nette dissémination des méta-données froides mais un regroupement plus fort des données chaudes, ici les méthodes, au moins dans des même pages de Flash (i.e. un bloc délimité en gris clair sur l'axe des ordonnées).

Un autre élément qui apparait ici est que chaque point est très petit et bien plus petit que les structures de données de la *Rom* elle-même. La consommation des méta-données se joue en fait à une granularité plus petite que la structure de méta-données. À la différence du code où le bloc de base complet est consommé, la structure de méta-données est quant à elle picorée sur quelques octets et sur quelques champs, mais jamais dans sa totalité. Une structure de méta-données est ainsi, elle aussi, composée de points chauds et de points froids. Dans ces conditions, descendre à une granularité aussi basse que 2 ou 4 octets devient

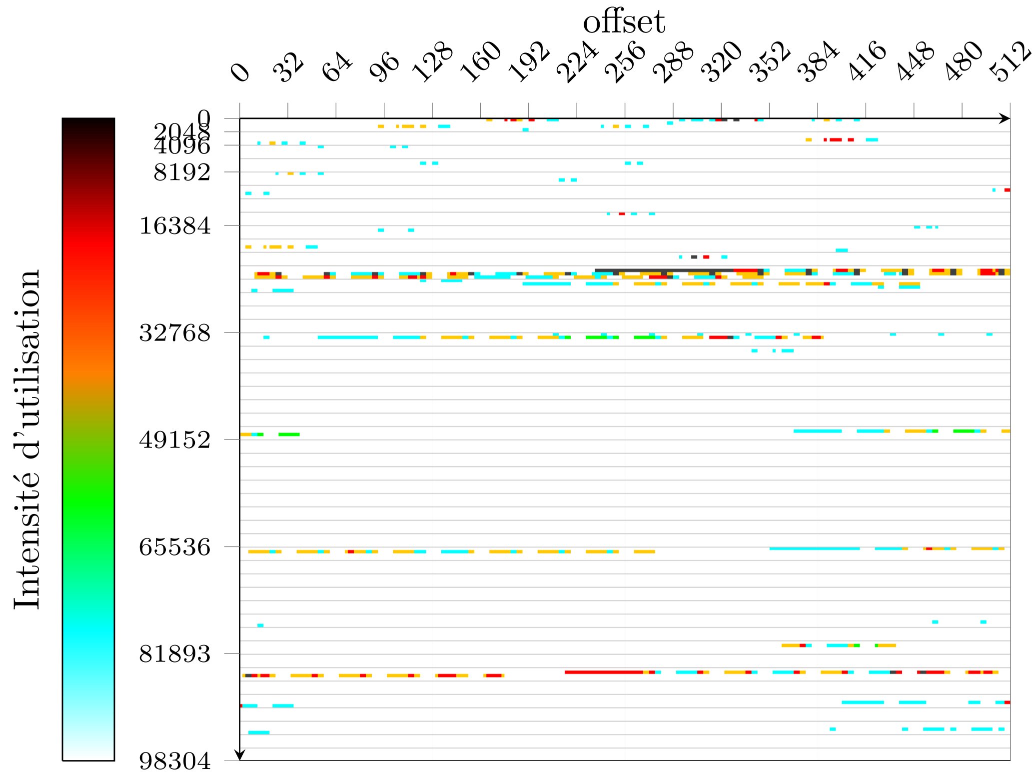


Figure 5.10: Cartographie des points chauds de la *Rom* KVM pendant le test Richards Version 7

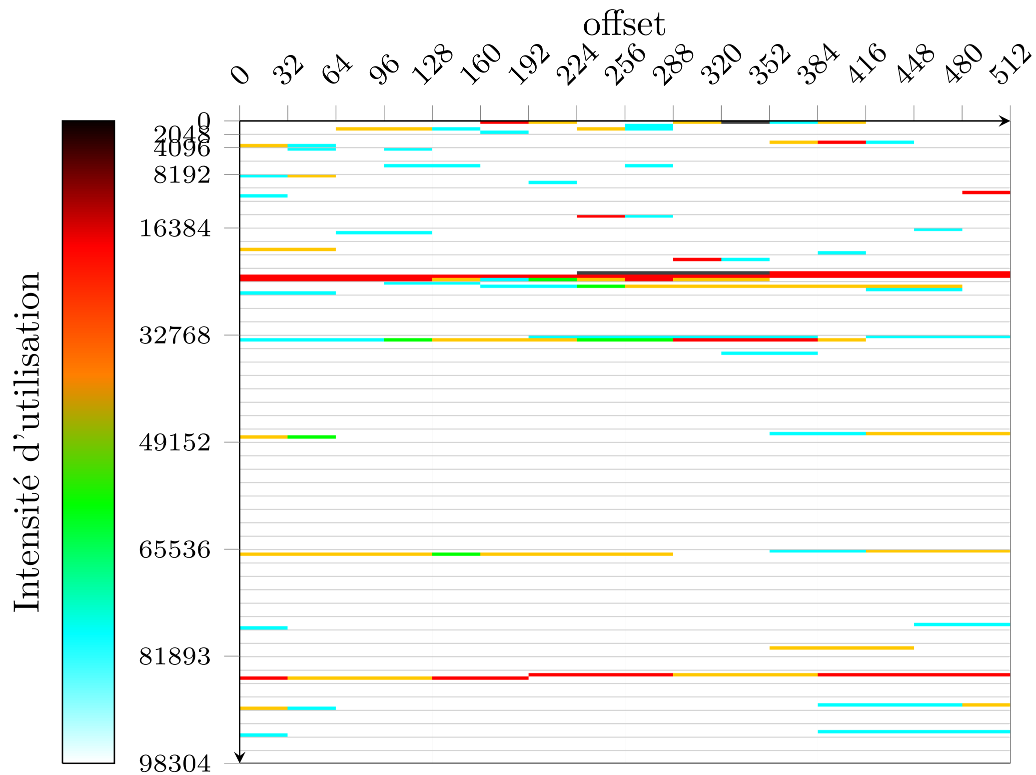
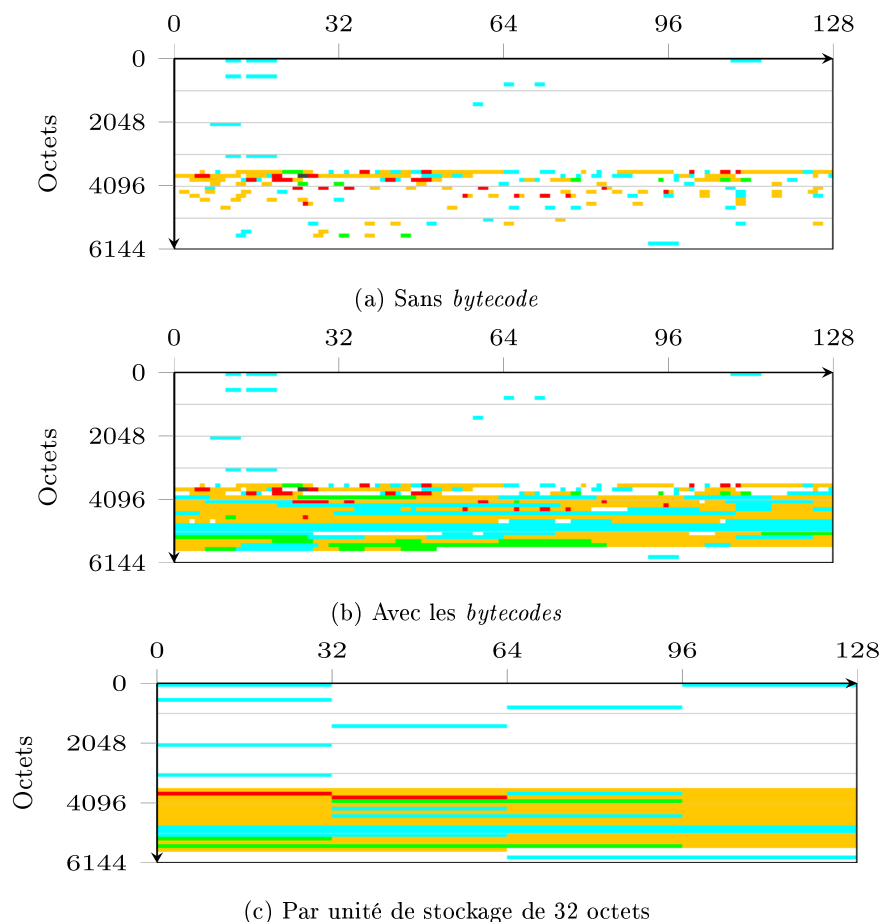


Figure 5.11: Cartographie des points chauds de la *Rom* KVM par unité de stockage de 32 octets pendant le test Richards Version 7



(a) Sans *bytecode*

(b) Avec les *bytecodes*

(c) Par unité de stockage de 32 octets

Figure 5.12: Cartographie des points chauds des CAP du test Richards V7

réellement problématique pour l'exécution en place avec ou sans cache, non pas tant à cause du volume mais aux vues du nombre de données élémentaires différentes. Nous avons en effet vu que récupérer une page complète de Flash n'était pas très profitable, mais qu'il n'était pas non plus pertinent de multiplier les accès pour récupérer seulement quelques octets.

Il y a donc là une différence majeure d'usage avec les blocs d'instructions, à moins de revoir complètement le modèle de données pour l'adapter à ce problème, ce qui n'est pas forcément trivial. D'une part, regrouper des champs pour combler les trous dans la structure laisserait au contraire des trous entre structures du même type dans la *Rom* et produirait le même résultat. A l'inverse, casser et scinder les structures par sous-fonctions changerait en profondeur la conception de la JVM, elle-même.

5.6.1.2 Cartographie de JavaCard

La stratégie de placement des données à l'intérieur d'un fichier CAP n'est pas la même que pour la *Rom* KVM. Dans cette *Rom*, les méthodes et le code sont stockés dans deux régions bien distinctes. Dans un CAP JavaCard, méthodes et codes sont dans le même composant alors que les tables de méthodes virtuelles sont dans le composant *Class*.

Les figures 5.12a et 5.12b donne une cartographie des méta-données JavaCard, respec-

tivement sans et avec les *bytecodes*. On note d'emblée que l'espace est beaucoup plus resserré que dans la *Rom* KVM, du fait à la fois de la compacité du MMD et du fichier CAP, mais aussi du regroupement des méta-données par applications plutôt que par types comme dans la *Rom* KVM.

En comparant les deux figures, on constate à nouveau que les points chauds sont essentiellement des méta-données et non du code. On constate également que ces points sont encore plus petits que ceux du MMD de KVM, mais extrêmement localisés. Les points chauds situés avant l'ordonnée 4096 sont des entrées de tables de méthodes virtuelles dans des *class_info*, les point chauds situés après cette marque sont des méta-données de type *method_info*.

5.6.1.3 Dilution spatiale

JavaCard Dans ces conditions, le dilution spatiale dans un CAP JavaCard est extrêmement faible dans l'espace des tables de méthodes virtuelles comme le montre le figure 5.12c pour une unité de stockage de 32 octets. Celle-ci est toutefois légèrement plus prononcée pour les *method_info* qui sont diluées avec le code, qui comme nous l'avons vu, est globalement moins utilisé en volume. Cependant, cette proximité réduit le nombre de blocs de cache visités lors d'un parcours à travers les méta-données comme lors d'une résolution d'une méthode virtuelle.

KVM Dans KVM, la dilution spatiale a deux incidences (figure 5.11, page 94). La première est liée à l'éclatement qui fait que la dilution spatiale augmente artificiellement la couverture de méta-données, et donc la pollution potentielle du cache. Ce qui n'est pas le cas pour JavaCard.

La seconde incidence est l'éloignement entre méta-données liées dans un même chemin d'accès et notamment le chemin critique des appels de méthodes virtuelles. Le MMD KVM et la fabrication de la *Rom* obligent plusieurs lectures à des endroits très différents, augmentant ainsi le nombre de données adjacentes moins utiles, mais lues de manière concomitante. Avec cette fois peu de chance que ces données « subies » soit plus tard utilisées.

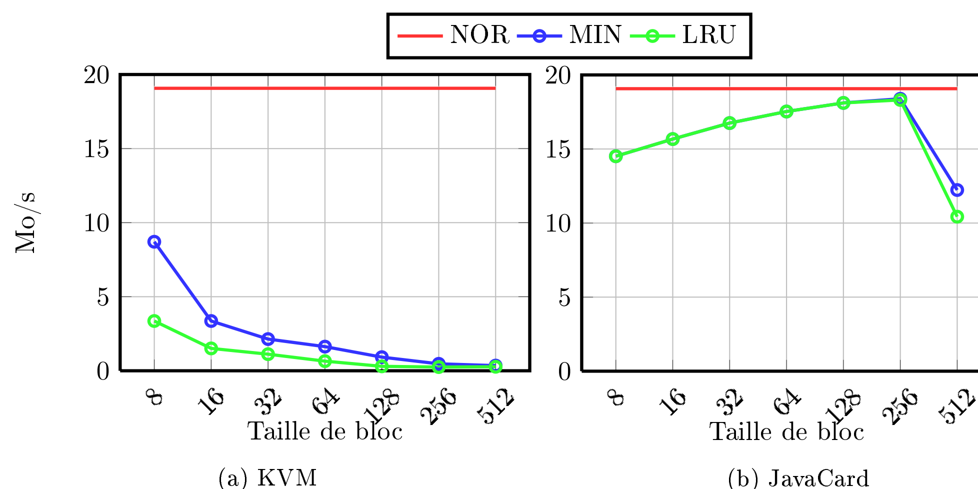
Ce phénomène est visible en fin de région « méthode » dans la figure 5.10, page 94, où tous les segments blancs entre points chauds sont devenus rouge dans la figure 5.11, alors que ces données sont de la pollution directe. Cela signifie que comme les réels points chauds de cette région sont extrêmement utilisés, alors la pollution qui les accompagne sera durablement en cache.

5.6.2 Cache de méta-données

Nous pouvons maintenant mettre en évidence, par le prisme d'un cache, les constats que nous avons établis jusqu'à présent dans notre comparaison des deux MMD, KVM et JavaCard. Pour un cache de méta-données, la problématique des stratégies de recherche développée section 4.1.4, page 51, reste la même que pour le code. Le type de contenu n'influence que le placement en cache et le déclenchement des défauts de cache. Comme dans la section **??**, page **??**, analysant les stratégies de placement et de renouvellement d'un cache de code, nous analysons donc seulement le débit du cache de méta-données en Méga-octets par seconde.

La figure 5.13, page 97, met en parallèle les résultats d'un cache de méta-données de 2048 octets sur les MMD KVM et JavaCard. Le programme de test représenté dans cette figure est la version 7 de la suite Richards. La nomenclature des deux graphiques est comparable à celle utilisée précédemment, à savoir le débit en Mo/s en ordonnée, et un changement de

la taille de l'unité de stockage en cache sur les abscisses. Cette figure compare également l'écart entre la politique de remplacement optimale MIN, en bleu, et la plus utilisée, LRU en vert. Ce à quoi s'ajoute la comparaison avec le débit de la NOR, soit l'état actuel des choses pour le stockage de méta-données dans une carte à puce. Notons enfin, que ces résultats ne portent que sur l'accès aux méta-données et n'incluent donc pas la mise en cache des *bytecodes*.

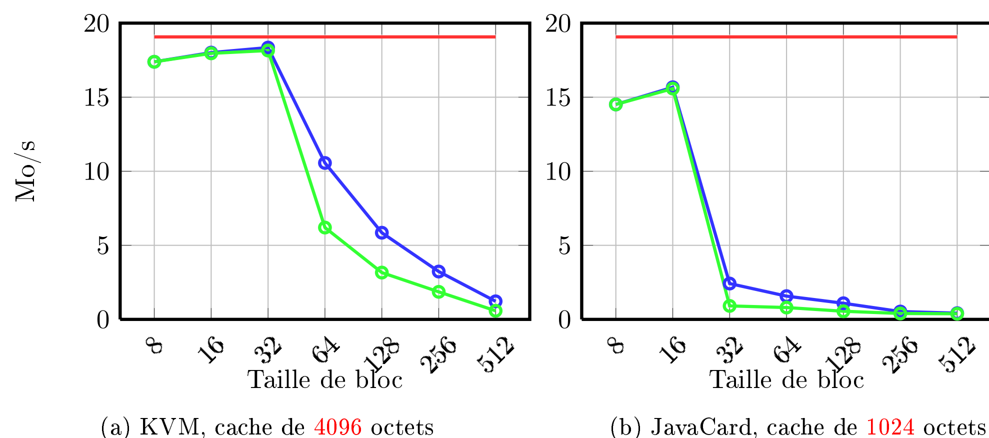


(a) KVM (b) JavaCard

Figure 5.13: Débit d'un cache de méta-données de 2048 octets en Mo/s.

Le premier constat clair est que la performance de JavaCard est bien meilleure que celle de KVM. De part son MMD compact et ses chemins critiques courts, une JCVM JavaCard passe facilement à l'échelle d'un cache de 2 Ko, ce qui n'est pas du tout le cas pour KVM. Pour approcher les résultats de la JCVM (18 Mo/s), KVM a besoin d'un cache d'au moins 4096 octets, voir figure 5.14a. Alors que le MDD JavaCard atteint encore de bon résultat (15,6 Mo/s) avec un cache de 1024 octets, voir figure 5.14b.

Le deuxième constat porte sur l'importance de la taille de l'unité de stockage, comme dans le cas d'un cache de code. Par exemple pour KVM, le cache est plus efficace avec une unité de stockage de 8 octets, car c'est à ce niveau que la dilution est la moins prononcée



(a) KVM, cache de 4096 octets (b) JavaCard, cache de 1024 octets

Figure 5.14: Débit avec d'autres tailles de cache de méta-données.

et donc la pollution la moins importante, comme nous l'avions énoncé dans la section précédente. Idem pour le MDD JavaCard, mais avec l'effet de dilution inverse. Le taux de concentration en points chauds reste élevé même avec des plus grosses unités de stockage, ce qui induit que le cache bénéficie de la récupération et du maintien en cache de points chauds, où même la présence de points tièdes, utiles à d'autres moments de l'exécution de l'application.

Dans la figure 5.14b, la meilleure unité de stockage JavaCard est plus petite car cette fois, le nombre d'entrée à disposition pour la politique de remplacement est le premier facteur qui permet d'endiguer le nombre de *Miss* plus important. Ce nombre de défaut de cache augmente en effet du fait d'un cache deux fois plus petit que dans les hypothèses de la figure 5.13b.

Notons, une nouvelle fois, que l'efficacité du cache est donc toujours l'interaction complexe de plusieurs paramètres. Pour une configuration optimale, ces paramètres doivent ainsi être choisis en groupe plutôt qu'individuellement. Prenons par exemple une préoccupation première qui serait la réduction de l'empreinte mémoire entre deux versions de développement d'une JVM. L'unité de stockage doit alors être choisie en fonction de cette empreinte et non rester celle d'un cache plus gros, même si elle était reconnue efficace avec lui.

5.7 Synthèse

L'étude sur le stockage de méta-données dans une mémoire non-adressable que nous avons menée dans ce chapitre a permis de montrer leur importance par rapport au code interprété par une machine virtuelle Java. Le volume de méta-données consommées par une JVM peut en effet être plus grand que le volume de code, lorsque les applications Java sont effectivement développées avec toute la richesse du langage.

Nous avons également mis en évidence que le modèle de méta-données jouait un rôle prépondérant dans la performance d'une JVM, puis d'un cache, et ce dès sa conception. Un modèle paraissant fiable de prime abord dans son contexte d'origine comme le modèle de KVM, s'avère en fait globalement inadapté lorsque l'utilisation d'une mémoire non-adressable couplée à un cache exacerbe ses faiblesses.

Globalement, des blocs de données de type méta-donnée sont plus petits que des blocs de données de type bloc-de-base. Ils n'ont donc pas tout a fait les même effets sur un cache logiciel. D'une certaines manière, les chemins parcourus entre méta-données ressemblent au graphe de flot de contrôle. Cependant, chaque nœud du graphe ne fait rarement plus de 4 octets. Raccourcir ces chemins et les rapprocher du code permet une meilleure localité spatiale entre code et méta-donnée, à l'image de que font les compilateurs en optimisant l'ordre des blocs de base au sein du binaire pour qu'il évite des *Miss* dans le cache processeur. Sur ce plan, le fichier CAP est donc une bien meilleure approche pour qu'une application Java puisse être mise en un cache efficacement, en fournissant ces deux préconisations.

C'est finalement la seule combinaison qui permette à l'empreinte mémoire et au coût d'exécution du cache de méta-données logiciel de converger vers de bonnes performances globales.

Chapitre 6

Pré-interprétation de code JavaCard

Dans ce chapitre, nous développons notre intuition qu'il est possible d'améliorer les performances d'un cache logiciel en groupant les accès qui y sont fait. L'idée principale part du constat que puisqu'une copie partielle du code est déjà présente en RAM, alors il doit être possible de l'utiliser directement sans contrôles systématiques de sa présence en cache. Cette intuition s'appuie sur les propriétés du code et des méta-données mises en évidence dans les chapitres précédents et que nous avons mesuré avec le degré de séquentialité.

Ce chapitre est structuré en quatre parties. Dans un premier temps, nous introduisons les clés qui mènent de l'approche classique pour implémenter un cache logiciel à notre nouvelle approche basée sur une pré-interprétation du code, en tant qu'étape de construction préalable à une interprétation concrète. Nous présentons ensuite l'architecture générale d'un pré-interpréteur et nous listons les critères qui définissent son seuil de rentabilité. Dans une troisième section, nous prenons comme exemple l'exécution d'applications JavaCard, modèle de post-issuance *le plus répandu dans les cartes à puce, pour décrire en profondeur comment nous pouvons concevoir puis élaborer notre approche. Notamment, nous analysons en détail le composant central de notre approche qui est le pré-décodage des* bytecodes *par une analyse de code. Puis nous étudions l'opportunité de lui greffer, ou non, un mécanisme de pré-chargements de pages de cache. Enfin, dans une dernière partie, nous démontrons l'efficacité et la performance de notre approche à l'aide d'une preuve de concept complète réalisée sur du matériel typique JavaCard 2.2.*

6.1 Vers une pré-interprétation de code

Dans cette section, nous revenons sur les constats présentés en conclusion des chapitres précédents, afin de mieux cerner le nouveau levier de performance que nous y avons implicitement identifié. Ce levier consiste à réduire de manière drastique le nombre d'accès au cache par son interface logicielle puisque c'est bel et bien elle qui constitue le principal goulot d'étranglement identifié chapitre 4.

6.1.1 Interactions entre l'interpréteur et le cache logiciel

L'interpréteur classique d'une JVM est principalement formé d'une boucle déroulant quatre étapes. Un cycle d'interprétation est d'abord initialisé par la récupération de ce *bytecode*, localisé dans une méthode Java par un pointeur d'instruction (IP[1]). L'étape suivante de décodage est la phase d'interprétation au sens littéral du terme puisqu'elle consiste à trouver l'association entre un *bytecode* et le code natif (*handler*) auquel il correspond. La

[1] *Instruction Pointer*, ou parfois aussi appelé PC, pour *Program Counter*.

troisième étapes constite alors à exécuter ce *handler*. Enfin, l'interprétation est finalisée par la modification de IP qui est alors positionné à l'endroit où se trouve le prochain *bytecode* à interpréter ; soit à la suite du *bytecode* courant, soit au point d'arrivée d'un rupteur de flot de contrôle. Généralement, l'opérande de l'instruction est quant à elle récupérée et décodée directement par le *handler*, qui peut également dans certaines circonstances prendre en charge directement la modification de PC.

Listing 6.1: Implémentation basique d'un interpréteur

```
int pc;
bytecode *code;

while(true){
   bytecode bc= code[pc]; /* Récupérer    */
   switch(bc){            /* Décoder   */
      case ADD:
      do_add();           /* Exécuter */
      pc ++;              /* Incrémenter */
      break;
   ...
```

Dans un sytème où une application serait stockée dans une mémoire secondaire puis mise-en-cache, au moins deux étapes peuvent déclencher des défauts de cache ; d'abord dans la phase de récupération du nouveau *bytecode* ; puis dans le *handler* pour récupérer une opérande si elle existe et/ou accéder à des méta-données si le type de *bytecode* le nécessite.

Ces interactions entre l'interpréteur et le cache sont reprises dans le diagramme de séquence 6.1, page 101. Ce diagramme reprend les quatre étapes de l'interprétation décrites dans le Listing 6.1, imbriquées avec le fonctionnement général du cache. Ce diagramme prend l'exemple où la récupération de la prochaine instruction à exécuter provoque un défaut de cache. Plus bas, la récupération de l'opérande quant à elle profite du chargement en cache précédent en déclenchant un *Hit*.

Dans ce schéma classique, l'interpréteur et le cache évoluent de manière strictement cloisonnés et dialoguent par le biais d'une interface logicielle[2]. Cette fonction masque alors complètement à l'interpréteur l'origine des instructions à exécuter. Dans ces conditions, le cache ne sait pas à l'avance ce que va lui demander l'interpréteur. Et de son coté, ce dernier ne sait pas combien de temps mettra le cache à lui répondre.

Cette approche est néanmoins source de simplicité. En effet, l'interpréteur n'a jamais besoin de savoir si tout le corps d'une méthode est disponible ou si seule une copie partielle est accessible immédiatement. Il peut donc fonctionner sans anticiper cette problématique. L'interpréteur ne travaille ainsi que sur des adresses virtuelles qui sont à la fois indépendantes de leur localisation dans l'espace de cache, mais aussi dans la Flash série.

Cette adresse est dite virtuelle car elle est sujette à une double traduction. Une première traduction est réalisé par le cache logiciel lorsqu'il recherche une correspondance avec une adresse physique dans son espace de stockage. Puis une deuxième lors d'un défaut de cache où la FTL traduit cette adresse virtuelle en adresse physique dans la Flash NAND.

[2] *i.e.* du type *data* cache_getDataAtAdress(addr_t address)*

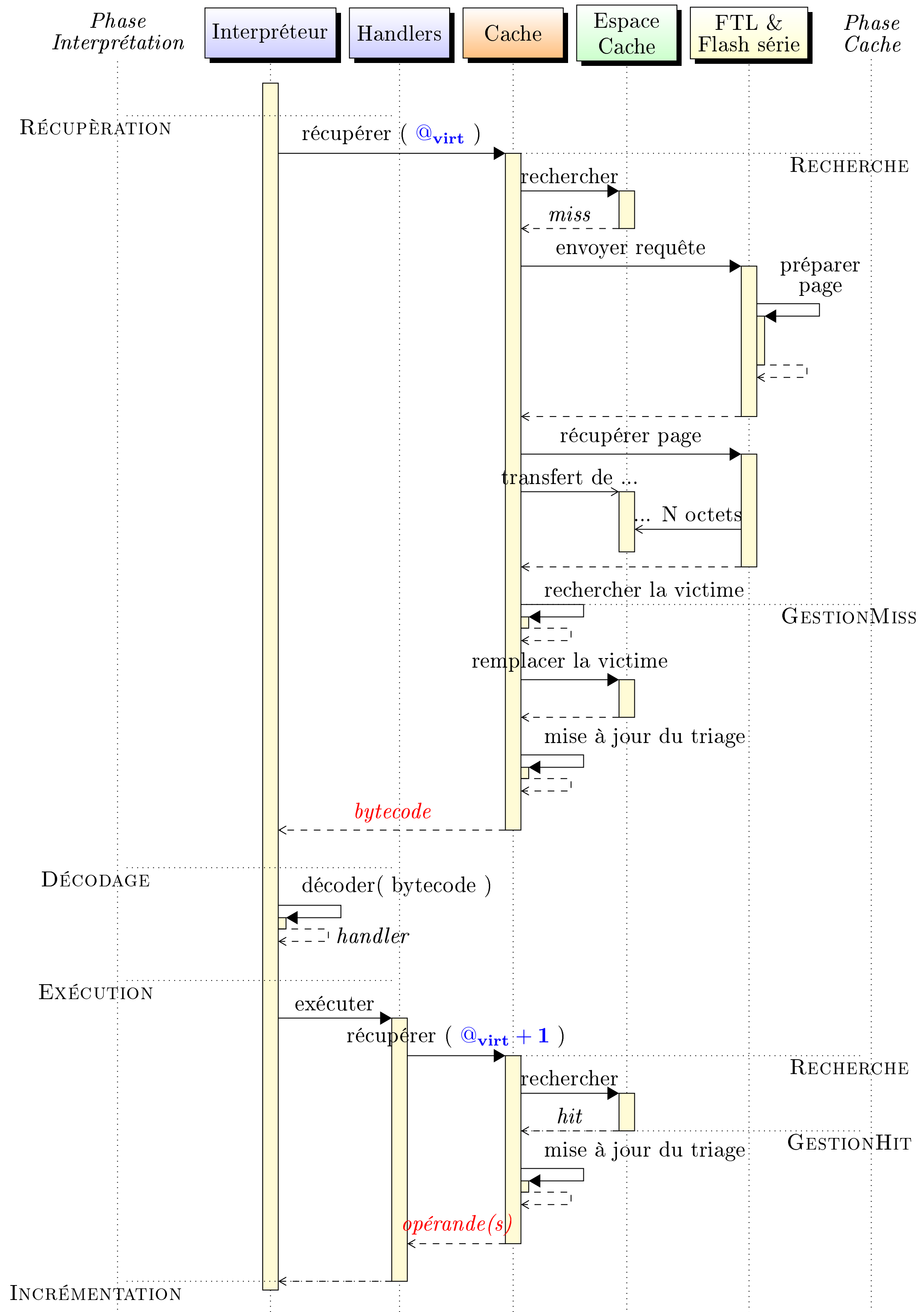


Figure 6.1: Intéraction entre un interpréteur et un cache logiciel

6.1.2 Problème posé par une interaction systématique

Le modéle du diagramme 6.1 est simple et efficace mais connait malheureusement une limite de performance. Dans le chapitre 4, nous avons toutefois remarqué qu'un cache logiciel exploitait très mal la propriété de séquentialité d'un bloc de données, surtout s'il s'agit d'un bloc de base. Il nous est apparu en effet que ce dont a besoin un opérateur d'exécution pour regagner en performance est devenu disponible en mémoire principale grâce au cache. Intuitivement, il semble possible de se passer du cache pour accéder aux données d'un bloc de base.

Cependant, celui-ci impose un accès systèmatique pour chaque donnée et les rend ainsi inatteignables directement. Ce mode de fonctionnement garantit une utilisation sûr des données en cache, car l'interpréteur n'est pas maître du contenu du cache.

Prenons un exemple. Comme point de départ, supposons que l'interpréteur ait besoin d'accéder à une valeur K à l'adresse virtuelle $@_{virt}$. Imaginons ensuite que le cache, au lieu de renvoyer K, renvoie l'adresse physique $@_{phy}\{P_0+8\}$, un offset dans la page n°0 du cache, en RAM, où se trouve une copie de la valeur K. Si maintenant le pointeur d'instruction (IP) courant de l'opérateur d'exécution est redirigé vers $@_{phy}\{P_0+8\}$ plutôt que $@_{virt}$, on peut penser « naïvement » que le programme puisse continuer son exécution sans devoir à nouveau requêter le cache, simplement en incrémentant IP comme il se doit.

Or, ce principe de fonctionnement est invalide car, de fait, rien n'indique que IP continue à lire et exécuter du code valide au-delà de $@_{phy}\{P_0+8\}$. Tout simplement parce que cette adresse est une adresse dans une page de cache, et que $@_{virt}+1$ peut en réalité se trouver dans une autre page de cache, à l'adresse $@_{phy}\{P_5+0\}$ par exemple, et non à $@_{phy}\{P_0+9\}$. Ce phénomène est dû à la politique de remplacement et aux branchements dans le programme qui font que deux pages de cache adjacentes physiquement se retrouvent inévitablement[3] à contenir des copies de blocs de données non-contigües du binaire du programme.

Dans ce cas, l'interpréteur ne sait pas qu'il n'évolue plus sur une plage d'adresses physiques continues et son comportement devient alors indéterminé. À ce stade, la seule parade est donc bien d'effectuer un accès au cache par son interface logicielle pour chaque instruction ou opérande à récupérer.

6.1.3 Dépasser le modèle d'interactions systématiques

6.1.3.1 Identification du verrou

L'intérêt avéré d'un cache est qu'il gomme pour une bonne part le plus gros défaut de la Flash série qu'est sa latence. Cependant, sa rigidité et sa propre latence entraînent les conditions d'un usage souvent contre-productif des données en cache.

En effet, si le cache ne garantit pas que deux adresses virtuelles soient physiquement contigües dans la RAM, cela ne veut pas dire qu'elles ne le sont pas réellement. Au contraire, elles le sont effectivement la plus part du temps, et nous avons mesuré cette probabilité grâce au degré de séquentialité introduit chapitre 4. Le cache forme donc encore une frontière, logicielle, qui empêche une exécution plus « naturelle » de nombreuses séquences d'instructions.

Cependant, cette barrière ne repose en fait que sur l'absence à un instant T d'une `garantie` de continuité d'adresses physiques qui serait à l'image de la continuité avérée d'une plage d'adresses virtuelles. Dans ces conditions, réussir à offrir cette garantie, quelque soit l'instant, permettrait de dépasser ce verrou que constitue le modèle d'interactions systématiques.

[3] Le contraire serait le fruit du hasard.

Nous avons montré dans le chapitre 4, section 4.2.3.3, page 68, que le CoI moyen d'un accès multiplié par leur très grand nombre constituait le principal goulot d'étranglement entre l'interpréteur et le cache logiciel. Par conséquent, ce coût est imputable essentiellement à ce modèle d'interactions systématiques. Si une solution est capable de réduire le nombre d'accès au cache nécessaire à l'exécution d'un bloc de base, alors le gain espéré se mesure facilement grâce au degrès de séquentialité. En effet, si des blocs de base ont une taille moyenne de 20 octets, alors le nombre d'accès évitables peut atteindre dans le meilleur des cas 95 %. Et ainsi réduire de manière significative la latence propre d'un cache logiciel.

6.1.3.2 Clés de l'efficacité

Soit un outil *OutilAnalyse* du cache, avec son coût propre, mais qui permet d'identifier dans l'espace de stockage RAM du cache des plages d'adresses physiques, délimitées et sûres. Soit ensuite, les éléments formant le temps de latence d'un cache logiciel, tel que nous les avons analysés dans le chapitre 4, page 43 (*e.g.* $N_{Accès}$, $T_{Recherche}$ et $T_{GestionHit}$).

Alors une solution basée sur un autre modèle que le modèle systématique aura un intérêt si elle reste en dessous du seuil de rentabilité définit pas l'inéquation suivante (où N signifie nombre et T signifie temps) :

Inéquation 2. *Seuil de rentabilité de notre approche*

$$\begin{cases} Coût_{OutilAnalyse} & < & Coût_{Accès\,cache\,évités} \\ Coût_{OutilAnalyse} & = & T_{OutilAnalyse} * N_{OutilAnalyse} \\ Coût_{Accès\,cache\,évités} & = & N_{Accès,évités} * (T_{Recherche} + T_{GestionHit}) \end{cases}$$

6.1.4 Vers une pré-interprétation de code

6.1.4.1 Clés de conception

Pour que l'interpréteur puisse travailler temporairement sur des adresses physiques en RAM dans l'espace de cache plutôt que sur des adresses virtuelles par l'interface de cache, le cadre d'usage sûr et strict se définit selon les quatre points suivants.

1. Le premier est la définition d'un couple adresse-virtuelle/adresse-physique marquant le début d'une plage d'adresses accessibles directement par l'interpréteur.
2. Le deuxième est la détermination à l'avance de la fin de cette plage d'adresses.
3. Le troisième est la garantie que cette plage ne soit ni déplacée, ni supprimée par le cache tant que l'interpréteur l'utilise directement.
4. Le quatrième est l'assurance à l'avance que l'interpréteur n'utilise pas le cache pour autre chose tant qu'il travaille encore sur cette plage, pour éviter que ce dernier ne viole la troisième condition par renouvellement forcé.

Pour le quatrième point cependant, ce point est alors souhaitable mais non-obligatoire, dans le sens où une solution peut consister à figer plusieurs blocs de donnée en cache en même temps, dans des pages différentes. Dans ces conditions, d'autres précautions doivent être prises, revenant à multiplier les point 1 à 3. Dans le cas des langages interprétrés, cet aspect peut typiquement servir à couvrir le cas des méta-données.

Le troisième point quant à lui est la condition centrale. Pour qu'elle soit satisfaite, il faut que la plage d'adresses soit négociée entre le cache et l'interpréteur à l'avance. Il faut donc que les adresses de début et de fin soient fixées avant l'utilisation directe de la plage par l'interpréteur pour régler le problème de débordement de IP. La détermination de limites de plages doit ainsi être évaluée dès le moment du choix de l'adresse de début.

6.1.4.2 Détermination des limites de plages d'adresses physiques

La plage d'adresse physiques idéale correspond à un bloc de base complet car il s'agit de la plus petite unité de séquentialité d'un programme. Malheureusement, cette information connue à la compilation n'est plus disponible dans le binaire exécutable. L'idée principale de notre approche est donc de reconstituer artificiellement dans l'espace de stockage du cache, les limites de début et de fin d'un bloc de base. C'est ce bloc qui sera ensuite donné à l'interpréteur pour exécution.

Dans ces conditions, l'adresse de départ est l'adresse de l'instruction de tête du bloc de base, associée à la page de cache courante qui la contient. La reconstitution d'un bloc consiste alors à analyser le code qui débute immédiatement à cette adresse pour détecter à l'avance à quel moment :

- IP finira par pointer vers un rupteur de flot de contrôle sortant de la page de cache courante ;
- IP finira par pointer vers l'adresse physique du dernier octet de la page de cache courante (qui peut ne pas être un rupteur de flot de contrôle) ;
- la/les méta-données d'une opérande sont résolvables sans défaut de cache.

L'évaluation de la première condition dépend donc de la détection de rupteurs de flot et la seconde nécessite une connaissance de la configuration du cache et du contenu de celui-ci à un instant T. Pour le contexte de la *post-issuance*, ces dépendances obligent donc que cette évaluation soit réalisée en-ligne, car c'est le seul endroit où tous ces paramètres sont connus. Une application *post-issuance* ne peut en effet pas être à la fois indépendante du système cible et compilé pour les propriétés d'un système précis. À la différence de la JVM, le cache et la Flash série ne sont pas standardisés et homogènes et sont donc à considérer comme différents, voir absents, d'une carte à l'autre.

6.1.4.3 Préparation précoce de l'interprétation concrète

Pour préparer, en-ligne, la plage d'adresses physiques et détecter une adresse de fin adéquate, nous proposons une approche basée sur une analyse du code. Cette approche offre une série d'optimisations des échanges de données entre l'interpréteur principal et la mémoire non-adressable, tout en étant complémentaire au cache.

La première optimisation consiste donc à laisser l'interpréteur concret[4] travailler directement dans l'espace de stockage physique du cache. L'objectif est de réduire drastiquement le nombre d'accès à ce dernier par l'interface d'adresses virtuelles.

La deuxième optimisation consiste à utiliser les résultats intermédiaires de l'analyse pour pré-interpréter les *bytecodes* analysés. Cette démarche consiste à réaliser à l'avance les étapes de récupération et décodage de l'interprétation concrète sur chaque *bytecode* analysé.

[4] *i.e.* l'interpréteur de la JVM, qui exécute concrètement le code, par opposition à l'interpréteur partiel qui se charge de l'analyse de code.

La troisième et dernière optimisation, consiste à détecter à l'avance de possibles défauts de cache et soumettre un pré-chargement des informations manquantes à la mémoire non-adressable. Cette détection est elle aussi rendue possible par l'analyse anticipée du code qui évalue sa présence en cache. Cette approche permet donc de résoudre plus simplement les approches de pré-chargements plus complexes décrites section 3.3.2.3, page 32. Elle peut permettre également d'anticiper des chemins critiques de méta-données plus profonds que l'opérande.

Ces trois optimisations des échanges de données entre l'interpréteur et la mémoire Flash non-adressable, s'ajoutant à celle que constitue un cache, forment donc une phase homogène de `pré-interprétation` consistant en :

- une **préparation** d'un groupe d'instructions interprétables concrètement, séquentiellement et sans accès « logiciel » au cache d'instructions ;
- un **pré-décodage** des instructions de ce groupe, qui équivaut à une interprétation partielle et précoce de chaque instruction qui le constitue ;
- une **anticipation** des défauts de cache susceptibles d'intervenir lors de l'exécution du prochain groupe d'instructions où lors de l'accès à des méta-données.

En d'autre terme, nous proposons de **scinder** l'interpréteur originel en deux phases d'interprétation distinctes : une pré-interprétation manipulant le code de façon anticipée pour préparer une interprétation concrète, différée et indépendante du gestionnaire de cache.

6.2 Conception d'un pré-interpréteur

La conception d'un pré-interpréteur est guidée par les quatre principes de l'encadré page 103. À ces principes opérationnels se greffe la contrainte impérative de produire une pile d'accès à la mémoire non-adressable ayant une empreinte mémoire maîtrisée et faible. Sachant que dans cet ensemble d'algorithmes et d'occupation mémoire que constitue cette pile, le cache occupe déjà une part importante de la consommation d'espace mémoire.

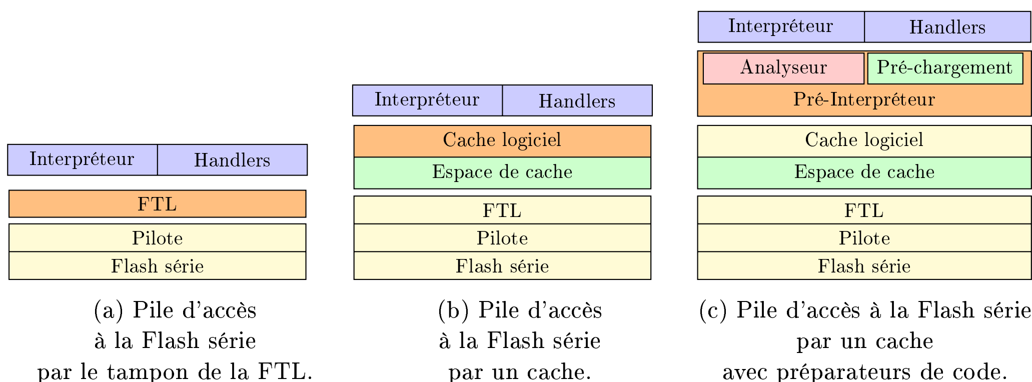
Nous présentons dans cette section comment construire un outil de pré-interprétation de code sachant maîtriser à la fois son empreinte mémoire mais également le coût de son exécution.

6.2.1 Architecture générale

6.2.1.1 Surcharge de la pile d'accès à la mémoire non-adressable

La figure 6.2a, page 106 rappelle la constitution de la pile d'accès à la mémoire non-adressable par une exécution en place dans un tampon. La figure 6.2b représente celle de l'utilisation classique d'un cache. La figure 6.2c décrit quand à elle l'architecture que nous proposons. Elle garde l'utilisation d'un cache mais place entre lui et l'interpréteur une nouvelle couche de gestion de l'exécution qui ordonnance la pré-interprétation, l'interprétation concrète différée et la gestion anticipée des défauts de cache.

Dans ces schémas, nous séparons l'interpréteur concret principal qui consomme essentiellement des *bytecodes*, et les *handlers* qui consomment à la fois du code (*les opérandes*) et des méta-données. Comme nous l'avons vu ces deux types de données ne sont pas utilisés de la même façon. Nous avons vu notamment que les méta-données modifiaient la notion de séquentialité d'un bloc de base en lisant des données dans une page de cache potentiellement différente. Cet aspect est donc un point important de la recherche de plages séquentielles



(a) Pile d'accès à la Flash série par le tampon de la FTL.

(b) Pile d'accès à la Flash série par un cache.

(c) Pile d'accès à la Flash série par un cache avec préparateurs de code.

Figure 6.2: Piles d'accès à la Flash série.

qui ne rompent pas la condition n°3 de l'encadré page 103, *i.e.* la modification intempestive du cache.

6.2.1.2 Principe de fonctionnement

La boucle principale qui est à l'origine de l'interprétation classique est remplacée pour une boucle étendue regroupant un pré-interpréteur et un interpréteur concret. Un cycle complet est représenté dans le diagramme de séquence 6.3, page 107. Dans ce diagramme apparaissent toujours les phases classiques de l'interprétation d'un *bytecode*, à savoir la récupération de l'instruction, son décodage, son exécution, et le repositionnement du pointeur d'instructions. Il n'y a pas de changement dans l'ordre de ces phases car elles ne peuvent pas être interverties. Cependant, le changement d'approche globale que nous proposons conduit à réaliser des actions différentes au sein de chacune de ces phases.

Récupération L'étape de récupération s'attache à transformer l'adresse virtuelle d'une instruction de tête vers son adresse physique en RAM. Si cette instruction est absente du cache, alors elle est récupérée depuis la mémoire non-adressable par le canal normal du cache. À cette étape, l'instruction récupérée est toujours et uniquement une instruction de tête, propriété garantie par un cycle complet, et notamment par l'analyse du code.

La refonte de cette étape est la résolution de la condition n°1 de l'encadré page 103, à savoir, la définition d'un couple adresse-virtuelle/adresse-physique marquant le début d'une plage d'adresses accessibles directement par l'interpréteur concret.

Décodages multiples Le remodelage de cette étape de l'interprétation classique est le cœur de notre nouvelle approche. C'est en effet à cet instant que se déroule l'analyse de code dont le but est la délimitation d'une plage d'adresses en RAM, « exécutable » par l'interpréteur concret sans accès au cache. Le travail d'analyse a pour but de mettre en œuvre de la condition n°2 de l'encadré page 103, à savoir la détermination des limites de la plage d'adresses physiques.

Indirectement, mais effectivement, cette analyse participe à la récupération et au prédécodage de tous les *bytecodes* que contiendra cette plage, mais aussi certaines méta-données si elles sont analysables hors-contexte d'exécution. En effet, à fin d'avancer dans le code du bloc de base pour en trouver la limite, chaque *bytecode* est décodé pour estimer s'il s'agit d'un rupteur de flot, pour calculer l'adresse du *bytecode* suivant, mais aussi évaluer certaines opérandes accédant à des méta-données. Ainsi, de façon précoce, le bloc de base complet

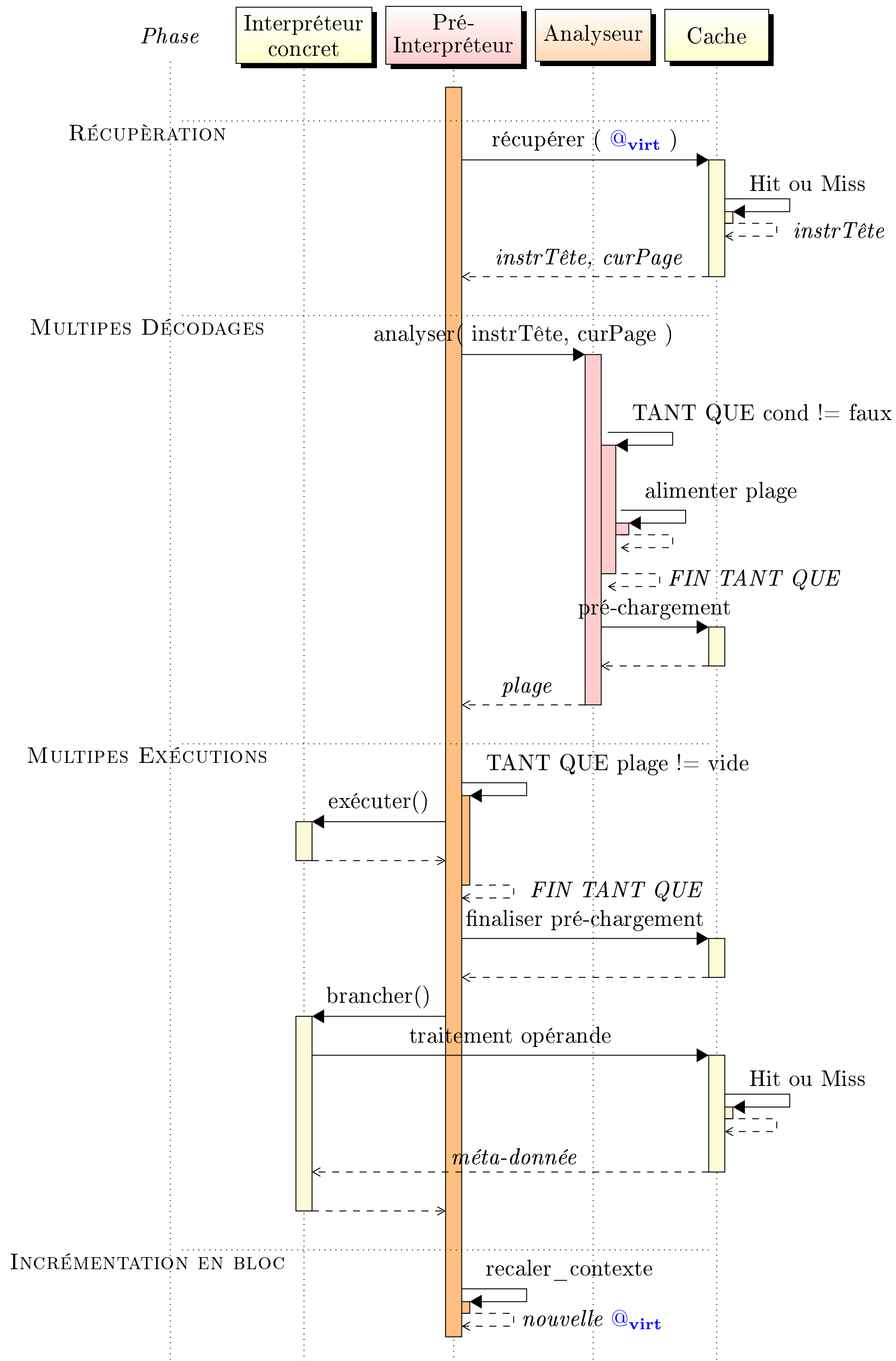


Figure 6.3: Schéma de pré-interprétation de code avec décodages et exécutions groupés de *bytecodes*

sera entièrement pre-décodé en fin de cycle d'une analyse. Enfin et surtout, on peut dire alors que chaque *bytecode* a été « artificiellement » récupéré, sans qu'il n'y ait eu besoin d'accéder au cache par son interface logicielle.

Exécutions multiples À cette étape, la plage d'adresses décodée est fournie à l'interpréteur concret pour exécution. Son pointeur d'instruction est positionné en début de plage physique et commence l'interprétation concrète de *bytecodes* déjà décodés. La plage entière est alors exécutée dans les même conditions de performance que si l'application était stockée dans une mémoire adressable.

Incrémentation par bloc Le calcul de la plage d'adresse principale (*i.e.* celle du bloc de base, hors méta-données) fixe un IP physique de départ et un IP physique de fin. Lors de l'exécution de l'interpréteur concret, IP courant est contrôlé après chaque exécution de routines d'interprétation jusqu'à ce que celui-ci soit égal à celui de fin. Il est alors retraduit en adresse virtuelle pour que puisse recommencer un nouveau cycle de pré-interprétation.

Description du nouvel ordonnancement Les conséquences et effets des différences d'approche entre l'interpréteur classique et un pré-interpréteur sont illustrées par les figures 6.4 et 6.5.

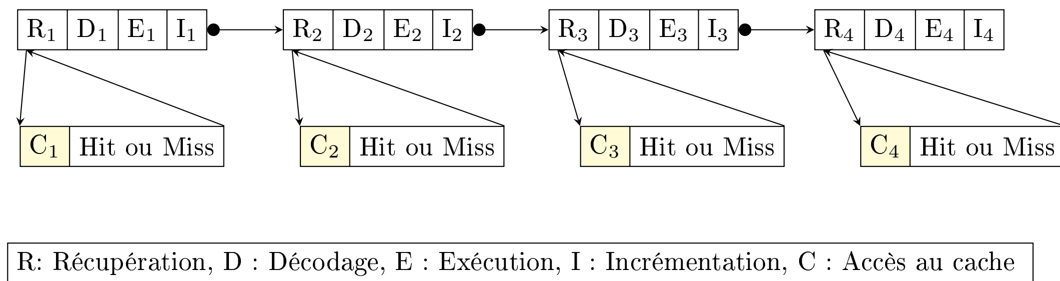


Figure 6.4: Interprétation classique avec accès systématiques au cache

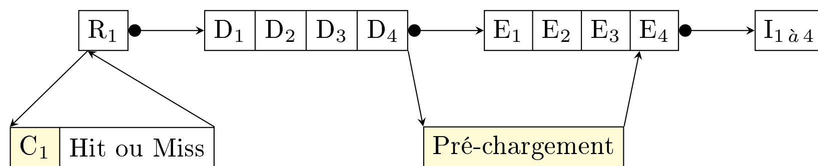


Figure 6.5: Cycle de pré-interprétation

Fondamentalement, la pré-interprétation ne réduit pas le temps d'exécution d'un décodage ou d'une exécution de *handler*. Là n'est pas le but. Cependant, le fait de décoder précocement les instructions permet de réduire l'impact du coût de l'analyse de code sur chaque cycle d'interprétation.

Le point important est sans conteste le nombre d'accès au cache évités. C'est sur cet aspect et sur le gain qu'il représente que repose la performance finale de la pré-interprétation par rapport à l'approche classique.

Le second gain possible est celui-ci des pré-chargements de blocs de données. Grouper des décodages, permet de les décorréler de leur exécution immmédiate. L'analyse permet de voir plus loin que le *bytecode* courant, ce qui permet une recherche anticipée d'autres *bytecodes* susceptibles de déclencher des défauts de cache. Lancer un pré-chargement permet

ainsi d'exécuter une série de *bytecodes* pendant que la Flash non-adressable prépare la page de données à charger en cache.

6.2.2 Objectif : maîtriser le coût d'exécution

La fréquence des analyses est au moins égale au nombre de blocs de base exécutés durant le cycle de vie d'un programme. Ce nombre augmente avec le nombre de blocs de base tronqués soit par la segmentation soit l'accès à des méta-données dont la présence en cache ne peut pas être évaluées lors de la phase d'analyse statique. Toutefois, quoi qu'il arrive, ce coût restera acceptable si et seulement si il est inférieur au coût des accès au cache évités.

De plus, si le gain le permet, et si l'efficacité reste suffisante, il sera même alors possible de réduire l'empreinte mémoire du cache, bénéfice tout aussi intéressant. Pour ce faire, il faut alors compter sur le fait que le temps gagné par la pré-interprétation puisse compenser les temps de latence de quelques défauts de cache supplémentaires, induits par un espace de cache plus petit. Pour avoir une empreinte mémoire faible, le succès de notre approche repose donc finalement exclusivement sur l'autre challenge de la maîtrise du coût d'exécution de la pré-interprétation.

6.3 Pré-interprétation de code JavaCard

Pour poursuivre la démonstration de notre approche, nous nous baserons sur l'exemple de JavaCard 2.2, le modèle de *post-issuance* le plus répandu. Nous introduisons cette section par une brève présentation des spécificités du code JavaCard 2.2. Puis, nous décrirons plus en détail notre approche basée sur une pré-interprétation du code JavaCard ainsi que de sa problématique de méta-données. Enfin, nous reviendrons sur les incidences de notre approche sur le cache logiciel en tant que tel.

6.3.1 Code JavaCard 2.2

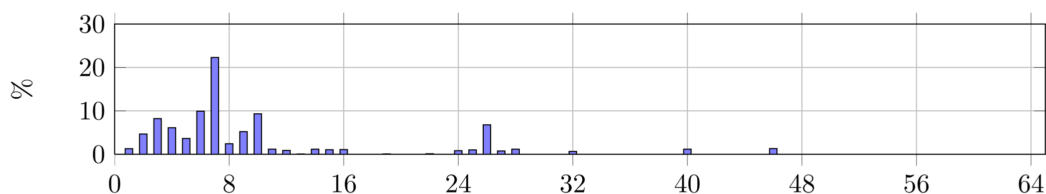


Figure 6.6: Distribution des tailles de blocs de base en octets de JCProfil.

Les informations qui nous intéressent plus spécifiquement sur le code JavaCard sont des statistiques sur les blocs de bases, sur ce qui les entourent, et sur les types de *bytecode* qui constituent leur rupteur de flot. Comme nous l'avons vu, ces informations sont à l'origine des performances mais aussi des opportunités inexploitées d'un cache d'instructions en logiciel. Le bloc de base est en effet la pierre angulaire du principe de localité mais aussi des constats de notre analyse, que ce soit le degré de séquentialité, le placement par segmentation et la dilution spatiale, ou encore l'unité de stockage du cache.

Ces informations nous permettrons d'étayer la suite de cette section sur l'analyse et la pré-interprétation de code JavaCard. Le tableau 6.1 fournit des statistiques sur les blocs de base dans les fichiers CAP, pour 3 exemples d'applications, et dans la dernière colonne pour tous les CAP à disposition. Tandis que le graphique 6.6 montre la distribution des blocs

Moyennes	RichardsV1	RichardsV7	JCProfil	Panel
Bytecodes par méthodes	29	11	24,1	25,3
Taille des blocs de base	6,7	4	6,5	7.7
Bytecodes par blocs de base	4,3	2,7	3,8	4

Tableau 6.1: Structure du code JavaCard par granularité

Catégorie	Tous *bytecodes* confondus	Rupteurs de flot seuls
Branchements conditionnels	5,5 %	23,8 %
Branchements inconditionnels	2,84 %	11,9 %
Appels de méthodes	10,43 %	43,83 %
Retour de méthodes	4 %	16,95 %
Création d'objets	0,36 %	1,55 %
Contrôle d'objets	0,19 %	0,82 %
Switch/throw	0,28 %	1,16 %
Autres	76,2 %	-

Tableau 6.2: Répartition des *bytecodes* par type en JavaCard

de base pendant l'exécution de l'application JCProfil (voir page 40), par rapport à leur taille, en octets (*i.e.*, instruction et opérandes). Enfin, le tableau 6.2 fournit la proportion de chaque type de *bytecode*. La première colonne tient compte de tous les *bytecodes*, tandis que la deuxième donne les pourcentages des rupteurs de flot uniquement.

De manière générale, on constate qu'un bloc de base JavaCard n'est pas très large et ne compte que 4 *bytecodes* en moyenne, pour 3,7 octets d'opérandes. Ce qui en fait des programmes relativement peu séquentiel par rapport à des programmes compilés. Il est toutefois intéressant de noter que cela n'enlève rien à la possibilité de tirer partie de cette séquentialité lorsqu'il s'agit de la confronter à la problématique des accès au cache. Un rapport de 1 pour 7,7 reste une opportunité de gain important.

6.3.2 Pré-décodages : mode opératoire

Préambule L'analyse du code a deux objectifs. Le premier doit réduire le nombre d'accès au cache et gagner un maximum de temps sur la latence du cache. Le second objectif est d'anticiper le plus tôt possible un défaut de cache.

Sans ce second objectif, l'analyse n'est pas obligatoirement fonction d'un bloc de base complet et pourrait se limiter à quelques instructions, voire une seule. L'esprit des accès au cache évités resterait alors valide si une analyse intermédiaire se portait garant du respect des couples d'adresses physiques/virtuelles. Cela augmente donc le nombre de recontrôle de présence de données en cache entre chaque phase d'exécution de *bytecodes*. Certes ce processus est moins long *a priori* que l'interface logicielle de cache mais vaut donc le coup d'être.

6.3.2.1 Support d'analyse

L'analyse du code avance en évaluant un *bytecode* à la fois pour déterminer la taille de son opérande, puis contrôler s'il s'agit d'un rupteur de flots. Cette évaluation est généralement implémentable de deux manières différentes.

La première est un *switch/case* dans lequel chaque *bytecode* représente un cas et chacun de ces cas renvoie la/les valeurs correspondantes. Cette approche est relativement coûteuse, d'une part en empreinte mémoire de code, mais surtout en coût d'exécution du fait d'un grand nombre de branchements.

La deuxième façon de procéder consiste à utiliser un tableau contenant une pré-évaluation. Dans ce mécanisme, un *bytecode* est un index dans le tableau dont l'entrée contient la valeur associée. Dans cette approche et aux vues du nombre de *bytecode*, un tableau de 190 entrées est nécessaire. Au sein de ce tableau, pour réduire l'empreinte mémoire, les informations de taille d'opérande et de type de branchement peuvent être fusionnées sur un octet : 4 bits pour la taille, 4 bits pour le type.

6.3.2.2 Éviter la redondance des décodages

L'analyse a besoin de décoder un bytecode pour avancer. Selon la façon dont l'interpréteur est implémenté, ce décodage peut reservir pour ne pas avoir besoin de re-décoder le *bytecode* pour identifier sa routine d'interprétation avec son exécution. Trois techniques sont généralement utilisées pour implémenter le cœur d'un interprétateur. Elles sont présentées dans l'encadré page 112.

Pour des raisons de portabilité, la JCVM dont nous disposons utilise la technique de *Indirect Threading*. Dans cette approche, le bytecode est décodé avec la même méthode que notre analyseur. Par conséquent, pour éviter effectuer une seule indirection mémoire, nous fusionnons le tableau de *handlers* avec le tableau de l'analyseur, ou une entrée est alors de 5 octets (1 octet pour l'analyseur, 4 pour le pointeur de fonction).

Cependant, une fois le handlers obtenu, celui-ci doit être mémorisé jusqu'à la fin de la phase d'analyse. Ce qui ré-ajoute un coût d'accès mémoire pour le CPU, pour accéder à l'endroit mémoire où est stocké le pointeur de fonction décodé. Ce coût se présente de la façon suivante :

Listing 6.2: Mémorisation et exécution des routines d'interprétation

```
//Déclaration
typedef  void (*jcvm_handler) (void);
//stockage temporaire des handlers
jcvm_handler fifo_handler[FIFO_SIZE];
...
//Décodage durant l'analyse :
fifo_handler[basic_bloc_size++] = decoding_table[bytecode].handler
    ;
...
//Phase d'éxécution :
for(i=0; i < basic_bloc_size ;i++)
{
   fifo_handler[i]();
}
...
```

L'approche du Listing 6.2 peut être améliorée en modifiant à la volée de code conçu pour appeler les *handlers* les uns à la suite des autres, en s'inspirant du *Direct Threading*. Puisqu'il faut mémoriser le décodage, il est alors plus intéressant d'avoir du code sachant éviter les faiblesses de la phase d'éxécution du Listing 6.2 que sont : un nombre important de branchements, et un nombre important d'accès indirect à la RAM.

Techniques d'implémentation d'interpréteurs.

1. La technique dite *Decode-and-dispatch*, basée sur un *switch* comme dans le Listing 6.1.1, page 100. Cette technique est toutefois connue pour être peu performante du fait d'un nombre de branchements importants [Ertl 2001].

2. Deux techniques classées sous le terme *Threaded interpretation* [Klint 1981] dont le but est l'élimination des branches et de leur coût.

   Direct Threading [Bell 1973] : dans cette approche, l'instruction intermédiaire est remplacée à l'exécution par l'adresse de la routine d'interprétation, supprimant ainsi toutes les indirections liées au décodage des instructions. Pousser cette logique plus en avant, aboutit alors aux compilateurs JIT.

   Indirect Threading [Dewar 1975] : dans cette approche, les branchements sont remplacés par un tableau contenant les adresses de chaque routine d'interprétation. Cette approche est plus simple à mettre en œuvre, légèrement[a] moins performante que la précédente, mais offre l'avantage d'être indépendante de l'architecture processeur [Smith 2005].

3. Si le compilateur du matériel cible et le *design* de la JVM le permettent, une autre solution élégante se trouve dans la fonctionnalité de *Labels as Values* de GCC[b]. Cette approche d'*Indirect Threading* repose sur des adresses de labels (*goto*) plutôt que des appels de fonctions, ce qui la rend souvent plus performante mais plus structurante.

[a] Cela dépend du processeurs, voir http://www.complang.tuwien.ac.at/forth/threading/. De plus, cette technique a un impact fort sur le cache de données du CPU. Cependant, une carte à puce n'en possédant pas, cet inconvénient y est alors moins préjudiciable.

[b] http://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html

Nous avons spécialement écrit du code assembleur efficace dont une partie dédiée est modifiée à la volée. Le code est copié en RAM à la mise sous-tension[5] comme peut l'être parfois le code de démarrage (*bootloader*) ou une fonction d'effacement de la Flash NOR à chaud (*In-system programming*). Cette relocation du code dans un tampon en RAM permet la modification d'une partie de son contenu plus rapidement que s'il était dans l'espace de code standard en NOR.

Lors des décodages, les adresses de handlers sont stockées en fin de tampon avant que celui-ci ne soit plus tard exécuté comme une fonction. Ainsi, seules les adresses ont besoin d'être changées en écrasant les valeurs précédentes. Cette technique n'utilise que des accès directs à la RAM et supprime tous les branchements.

Le corps de la fonction du tampon est donné en assembleur ARM dans le Listing 6.3. Pendant l'exécution, les propriétés du code ARM font que le mode de contrôle d'un handler nul s'effectue très rapidement sans branchement conditionnel et donc séquentiellement sans rupture de flot. Ce qui garantit un état optimal du CPU.

[5] avec les données de la section *.data*

Listing 6.3: Code généré pour l'exécution des routines d'interprétation

```
ldr    r4, .L5         ;charger adresse du premier handler
cmp    r4, #0          ;si handler != 0
movne  lr, pc
bxne   r4              ;appeler premier handler
ldr    r4, .L5+4       ;charger adresse du deuxième handler
cmp    r4, #0          ;si handler != 0
movne  lr, pc
bxne   r4              ;appeler deuxième handler
...
.L5:                   ;stockage des adresses de handlers
  .word  229509        ;adresse #1
  .word  251280        ;adresse #2
  .word  0             ;fin de plage de handlers décodés
  ...
  .word  0             ;
```

Pour être le plus efficace possible entre empreinte mémoire et vitesse d'exécution, le nombre d'entrée associée au code ci-dessus est fixée à quatre. Ce qui correspond à la taille moyenne des blocs de base JavaCard. Cette façon de procéder est ainsi deux fois plus rapide que celle du Listing 6.2.

6.3.3 Analyse et rupture de flots

Après avoir donné une description opérationnelle de ce qu'est notre implémentation du cœur du pré-interpréteur (*e.g.*, analyse de code, pré-décodage multiple et exécution multiple), nous abordons dans cette section le traitement des rupteurs de flots et des méta-données.

6.3.3.1 Preuve d'une analyse bornée

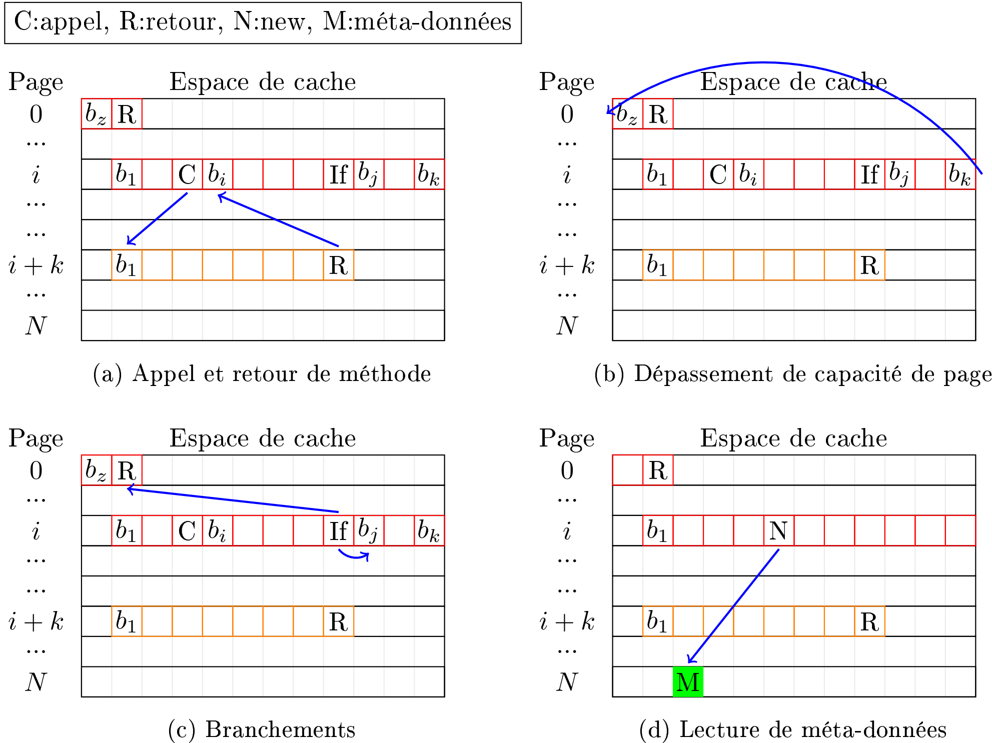
La condition *sine qua none* pour l'utilisation d'une analyse de code en-ligne est de fournir la preuve que celle-ci s'arrêtera et ne bouclera pas à l'infini. Nous avons fixé une limite, page 104, que nous rendons maintenant opérationnelle.

L'exécution symbolique qu'effectue l'analyse ne doit jamais sortir d'un nœud du graphe de contrôle. Elle n'empreinte donc jamais d'arc sur ce graphe et s'arrête alors sur un rupteur de flot. Néanmoins, l'analyse ne peut se terminer que sur la détection de cette seule condition. Elle doit également prendre fin lorsque le flot d'instructions exécuté symboliquement ne correspond plus à une plage d'adresses linéaires en RAM – au moins lorsqu'il s'agit de code.

Quatre conditions d'arrêt sont donc ainsi possibles pour déterminer si l'analyse sort de l'espace de cache :

1. un appel ou un retour de méthode (figure 6.7a) ;
2. un saut conditionnel ou inconditionnel (figure 6.7c) ;
3. un dépassement de page (figure 6.7b) ;
4. l'accès à une méta-donnée (figure 6.7d) ;
5. la limite fixée pour la profondeur d'analyse est atteinte.

Pour les conditions 3 et 4, le contrôle doit être actif sur des propriétés autre que le code, et qui sont observables par un instantané du contenu du cache au démarrage de l'analyse.



(a) Appel et retour de méthode

(b) Dépassement de capacité de page

(c) Branchements

(d) Lecture de méta-données

Figure 6.7: Conditions d'arrêt d'analyse

Pour les conditions 1 et 2, la sémantique du code Java garantit qu'un bloc de base se termine toujours par un rupteur de flot[6], y compris pour les retours de méthodes de type *void*. Pour que cela soit valide, il faut bien évidemment que le code analysé ait été vérifié lors du chargement de la classe associée, ce que nous supposons comme acquis. Cette vérification contient également la preuve que tous les rupteurs de flots ont un bloc de base de destination valide. L'étape de recalcul de contexte effectuée en fin de cycle de pré-interprétation interviendra donc sur un *bytecode* aux propriétés valides. Ce qui permet un recalcul de l'adresse virtuelle d'IP plus simple et sûr, sans contrôle explicite de la destination.

Tout ceci garantit donc qu'une borne existe, quelque soit le lieu de départ de l'analyse de code. Y compris pour les cas sans rupture de flots comme les blocs tronqués par des fins de pages de cache.

6.3.3.2 Pré-décodage de méta-données

Nous avons vu dans le chapitre précédent, que certains *bytecodes* nécessitaient un processus de résolution de méta-données à partir de leur opérande. De plus, ces *bytecodes* ne sont pas tous des rupteurs de flots et ne stoppent donc pas tous l'analyse sur les mêmes modalités.

Notre analyseur garantit jusqu'ici que le code pré-interprété donné à l'interpréteur concret est sans défaut de cache ni manipulation intermédiaire de celui-ci, comme l'impose le point n°3, point crucial du fonctionnement **sûr** de notre approche (voir encadré page 103). Or, les lectures de méta-données par ces *bytecodes* non-rupteurs de flot[7] cassent cette cer-

[6] En considérant les mécanismes de levée d'exceptions comme des rupteurs de flot, et la clause *finally* comme une sous-routine.

[7] Accesseurs aux champs statiques et d'instances (GETSTATIC_x, PUTSTATIC_x, GETFIELD_x, PUTFIELD_x), le *bytecode* de création d'*objets* (NEW) et celui de tableaux d'*objets* (ANEWARRAY), et

titude.

Un traitement particulier doit donc leur être réservé. Ce traitement dépend des optimisations qui ont pu être appliquées à ces *bytecodes* comme cela a été discuté section 5.4.2, page 85, à savoir la résolution du *ConstantPool* et notamment la Phase 1, commune à tous ces *bytecodes*.

Nous distinguerons deux types de résolution d'opérande, une simple et une complexe. La simple correspond à aucune ou une seule et unique lecture de méta-données, par opposition à la complexe qui requiert quant à elle au moins deux lectures. L'optimisation citée précédemment est donc un facteur discriminant de résolution d'opérandes. Si le CAP n'a subit aucune optimisation du *ConstantPool*, alors toutes les résolutions d'opérandes sont complexes car elle nécessite toujours deux phases comme dans le cas présenté section 5.4.2.

1. **Aucune lecture** : le cas le plus simple. C'est le cas par exemple des *bytecodes* de type GETFIELD_x où l'optimisation consiste à remplacer l'opérande résolue par un index ou un offset dans la structure mémoire d'un objet dont l'entrée contient la valeur du champs demandé. Cet *objet* est obtenu par le *handler* en introspectant la pile, et ne déclenche donc aucune lecture en dehors de la plage d'adresses en cours de constitution.

2. **Une seule lecture** : ce cas est celui-ci par exemple du *bytecode* NEW. Lors de la résolution, la Phase 1 a décodé une structure *class_ref* vers une structure concrète de type *class_info* (voir le modèle JavaCard, page 86). L'interprétation du *bytecode* NEW extrait alors l'information *class_info.declared_instance_size* pour créer un *objet* en mémoire de cette taille. Ce qui correspond à une seule lecture de méta-données en dehors de la plage d'adresses en cours de constitution.

3. **Plusieurs lectures** : ce cas couvre par exemple les *bytecodes* d'appels de méthodes. Comme présentée pour un INVOKEVIRTUAL, la résolution de l'opérande doit d'abord passer par la classe pour scanner la table des méthodes, puis atteindre une structure concrète de type *method_info* et finalement placer cette méthode sur la pile d'exécution. Ce placement requiert lui aussi plusieurs lectures de champs dans la *method_info*, s'ajoutant aux deux indirections précédentes.

Dans l'hypothèse qui est la notre où le fichier CAP a été optimisé, alors les *bytecodes* du cas n°1 ne sont plus des consommateurs de méta-données et n'ont aucune incidence sur l'analyse. Les *bytecodes* du cas n°3 deviennent des rupteurs de flots, si tel n'est pas déjà le cas. Dans le sens où ils rompent le flot « exécutable » de la plage d'adresses en RAM à l'intérieur de l'espace de cache.

Le cas n°2 se décompose quant à lui en deux sous-ensembles. La décomposition est réalisable par un test vérifiant si la méta-donnée à accéder est présente en cache. Si tel est le cas, l'analyse peut continuer car la non-modification du cache est garantie par l'absence de défaut de cache lors de l'exécution de ce *bytecode* par l'interpréteur concret (cas 2.1). À l'inverse (cas 2.2), si la méta-donnée est manquante, alors le *bytecode* doit être considéré comme un rupteur de flot, pour les mêmes raisons que dans le cas n°3.

Le cas 2.1 est intéressant car il permet de valider que la lecture d'une méta-donnée ne provoque pas de défaut de cache et permet donc à l'analyse de continuer à produire une plage d'adresses physiques plus large. De plus, dans ce cas, pour faciliter le travail de l'interpréteur concret, la méta-donnée testée et lue peut être mémorisée et ainsi éviter un accès au cache dans le *handler* correspondant La première méta-donnée qui est lue par le cas n°3 peut être mémorisée de la même manière. Ces actions participent alors à la

les bytecodes de contrôle dynamique de typage (CHECKCAST, INSTANCEOF)

maîtrise du coût global de la pré-interprétation en évitant d'introduire des redondances avec l'interpréteur concret.

Pour résumer, la pré-interprétation doit donc évaluer les opérandes de certains *bytecodes*. La détection de branchements ne peut donc pas être booléenne. Il faut en plus qu'elle interprète le type de branchement pour catégoriser ceux-ci dans les 3 cas de figures (*e.g.* 2.1, 2.2, et 3) de gestion des méta-données, en plus du cas des branchements conditionnels et inconditionnels.

6.3.4 Pilotage actif du cache

Nous abordons maintenant les incidences et actions de notre approche sur le gestionnaire de cache.

6.3.4.1 Forcer la politique de remplacement

La force de LRU réside dans le maintien perpétuel d'une liste triée dans l'ordre des accès, du plus récent au plus ancien. Contourner l'interface logicielle du cache empêche le déclenchement automatique de l'algorithme de gestion des *Hit*. Cependant, la liste LRU ordonne des pages et non des instructions. Par conséquent, cela ne pose aucun problème pour l'accès aux *bytecodes* décodés puisque la pré-interprétation reste par construction dans la même page.

Le problème se pose pour l'accès aux méta-données qui sont éventuellement dans une page différente. Il est impératif de garder la cohérence de la liste en reproduisant les mouvements qu'aurait générés l'algorithme LRU pour un ensemble de *Hit*. Car ces mouvements ne sont en effet pas anodins. Bouger une page de la fin de la liste vers la tête, engage cette page à rester durablement dans le cache. Dans le cas inverse, cette dernière page resterait la candidate déclarée à l'éviction et modifierait le rapport qu'entretient le cache avec le principe de localité temporel porté par LRU.

Toutefois, les modifications apportées par LRU peuvent être compactées en un seul mouvement. Pour le cas 2.1 des lectures de méta-données, LRU aurait procédé à 2 mouvements. Le premier pour déplacer la page contenant la méta-donnée de là où elle est vers la tête, et le second pour la remplacer à nouveau par la page de code courante. Ce qui revient à positionner la page de méta-donnée directement en deuxième position.

Pour le cas n°3, la solution est plus simple et place la page de méta-données en tête de liste. L'anticiper immédiatement est important car nous avons vu que 90 % des accès au cache se faisait dans la page de tête. Si cette tâche est laissée au *handler* du rupteur de flot, celui-ci devra passer par l'interface classique du cache, tâche finalement plus lente du fait de la ré-exécution de l'étape de recherche que nous avons déjà réalisée implicitement lors du pré-décodage.

Ces actions participent elles-aussi à la maîtrise du coût global de la pré-interprétation.

6.3.4.2 Pré-chargements

La pré-interprétation permet d'identifier à l'avance que certaines méta-données sont manquantes dans le cache, et permet d'envisager un pré-chargement de celles-ci. Cette éventualité est le résultat du pré-décodage de l'opérande des rupteurs de flot. Rappelons que dans le cas 2.1, l'opérande est forcement présente en cache et le *bytecode* n'est donc pas considéré comme un rupteur de flot. Cela signifie que l'éventualité d'un pré-chargement ne peut effectivement avoir lieu que lors de la pré-interprétation d'un vrai rupteur de flot.

Le pré-chargement est déclenchable avant le lancement de l'interprétation concrète. Ce qui laisse un certain temps durant lequel deux opérations évoluent en parallèle. Le résultat est alors un gain de temps par recouvrement d'opération.

La pré-interprétation permet également de détecter certains *Miss* sur du code. Par exemple, si le branchement est de type inconditionnel (*i.e. goto*), alors la certitude qu'il fournit sur l'adresse du bloc de base suivant, rend également certaine l'opportunité d'un pré-chargement s'il s'avère absent du cache.

Toutefois, le pré-chargement implique plusieurs contraintes.

Tout d'abord, il faut que techniquement la Flash NAND supporte les lectures asynchrones. Ce qui n'est malheureusement pas toujours le cas.

Ensuite, il faut que le pré-interpréteur soit adapté pour gérer à la fois les soumissions mais surtout valider la réception de données. En effet, le temps de recouvrement est de durée variable du côté de l'interpréteur concret, suivant les types de *bytecodes* qu'il exécute. Cela signifie que lorsqu'il arrive au moment de l'exécution du rupteur de flot, l'interpréteur concret doit parfois se mettre en attente du pré-chargement. Pour ne pas tester cet état d'attente entre chaque exécution du Listing 6.3, le test de fin de pré-chargement est délégué uniquement au *handler* susceptible de les subir.

Enfin, le coût de contrôle d'un pré-chargement possible ne doit pas détériorer le coût de la pré-interprétation. Comme discuté section 3.3.2, page 32, l'insertion d'un point de pré-chargement n'implique pas nécessairement qu'un pré-chargement soit effectivement soumissible. Malgré l'intérêt qu'il peut porter, le pré-chargement doit donc être considéré comme une opportunité optionnelle qu'il convient d'évaluer avant de la mettre en œuvre. Comme le nombre de pré-chargement est lié au contenu du cache, et donc au nombre de défaut de cache, notre approche peut donc se trouver ralentie par ce composant si le cache a un taux de *Hit* suffisamment élevé.

6.3.5 Synthèse

Nous avons abordé dans cette section et la précédente, les points qui guident la conception d'un outil de pré-interprétation du code, dont le but est la réduction drastique du nombre d'accès à un cache d'instructions déjà existant. Les points que nous avons présenté permettent d'aborder une construction fiable et efficace de cet outil. Il s'agit donc maintenant d'évaluer ces points dans leur contexte pour apporter des précisions sur leur réelle performance.

La mise-en-œuvre efficace d'une pré-interprétation et des optimisations qu'elle offre, reste tributaire d'un facteur dominant en terme d'efficacité et de bénéfice. Ce facteur est le rapport entre le gain qu'elle apporte et le coût supplémentaire qu'elle ajoute à la traversée de la pile d'accès à la mémoire non-adressable, telle qu'illustrée dans la figure 6.2c, page 106. C'est par cet angle de vue, que nous abordons maintenant l'évaluation de notre approche.

6.4 Confrontation du gain et du coût

Dans cette section, nous proposons une série d'expériences dont le but est d'évaluer puis confirmer les bénéfices de notre approche en rupture avec l'état de l'art sur les caches. Cette section est donc l'occasion de confronter le gain et le coût d'une pré-interprétation du code puisque c'est ce rapport qui conditionne le succès et la performance ajoutée de notre proposition.

6.4.1 Maîtrise du coût d'exécution

L'approche que nous proposons repose sur une analyse du code qui sera très prochainement exécuté, sous la forme d'une pré-interprétation. Du fait de cette position, l'analyseur est le principal centre de coût et de gain de la pré-interprétation.

Le rapport gain/coût s'articule autour du nombre d'analyses effectuées et du nombre d'accès au cache évités. Formellement, ce rapport implique un seuil de rentabilité qui se caractérise par l'inéquation suivante :

Inéquation 3. *Seuil de rentabilité de l'analyse de code*

$$\begin{cases} Coût_{Analyses} & < & Coût_{Accès-évités} \\ Coût_{Analyses} & = & T_{Analyse} * N_{Analyse} + \Delta \\ Coût_{Accèsévités} & = & N_{Accès} * (T_{Recherche} + T_{GestionHit}) \end{cases}$$

Le nombre d'analyses $N_{Analyse}$ n'étant pas égal au nombre d'accès au cache $N_{Accès-évités}$, la rentabilité doit être calculée en fonction de coûts globaux. De plus, le coût des analyses dépend également d'un coût Δ, variable par analyse et représentant l'appel hypothétique à des pré-chargements de pages ainsi que l'échange des pré-décodages de quantités variables avec l'interpréteur concret.

6.4.2 Évaluation expérimentale de la pré-interprétation

La vocation première de notre pré-interprétation est d'identifier une plage d'adresses continues dans l'espace de cache pour éliminer un certain nombre d'accès à ce dernier par son interface logicielle. D'un point de vue opérationnel, on peut dire que le pré-interpréteur est ainsi en quelque sorte une autre interface logicielle d'accès au cache. Par cet aspect, un certain nombre d'accès au cache via une interface classique par octets sont remplacés par des accès via une autre interface par bloc.

On souhaite donc essentiellement comparer le coût que représentaient auparavant les accès évités par la pré-interprétation, avec le coût de l'analyse de code, de son déclenchement jusqu'au moment où elle finalise artificiellement les mêmes groupes d'informations. Pour cela, nous reprenons les cas d'étude des chapitres précédents (voir page 41), qui nous ont permis de formuler les constats qui nous ont conduit jusqu'à cette proposition.

6.4.2.1 Protocole expérimental

Pour mesurer les bénéfices de la pré-interprétation, nous proposons une évaluation en deux temps. Nous commençons en environnement simulé pour pouvoir évaluer des programmes en comparaison avec les chapitres précédents, notamment avec la suite Richards, et en travaillant sur le même mode opératoire. Nous y ajoutons l'application JCProfil, pour les raisons décrites section 3.4.2, page 40. Nous mettons ensuite en pratique la pré-interprétation dans un environnement matériel complet et similaire à celui disponible dans une carte à puce JavaCard 2.2 typique.

Notre évaluation porte principalement sur les deux éléments technologiques utilisés dans notre approche, *e.g.* la pré-interprétation et le pré-chargement. Nous vérifions donc dans une premier temps l'utilité du premier élément par une mesure du coût de l'analyse de code par rapport au coût des accès au cache qu'elle doit éviter. Ceci fait, nous vérifions ensuite l'opportunité de greffer un mécanisme de pré-chargement de pages à la pré-interprétation grâce à un pré-décodage de méta-données.

Les résultats produits en environnement simulé par lequel nous commencerons sont toujours basés sur un calcul du coût en instructions (CoI). Pour rappel, le CoI n'est pas un nombre de cycles CPU et ne donne pas une mesure exacte en temps mais propose un ordre de grandeur vérifiable et indépendant de la fréquence du CPU. Ce mode opératoire nous permet de plus d'évaluer des programmes comme la suite Richards qui reste démesurés pour une carte à puce, mais qui sont toujours utiles en tant que points de comparaisons.

6.4.2.2 Évaluation de la pré-interprétation

Le premier indicateur du bénéfice apporté par la pré-interprétation est le pourcentage d'accès évités reproduit dans la première colonne du tableau 6.3, page 120. On constate que ce taux peut être très élevé, pour dépasser les 80 %. Ce taux est lié à la taille moyenne d'un bloc de base puisque les accès évités le sont à l'intérieur d'un bloc. Il est également inversement lié au volume de méta-données lues en dehors de ces blocs de base. C'est pourquoi, des programmes comme les versions 5 ou 7 de la suite Richards offrent un moins bon taux, car ceux-ci cumulent des blocs de base plus petits (voir colonne 3 et 4) et plus fréquemment entre-coupés par des méta-données dont le volume est globalement conséquent (voir chapitre précédent).

Le graphique 6.8 illustre la comparairon entre le CoI des accès évités (en bleu) et le CoI des différentes analyses de la pré-interprétation (en orange). Dans ces résultats, données en millions d'IOv, chaque programme n'est évalué que sur les 2 premiers millions de *bytecodes* exécutés par la JCVM pour qu'ils puissent être facilement comparés.

Le gain entre les deux approches évaluées est donc assez élevé, et varie selon les programmes de 58 à 70 %. Ce qui représente effectivement un gain significatif de la pré-interprétation sur l'approche plus classique. Cependant, ce gain n'est pas proportionnel au taux d'accès évités. Pour éclaircir ce point, nous pouvons regarder les colonnes 5 et 6 du tableau 6.3 qui reportent le coût moyen d'un accès au cache et le coût moyen d'une analyse.

Ces coûts varient selon le contenu du cache. Plus le IOv/accès est bas, plus le nombre d'accès au cache correspond à des accès à la tête de liste LRU. Au contraire, le temps de recherche dans le cache augmente lorsque le brassage des entrées LRU est très fréquent. C'est le cas notamment lorsque les distances et les longueurs de chemins entre méta-données sont importantes. À l'inverse, plus le bloc de base est petit, plus l'analyse se termine rapidement.

C'est pourquoi le gain pour la version 7 par exemple est beaucoup plus important car il cumule des analyses plus nombreuses mais plus rapides qui contre-balancent des accès au cache globalement plus coûteux, car allongés par des recherches en cache plus profondes.

Au final, le coût de la pré-interprétation devient comparable entre les versions 1 et 7, alors que le CoI des accès de la version 7 était beaucoup plus élevé. Notre approche permet donc de gommer en grande partie l'impact du modèle de méta-données sur les accès au cache. Cependant, il faut noter que dans le cas de la version 7, il reste encore 40 % de *Hits* à effectuer par le canal normal, contre 17 % à la version 1. Donc, même dans le cas où les versions 1 et 7 exécutent le même nombre de *bytecodes*, notre approche ne fait bien évidemment pas disparaitre l'écart de performances entre versions mais le comble fortement, par une meilleure méthode d'accès global aux données du cache.

6.4.2.3 Évaluation du pré-chargement

Critères d'évaluation La mise en œuvre d'un mécanisme de pré-chargement est à la base essentiellement conditionnée par la capacité du matériel à la supporter. Cette condition est la prise en charge d'un certain niveau d'asynchronie de la part du contrôleur de la Flash série ou le support matériel d'une DMA (*Direct Memory Access*).

	Pourcentage d'accès évités	Nombre moyen d'accès évités par une analyse	Nombre moyen de bytecodes décodés par une analyse	IOv moyen par accès	IOv moyen par analyse
JCProfil	82,87 %	6,43	3,25	66,87	185,84
RichV1	82,78 %	5,54	2,70	66,38	151,45
RichV2	83,02 %	5,78	2,81	65,17	148,69
RichV3	83,51 %	5,34	2,66	66,79	151,94
RichV4	71,60 %	5,13	2,35	68,81	138,84
RichV5	57,29 %	5,58	2,14	71,26	128,90
RichV7	59,37 %	5,75	2,14	72,04	125,75

Tableau 6.3: Indicateurs intermédiares pour la mesure du gain de la pré-interprétation

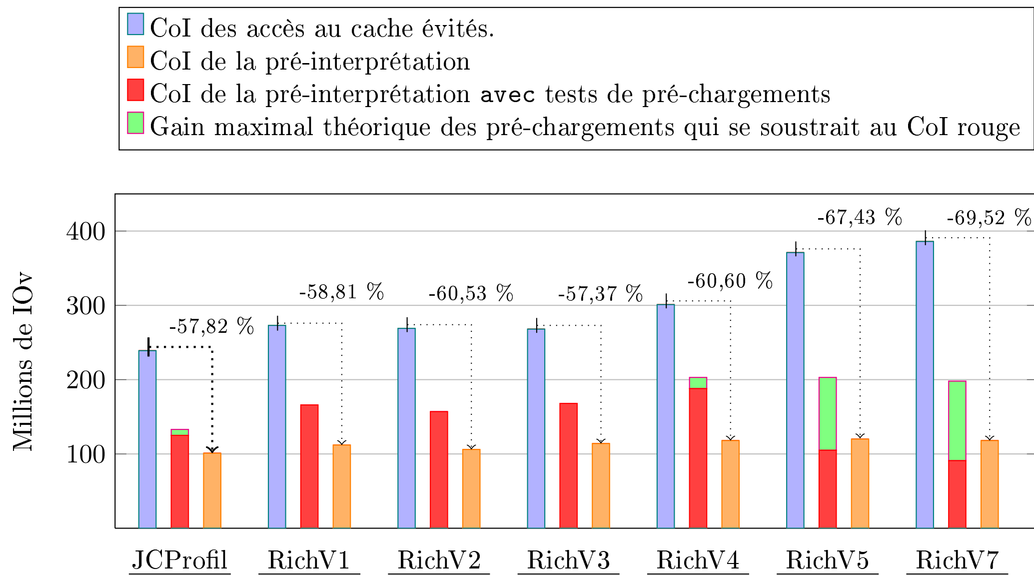


Figure 6.8: Comparaison du coût de la pré-interprétation
et du coût des accès au cache qu'elle évite au dessus d'un cache logiciel de 1024 octets.

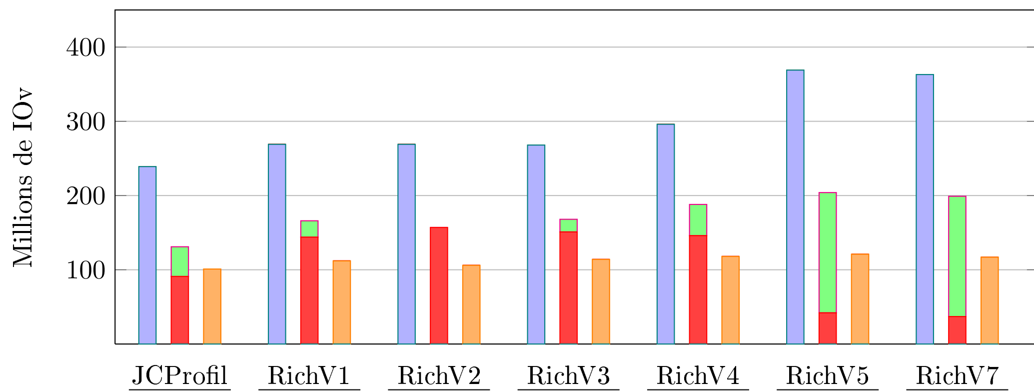


Figure 6.9: Comparaison du coût de la pré-interprétation
et du coût des accès au cache qu'elle évite au dessus d'un cache logiciel de 512 octets.

Malheureusement notre matériel à disposition ne permet pas de supporter cette approche. Néanmoins, il reste intéressant de mesurer le bénéfice théorique des pré-chargements pour déterminer si une modification du matériel serait rentable ou non.

Le mécanisme de pré-chargements se décompose en deux étapes et donc deux points de mesure. La première mesure regroupe les tests à effectuer pour déterminer si un pré-chargement est nécessaire. La seconde mesure consiste à mesurer le temps de recouvrement, c'est à dire à multiplier le nombre de défauts de cache déclenchés à l'avance par le temps de chargement du registre Flash. Au final, pour évaluer la rentabilité par ces deux étapes, nous mesurons donc le coût des tests auquel nous soustrayons le gain du recouvrement. Si ce nombre est toujours positif, alors les pré-chargements coûtent plus cher que ce qu'ils ne rapportent.

Mesure du gain maximal théorique Les CoI des tests de pré-chargement sont reproduits dans le Graphique 6.8, page 120, par les barres rouges. Sur ces barres, le gain apporté par les recouvrements d'opérations est matérialisé par les encarts en vert qui sont à décompter du total. Lorsqu'il n'apparait pas distinctement, cela signifie que ce gain est infime.

Ce gain en vert est présenté sous forme d'un gain maximal théorique. Le temps de latence de la Flash étant calculé en nombre de cycles, alors que nos graphiques sont en nombre d'instructions, nous comptons ici une instruction par cycle. Ce qui correspond au nombre maximal d'instructions qui peuvent être exécutées en parallèle du chargement du registre Flash, qui dure environ 145 μs. Dans les faits, ce maximum n'est jamais atteint pour deux raisons. Tout d'abord, ces instructions ne peuvent pas toutes correspondre chacune à un seul cycle. Et deuxièmement, il n'est pas toujours possible d'effectuer un recouvrement complet. En effet, entre le déclenchement du pré-chargement et sa fin, la JCVM peut avoir plus ou moins d'instructions à exécuter durant ce temps imparti, selon l'état dans lequel elle se trouve.

Évaluation de la rentabilité des pré-chargements Les pré-chargements sont directement liés aux défauts de cache « normaux » , puisqu'ils en sont des remplacements anticipés. Il ne peut donc pas y avoir plus de pré-chargements que de défauts de cache. Dans ces conditions, si le ratio de *Miss* est très bas, alors le nombre potentiel de pré-chargements l'est aussi.

Les tableaux 6.4 et 6.5 présentent pour l'application JCProfil le nombre de tests qu'il faut réaliser pour détecter un pré-chargement, le nombre de pré-chargements soumissibles et la proportion qu'ils représentent par rapport à tous les défauts de cache que génère cette application. Ces tableaux donnent des résultats pour des configurations de cache (colonne 1 et 2) qui ne sont pas toutes performantes dans l'absolu (cf chapitre 4). Cependant, elles sont données à titre d'exemple pour montrer le lien entre l'augmentation du nombre de *Miss* et la rentabilité des pré-chargements.

Malheureusement, du fait de notre matériel, nous ne pouvons pas donner dans ce document le gain réel des pré-chargement par rapport au gain théorique illustré dans le Graphique 6.8, page 120. Nous pouvons seulement donner une estimation se basant sur un profilage du temps d'exécution des principaux types de *bytecodes*.

Les *bytecodes* les plus longs à exécuter sont les appels de méthodes. Cependant, ceux-ci n'entrent jamais dans la catégorie des *bytecodes* recouvrables par des pré-chargements. Ils font en effet partis des *bytecodes* en attente de pages pré-chargées. Les autres *bytecodes* oscillent entre un temps d'éxecution de $20\mu s$ et $70\mu s$. Le temps réellement recouvert entre le pré-chargement et l'interpréteur concret dépend du nombre de *bytecodes* par blocs de base et de leur type.

Nombre de pages	Taille de pages	Nombre de tests	Nombre de pré-chargement	Nombre de *Miss*	Ratio / tests	Ratio / *Miss*
2	512	258 078	17 944	83 969	6,95%	22,42%
4	256	258 078	17 135	71 555	6,64%	23,95%
8	128	258 042	4 055	7 499	1,57%	54,07%
16	64	256 278	5 978	8 081	2,33%	73,97%
32	32	246 862	17 153	23 147	6,95%	74,10%

Tableau 6.4: Statistiques de pré-chargements pour un cache de 1024 octets.

Nombre de pages	Taille de pages	Nombre de tests	Nombre de pré-chargement	Nombre de *Miss*	Ratio / tests	Ratio / *Miss*
2	256	258 078	21 235	192 428	8,22 %	11,03 %
4	128	258 043	22 266	87 301	8.55%	25,27%
8	64	256 278	20 362	70 607	7,95 %	28.83%
16	32	246 862	28 308	92 477	11,47%	30,6%

Tableau 6.5: Statistiques de pré-chargements pour un cache de 512 octets.

Par extrapolation sur des données optimistes, si la taille moyenne d'un bloc est de 4, avec des *bytecodes* « rapides », il est possible d'espérer dans ce cas $80\mu s$ de recouvrement sur les $145\mu s$. Ce qui placerait la pré-interprétation avec et sans pré-chargement à peu près sur le même plan de performance dans la plupart des cas, comme le montre le Graphique 6.8, page 120.

Pour autant, cela ne signifie pas que les pré-chargements ne servent à rien. Ils ont en effet trois interêts majeurs. Le premier est de proposer une solution globale plus souple qui puissent s'adapter à un grand nombre d'applications téléchargeables, aux profils peut-être différents. Le second est d'accélérer certains moments de l'exécution pour niveller l'impact momentané de certains défauts de cache, et notamment lors de la mise-sous-tension de la carte lorsque le cache est vide. Enfin, le dernier intérêt est d'apporter une solution élégante pour améliorer les performances de cas extrêmes où le cache doit être très petit avec alors un nombre très élevé de *Miss*. Tout dépend donc du contexte d'utilisation industrielle.

6.4.3 Preuve de concept

Nous proposons maintenant une mise en pratique de notre approche sur du matériel typique JavaCard. Par cette preuve de concept, nous confirmons ainsi les bénéfices de notre approche en vérifiant qu'elle passe réellement à l'échelle des cartes à puce.

Support matériel : Le matériel sur lequel nous travaillons est basé sur un processeur ARM7 32 bits cadencé à 17.5 MHz. Ce processeur est doté de 32 Ko de ROM, 48 Ko de RAM et 768 Ko de Flash NOR adressable. Le micro-contrôleur quant à lui supporte les deux protocoles de transfert T=0 et T=1 de la norme ISO 7816-3 (voir page 7). Il propose également une interface d'accès à de la mémoire Flash sur un bus série, dont la Flash NAND. Celle que nous utilisons a les propriétés de la Flash NAND décrite page 28, avec des pages de 2048 octets.

Protocole expérimental : Notre preuve de concept utilise l'API JavaCard standard et une JCVM conforme aux spécifications [JCVM.2.2.1 2003, JCRE.2.2.1 2003]. Chaque test est réalisé unitairement sur l'application JCProfil. Celle-ci est d'abord téléchargée

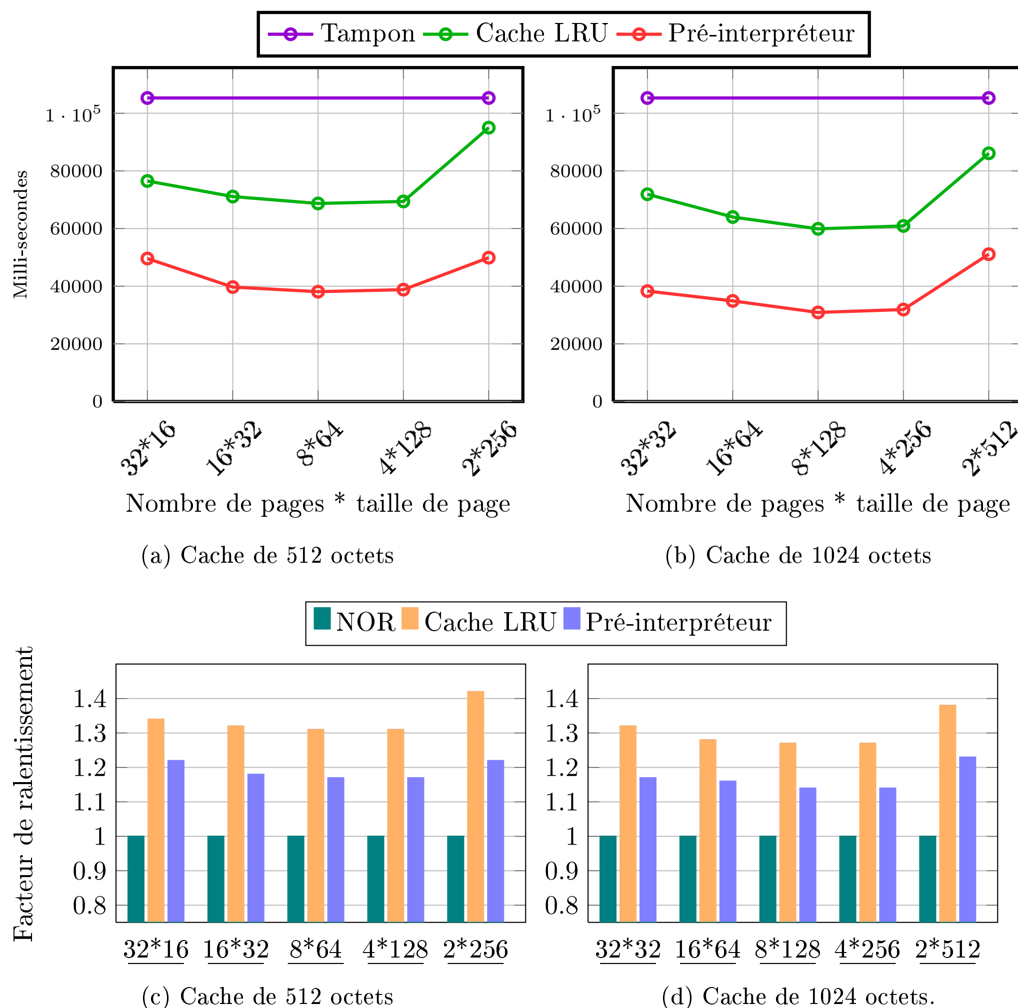


Figure 6.10: Comparaison entre la NOR, un tampon, un cache LRU et un pré-interpréteur dans une carte à puce à 17.5 Mhz.

sur la carte puis vérifiée et installée. Enfin, la carte reçoit une APDU lui demandant de lancer l'exécution de l'application. La prise de mesure commence à cet instant. Le temps d'exécution final est obtenu à la centaine de milli-secondes près par un point d'arrêt dans un débogueur matériel.

Notre base de comparaison est le cas où l'application JavaCard JCProfil est stockée dans la Flash NOR, et exécutée en place depuis celle-ci. Nous lui confrontons d'abord un stockage en Flash NAND et une exécution en place à l'aide d'un tampon de 2048 octets. Cette deuxième mesure forme la borne supérieure de notre preuve de concept. L'idée principale est donc de mesurer d'une part le facteur de ralentissement par rapport à la Flash NOR, puis d'autre part de mesurer comment les diverses approches présentées dans notre état de l'art puis la pré-interprétation réduisent la latence pour l'exécution en place depuis la Flash NAND.

Performances comparées : Les performances comparées de la Flash NOR, d'un tampon, d'un cache et de notre approche par pré-interprétation sont illustrées dans la fig-

ure 6.10, page 123. Dans les graphiques 6.10a et 6.10b, chaque pile d'accès (voir page 106) est évaluée sur un critère de temps, relatif à la Flash NOR. Les données présentées sont le temps supplémentaire en milli-secondes de chaque cas de test par rapport à la Flash NOR. Dans ces mesures, on constate qu'un cache LRU (en vert) est effectivement plus performant qu'un tampon (en mauve). Cependant, il se dégage clairement que notre approche par pré-interprétation et accès groupés au cache (en rouge) est bien meilleure encore, avec un gain oscillant autour de 50 % par rapport à un cache classique. Il est intéressant également de noter que notre approche sur un cache de 512 octets (Fig. 6.10a) offre de meilleures performances qu'un cache classique de 1024 octets (Fig. 6.10b). Ce qui confirme que notre approche permet également de réduire l'empreinte mémoire du cache car le gain total permet également de contre-balancer plus largement le temps cumulé des défauts de cache entre ces deux configurations.

Les graphiques 6.10c et 6.10d présentent une autre vue des résultats en fournissant cette fois le facteur de ralentissement de chaque pile d'accès à la Flash NAND par rapport au cas de base que représente la Flash NOR. Dans ces graphiques, la Flash NOR est normalisée à un. Les autres barres représente le coefficient multiplicateur qui donne la dégradation de la performance à stocker les applications JavaCard dans le Flash NAND.

Nous vérifions maintenant le critère de temps de la norme IOS-7816-3 qui demande pour rappel qu'une transaction dure entre 3 et 5 secondes. Si une telle transaction dure par exemple habituellement 4 secondes, avec un tampon elle en durera 5,88, avec un cache 5,2, et avec notre approche moins de 4,55 secondes. C'est à dire un ralentissement quasi-transparent du point de vue de l'utilisateur et conforme à la norme ISO, ce qui n'est pas le cas des autres approches plus classiques.

6.5 Conclusions

Nous avons présenté dans ce chapitre une nouvelle approche pour optimiser l'exécution de code stocké dans une mémoire non-adressable comme le Flash NAND. Cette approche est une pré-interprétation reposant sur une analyse de code. Cette pré-interprétation permet un pré-décodage d'instructions, comme des *bytecodes*, et de certaines méta-données propres aux langages interprétés comme le Java. Mais son réel atout est surtout le calcul d'une garantie ferme de pouvoir exécuter des groupes d'instructions sans la nécessité d'accéder au cache par son interface logicielle. C'est ce point qui offre alors à un cache logiciel un gain en performance significatif.

Notre preuve de concept démontre que notre approche est judicieuse et performante. Elle confirme également une excellente résistance au passage à l'échelle des cartes à puce et offre des temps de réponse quasi-transparents.

Notre preuve de concept démontre enfin qu'une pré-interprétation permet de réduire encore l'empreinte mémoire du cache avec une faible dégradation des performances comme le montrent les résultats pour un espace de stockage de cache de seulement 512 octets.

Nous avons également montré que si le matériel le supporte, ajouter un mécanisme de pré-chargements pouvait fournir de la souplesse face aux applicatifs inconnus qu'amène la *post-issuance*, sans trop peser sur le coût d'exécution globale du pré-interpréteur.

Chapitre 7

Conclusion

7.1 Synthèse

Dans ce document de thèse, nous avons abordé une problématique d'avenir pour la *post-issuance* qu'est l'exécution d'application depuis une mémoire non-adressable. Nous avons montré, dans notre chapitre 3 de Problématique, que la latence d'une telle mémoire est à la base un réel frein à une utilisation efficace de cette mémoire pour ce genre d'exercice. Nous avons alors abordé le problème en adoptant les apports et réponses de l'état pour les appliquer aux contraintes des cartes à puces comme la nécessité d'une empreinte mémoire faible.

Nous avons ainsi montré qu'une solution à base de cache logiciel reste une approche efficace, tant pour le code compilé que pour le code semi-compilé agrémenté de méta-données comme le langage Java. Nous avons mis en évidence comment certains paramètres du cache permettaient de mieux adapter un cache logiciel à la contrainte d'une empreinte mémoire faible, ce qui fut présenté dans ces papiers [Cogniaux 2010a, Cogniaux 2010b].

Nous avons ensuite réalisé une étude détaillée des méta-données Java/JavaCard. Si le code compilé a beaucoup été étudié pour des problématiques de caches d'instructions, les méta-données Java ont jusque là été peu documentées sur ce point alors qu'elles représentent des données exécutables à part entière. Dans le chapitre 5 et dans la publication suivante [Cogniaux 2011], nous nous sommes attelés à cette tâche en étudiant leurs similarités avec le code dans leurs comportement à l'exécution, ainsi qu'en identifiant leurs différences, notamment l'impact de blocs de données plus petits. Cette étude nous alors permis d'avoir une vision globale pour aborder la problématique de caches de méta-données logiciels, du même point de vue qu'un cache d'instructions logiciel.

Nous avons ainsi montré qu'un cache logiciel permet effectivement de réduire l'écart entre l'exécution en place depuis une mémoire secondaire et l'exécution en place depuis une mémoire adressable comme la Flash NOR. Cependant, nous avons également démontré que l'écart restant est encore une condition insuffisante pour résoudre de manière franche notre problématique, car un cache logiciel possède toujours une latence qui lui est propre.

Pour dépasser cet état de fait, nous avons constaté qu'un cache contenait, à un instant donné, une copie partielle, morcelée et désordonnée du binaire exécutable. Notre intuition a alors été de dire que si l'on regarde un cache en dehors de ces attributions (*e.g.* récupération, placement, renouvellement), nous pouvions remarquer que ce cache rendait disponible en mémoire principale tout ce dont nous avions pour gagner encore en performance : plusieurs blocs de données dans une mémoire `adressables`. De plus, cette intuition est renforcée par le degré de séquentialité de chaque programme qui nous indique une forte probabilité de recherches en cache finalement injustifiées et pénalisantes.

Fort de ce constat, nous avons proposé une nouvelle approche consistant à accéder aux données présentes en cache d'une manière plus directe. Sortant du modèle d'accès au cache octet par octet, nous avons développé une approche permettant au contraire de regrouper les accès, pour isoler une plage d'adresses physiques que nous garantissons sans nécessité d'accès à l'interface logicielle du cache. L'opérateur d'exécution peut alors pendant un laps

travailler en toute quiétude sur ces données isolées avec les même performances que lors de l'exécution en place depuis la Flash NOR. La longueur de cette période d'exécution optimale est directement mesurable par le degré de séquentialité qui nous avons introduit chapitre 4.

Notre approche consiste en une pré-interprétation du code, basée sur une analyse du flot d'exécution. Elle repose sur une scission de l'interpréteur originel en deux phases d'interprétation distinctes : une pré-interprétation manipulant le code et le cache pour préparer une interprétation concrète, différée, et rendue indépendante du gestionnaire de cache. Cette pré-interprétation est la source d'un regain de performance significatif car les informations qu'elle produit deviennent au final largement plus rentables que tous les accès au cache qu'elle évite.

Par ce résultat, il en devient même possible de réduire fortement l'empreinte mémoire de cache sans trop de dégradation de performance. Par ailleurs, nous avons montré par une preuve de concept sur matériel qu'il est de plus remarquable d'avoir par notre approche une performance qui reste supérieure à un système de cache sans pré-interprétation et mieux doté en espace de stockage.

7.2 Perspectives : transposer la pré-interprétation

Notre preuve de concept est une démonstration pour du code JavaCard, la plateforme technologique de *post-issuance* la plus plus répandue dans les cartes à puce. Cependant, elle ouvre la voie à d'autres challenges auxquels nous n'avons pas répondu.

Application au modèle de méta-données Java standard Nous avons vu dans notre chapitre 5, qu'un modèle de méta-données trop lourd pénalisait fortement la mise-en-cache d'application Java pour les exécuter depuis une mémoire non-adressable. Nous avons ainsi constaté que le modèle de méta-données de JavaCard 2.2 était moins pénalisant grâce à sa compacité.

Ce modèle est obtenu hors-ligne par un outil de conversion qui permet également d'obtenir un fichier applicatif (le fichier CAP) beaucoup plus léger que les fichiers de classes standards. Notre étude montre alors que des JVM pour systèmes embarqués basées sur du Java standard gagne à utiliser l'approche JavaCard 2.2 pour construire un modèle de méta-données plus compacte. Mettre en œuvre notre approche par pré-interprétation permet d'envisager pourquoi cette conversion au fil de l'eau en greffant à notre approche une sorte de convertisseur/compacteur à la volée. Offrant ainsi la possibilité de continuer à déployer des fichiers de classes standards, par exemple JavaCard 3 sur des cartes à puce, ou J2ME pour KVM.

D'autre part, notre approche de part sa conception pourrait très bien se prêter à la compilation de code Java/JavaCard à la volée. Au moins sur des compilations partielles et simples, limitées aux blocs de base identifiés par le pré-décodage.

Application au code compilé Le coût d'un accès au cache est y encore plus problématique car le handicap d'un cache d'instructions implémenté en logiciel face à un processeur est beaucoup plus important qu'avec une VM. Notre approche par accès groupés est alors d'autant plus bénéfique que l'écart entre l'exécution en place à l'aide d'un cache et l'exécution en place depuis la NOR est important.

Cependant, notre approche a une limite qui est de rendre exécutable le code pré-décodé sans outil de management comme une machine virtuelle. Dans ce cadre, une partie de réécriture du code natif est nécessaire pour la gestion du pointeur d'instruction, pour d'une

part contrôler les changements de contexte entre blocs de base et d'autre part encadrer les rupteurs de flot. Ces deux points sont incontournables pour intercaler entre ses flots d'exécution une pré-interprétation du code natif à des fins d'isolation de blocs de code. Des travaux portant sur la génération de code à la volée [Grimaud 1999], des mécanismes simples de mémoire virtuelles embarquées [Gu 2006], ou des changements de contextes managés par des co-routines [Duquennoy 2010], sont des pistes intéressantes pour adresser ce genre de problème malgré les contraintes physique des cartes à puces. Adapter ces différentes approches pour outiller la notre produirait un système permettant d'attendre les performances suffisantes pour mettre en pratique notre approche sur du code compilé.

Et si c'était à refaire...

Exécuter du code depuis une mémoire non-adressable. Pour ce sujet de réflexion, un cache est sans équivoque LA solution évidente. Tellement évidence qu'elle ne peut pas ne pas être étudiée, surtout si c'est pour l'utiliser dans un cas de figure qui lui est peu courant comme dans les contraintes d'une carte à puce. Une telle évidence assure également qu'il existe au moins une source de repère et de point de comparaison avec toutes autres solutions qui pourraient être proposées. De plus, derrière le cache au sens large, se trouvent d'autres évidences portées par l'état de l'art qui institue par exemple la taille de l'espace de stockage ou les politiques de remplacement comme principales sources de performance. Voire même toujours une source de gain potentiel, que le domaine de recherche concerné n'a peut-être pas encore atteint, s'il se compare à MIN. Ainsi, une longue partie du temps consacré à cette thèse s'est focalisée sur cette approche, que l'état de l'art et le bon sens rendent si évidente, pour preuve la place que son étude occupe dans ce document.

Néanmoins, notre nouvelle approche et ses résultats montrent qu'une autre voie est possible dans la manière d'aborder l'accès aux données qu'un cache met à disposition. Nos chapitres 4 et 5 sont un travail de fondation sur un large spectre de type de données exécutable à mettre en cache, du code natif au code interprété. Si le temps d'une thèse était à nouveau consacré au même sujet, sur base de ce document, il n'est à pas douté que des solutions encore plus élégantes puissent apparaitre. Peut-être passeraient-elles par une toute nouvelle manière de concevoir une JVM pour cartes à puce ou pour systèmes embarqués contraints, orientée composants temporairement relogeables d'une mémoire à une autre, au gré des besoins. Peut-être passeraient-elles par le stockage d'une version allégée d'une application en NOR et qui se serait mise à jour de manière incrémentale avec ses bouts de codes moins usités stockés en NAND, plutôt que d'utiliser un cache qui a oublié (ou n'a jamais su) ce qu'est une application. Peut-être...

Pour terminer en citant Stanislaw J. Lec, « il n'est point d'impasse là où on peut faire marche arrière ».

Bibliographie

[Adl-Tabatabai 1998] Ali-Reza Adl-Tabatabai, MichałCierniak, Guei-Yuan Lueh, Vishesh M. Parikh and James M. Stichnoth. *Fast, effective code generation in a just-in-time Java compiler.* SIGPLAN Not., vol. 33, no. 5, pages 280–290, May 1998. 15

[Al-Zoubi 2004] Hussein Al-Zoubi, Aleksandar Milenkovic and Milena Milenkovic. *Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite.* In Proceedings of the 42nd annual Southeast regional conference, ACM-SE 42, pages 267–272, New York, NY, USA, 2004. ACM. 24

[Angiolini 2004] Federico Angiolini, Francesco Menichelli, Alberto Ferrero, Luca Benini and Mauro Olivieri. *A post-compiler approach to scratchpad mapping of code.* In Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems, CASES '04, pages 259–267, New York, NY, USA, 2004. ACM. 22, 30

[Arnold 2000] Matthew Arnold, Stephen Fink, David Grove, Michael Hind and Peter F. Sweeney. *Adaptive optimization in the Jalapeno JVM.* SIGPLAN Not., vol. 35, no. 10, pages 47–65, October 2000. 15

[Baiocchi 2011] José A. Baiocchi and Bruce R. Childers. *Demand Code Paging for NAND Flash in MMU-less Embedded Systems.* In Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE), 2011, 2011. 25

[Bala 2000] Vasanth Bala, Evelyn Duesterwald and Sanjeev Banerjia. *Dynamo: a transparent dynamic optimization system.* SIGPLAN Not., vol. 35, no. 5, pages 1–12, May 2000. 22

[Banakar 2002] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan and Peter Marwedel. *Scratchpad memory: design alternative for cache on-chip memory in embedded systems.* In Proceedings of the tenth international symposium on Hardware/software codesign, CODES '02, pages 73–78, New York, NY, USA, 2002. ACM. 22, 30

[Bays 1977] Carter Bays. *A comparison of next-fit, first-fit, and best-fit.* Commun. ACM, vol. 20, no. 3, pages 191–192, March 1977. 22

[Belady 1966] L. A. Belady. *A study of replacement algorithms for a virtual-storage computer.* IBM Syst. J., vol. 5, no. 2, pages 78–101, June 1966. 24

[Bell 1973] James R. Bell. *Threaded code.* Commun. ACM, vol. 16, no. 6, pages 370–372, June 1973. 112

[Bernardeschi 2008] C. Bernardeschi, N. De Francesco, G. Lettieri, L. Martini and P. Masci. *Decomposing* bytecode *verification by abstract interpretation.* ACM Trans. Program. Lang. Syst., vol. 31, no. 1, pages 3:1–3:63, December 2008. 11

[Blackburn 2006] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee,

J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage and Ben Wiedermann. *The DaCapo benchmarks: java benchmarking development and analysis.* SIGPLAN Not., vol. 41, no. 10, pages 169–190, October 2006. 39

[Bloch 2008] Joshua Bloch. Effective java: a programming language guide; 2nd ed. Addison-Wesley, 2008. 40

[Brouwers 2009] Niels Brouwers, Koen Langendoen and Peter Corke. *Darjeeling, a feature-rich VM for the resource poor.* In Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems, SenSys '09, pages 169–182, New York, NY, USA, 2009. ACM. 79

[Bull 2000] Mark Bull, Lorna Smith, Martin Westhead, David Henty and Robert Davey. *Benchmarking Java Grande Applications.* 2000. 39

[Callahan 1991] David Callahan, Ken Kennedy and Allan Porterfield. *Software prefetching.* SIGOPS Oper. Syst. Rev., vol. 25, no. Special Issue, pages 40–52, April 1991. 32

[Carr 1981] Richard W. Carr and John L. Hennessy. *WSCLOCK a simple and effective algorithm for virtual memory management.* SIGOPS Oper. Syst. Rev., vol. 15, no. 5, pages 87–95, December 1981. 24

[Chang 2010] Yuan-Hao Chang, Jian-Hong Lin, Jen-Wei Hsieh and Tei-Wei Kuo. *A strategy to emulate NOR flash with NAND flash.* Trans. Storage, vol. 6, no. 2, pages 5:1–5:23, July 2010. 25

[Chen 2003] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske and M. Wolczko. *Heap compression for memory-constrained Java environments.* SIGPLAN Not., vol. 38, no. 11, pages 282–301, October 2003. 22

[Chilimbi 2002] Trishul M. Chilimbi and Martin Hirzel. *Dynamic hot data stream prefetching for general-purpose programs.* SIGPLAN Not., vol. 37, no. 5, pages 199–209, May 2002. 34

[Chung 2006] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee and Ha-Joo Song. *System Software for Flash Memory: A Survey.* In Embedded and Ubiquitous Computing, volume 4096 of *Lecture Notes in Computer Science*, pages 394–404. Springer Berlin Heidelberg, 2006. 7

[Cogniaux 2010a] Geoffroy Cogniaux and Gilles Grimaud. *Impact of Pages Sizes to Execute Code Using Demand Paging and NAND Flash at Smart Card Scale.* In Proceedings of the 2010 International Conference on Complex, Intelligent and Software Intensive Systems, CISIS '10, pages 794–799, Washington, DC, USA, 2010. IEEE Computer Society. 125

[Cogniaux 2010b] Geoffroy Cogniaux and Gilles Grimaud. *Key-Study to Execute Code Using Demand Paging and NAND Flash at Smart Card Scale.* In Dieter Gollmann, Jean-Louis Lanet and Julien Iguchi-Cartigny, editeurs, Smart Card Research and Advanced Application, volume 6035 of *Lecture Notes in Computer Science*, pages 102–117. 2010. 125

[Cogniaux 2011] Geoffroy Cogniaux, Michaël Hauspie and François-Xavier Marseille. *Étude Préliminaire À Une Utilisation De Mémoires Secondaires Pour Le Stockage Des Métadonnées Java Dans Des Systèmes Contraints.* In Conférence Française en Système d'exploitations, page 11, Saint Malo, France, 2011. 48, 125

[Courbot 2010] Alexandre Courbot, Gilles Grimaud and Jean-Jacques Vandewalle. *Efficient off-board deployment and customization of virtual machine-based embedded systems.* ACM Trans. Embed. Comput. Syst., vol. 9, no. 3, pages 21:1–21:53, March 2010. 79, 80

[Cousot 1977] Patrick Cousot and Radhia Cousot. *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.* In Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM. 11

[Denning 2005] Peter J. Denning. *The locality principle.* Commun. ACM, vol. 48, no. 7, pages 19–24, July 2005. 58

[Desoli 2002] Giuseppe Desoli, Nikolay Mateev, Evelyn Duesterwald, Paolo Faraboschi and Joseph A. Fisher. *DELI: a new run-time control point.* In Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture, MICRO 35, pages 257–268, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. 22

[Deville 2002] Damien Deville and Gilles Grimaud. *Building an "impossible" verifier on a java card.* In Proceedings of the 2nd conference on Industrial Experiences with Systems Software - Volume 2, WIESS'02, pages 2–2, Berkeley, CA, USA, 2002. USENIX Association. 11

[Dewar 1975] Robert B. K. Dewar. *Indirect threaded code.* Commun. ACM, vol. 18, no. 6, pages 330–331, June 1975. 112

[Dunkels 2006] Adam Dunkels, Niclas Finne, Joakim Eriksson and Thiemo Voigt. *Run-time dynamic linking for reprogramming wireless sensor networks.* In Proceedings of the 4th international conference on Embedded networked sensor systems, SenSys '06, pages 15–28, New York, NY, USA, 2006. ACM. 26

[Duquennoy 2010] Simon Duquennoy. *Smews : un système d'exploitation dédié au support d'applications Web en environnement contraint.* PhD thesis, Université Lille 1 – Sciences et Technologies, July 2010. 127

[Eisenbarth 2007] Thomas Eisenbarth, Sandeep Kumar, Christof Paar, Axel Poschmann and Leif Uhsadel. *A Survey of Lightweight-Cryptography Implementations.* IEEE Des. Test, vol. 24, no. 6, pages 522–533, November 2007. 4, 39

[Ertl 2001] M. Ertl and David Gregg. *The Behavior of Efficient Virtual Machine Interpreters on Modern Architectures.* In Rizos Sakellariou, John Gurd, Len Freeman and John Keane, editeurs, Euro-Par 2001 Parallel Processing, volume 2150 of *Lecture Notes in Computer Science*, pages 403–413. Springer Berlin / Heidelberg, 2001. 112

[Fontaine 2011] Arnaud Fontaine, Samuel Hym and Isabelle Simplot-Ryl. *On-Device Control Flow Verification for Java Programs.* In Úlfar Erlingsson, Roel Wieringa and Nicola Zannone, editeurs, Engineering Secure Software and Systems, volume 6542 of *Lecture Notes in Computer Science*, pages 43–57. Springer Berlin / Heidelberg, 2011. 11

[Ghindici 2007] Dorina Ghindici, Gilles Grimaud and Isabelle Simplot-Ryl. *An information flow verifier for small embedded systems.* In Springer, editeur, Proc. Workshop in

Information Security Theory and Practices 2007 Smart Cards, Mobile and Ubiquitous Computing Systems, volume 4462 of *Lecture Notes in Computer Science*, pages 189–201, Heraklion, Crete, Grèce, 2007. 11

[Gosling 2005] James Gosling, Bill Joy, Guy Steele and Gilad Bracha. Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley)). Addison-Wesley Professional, 2005. 14

[Griffioen 1994] James Griffioen and Randy Appleton. *Reducing file system latency using a predictive approach.* In Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1, USTC'94, pages 13–13, Berkeley, CA, USA, 1994. USENIX Association. 32

[Grimaud 1999] Gilles Grimaud, Jean louis Lanet and Jean-Jacques Vandewalle. *FACADE: a Typed Intermediate Language Dedicated to Smart Cards.* In In Software Engineering ESEC/FSE, number 1687, pages 476–493. Springer-Verlag, 1999. 12, 15, 27, 127

[Gu 2006] Lin Gu and John A. Stankovic. *t-kernel: providing reliable OS support to wireless sensor networks.* In Proceedings of the 4th international conference on Embedded networked sensor systems, SenSys '06, pages 1–14, New York, NY, USA, 2006. ACM. 26, 127

[Guthaus 2001] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge and R. B. Brown. *MiBench: A free, commercially representative embedded benchmark suite.* In Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538), pages 3–14. IEEE, 2001. 39

[Hölzle 1991] Urs Hölzle, Craig Chambers and David Ungar. *Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches.* In Proceedings of the European Conference on Object-Oriented Programming, ECOOP '91, pages 21–38, London, UK, UK, 1991. Springer-Verlag. 84

[Hoogerbrugge 2000] Jan Hoogerbrugge and Lex Augusteijn. *Pipelined Java Virtual Machine Interpreters.* In Proceedings of the 9th International Conference on Compiler Construction, CC '00, pages 35–49, London, UK, UK, 2000. Springer-Verlag. 26

[JCRE.2.2.1 2003] JCRE.2.2.1. JAVA CARD 2.2.1 Runtime Environment Specification. Sun Microsystems, Inc. 2003. 18, 122

[JCVM.2.2.1 2003] JCVM.2.2.1. JAVA CARD 2.2.1 Virtual Machine Specification. Sun Microsystems, Inc. 2003. 17, 18, 85, 122

[Johnson 1994] Theodore Johnson and Dennis Shasha. *2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm.* In Proceedings of the 20th International Conference on Very Large Data Bases, VLDB '94, pages 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc. 46

[Joo 2006] Yongsoo Joo, Yongseok Choi, Chanik Park, Sung Woo Chung, EuiYoung Chung and Naehyuck Chang. *Demand paging for OneNAND™ Flash eXecute-in-place.* In Proceedings of the 4th international conference on Hardware/software codesign and system synthesis, CODES+ISSS '06, pages 229–234, New York, NY, USA, 2006. ACM. 25, 30

[Kandemir 2001] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif and A. Parikh. *Dynamic management of scratch-pad memory space*. In Proceedings of the 38th annual Design Automation Conference, DAC '01, pages 690–695, New York, NY, USA, 2001. ACM. 30

[Kim 2011] Seungkyun Kim, Kiwon Kwon, Chihun Kim, Choonki Jang, Jaejin Lee and Sang Lyul Min. *Demand Paging Techniques for Flash Memory Using Compiler Post-Pass Optimizations*. ACM Trans. Embed. Comput. Syst., vol. 10, no. 4, pages 40:1–40:29, November 2011. 30

[Klint 1981] Paul Klint. *Interpretation Techniques*. Software: Practice and Experience, vol. 11, no. 9, pages 963–973, 1981. 112

[Knuth 1971] Donald E. Knuth. *An empirical study of FORTRAN programs*. Software: Practice and Experience, vol. 1, no. 2, pages 105–133, 1971. 45, 61

[Krall 1998] A. Krall. *Efficient JavaVM Just-in-Time Compilation*. In Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques, PACT '98, pages 205–, Washington, DC, USA, 1998. IEEE Computer Society. 15

[Lin 2007] Jian-Hong Lin, Yuan-Hao Chang, Jen-Wei Hsieh, Tei-Wei Kuo and Cheng-Chih Yang. *A NOR Emulation Strategy over NAND Flash Memory*. In Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA '07, pages 95–102, Washington, DC, USA, 2007. IEEE Computer Society. 25

[Lindholm 1999] Tim Lindholm and Frank Yellin. Java(TM) Virtual Machine Specification, The (2nd Edition). Prentice Hall PTR, 2 édition, April 1999. 12, 14, 16, 75, 84, 90

[McIlroy 2008] Ross McIlroy, Peter Dickman and Joe Sventek. *Efficient dynamic heap allocation of scratch-pad memory*. In Proceedings of the 7th international symposium on Memory management, ISMM '08, pages 31–40, New York, NY, USA, 2008. ACM. 30

[Micron 2006] Micron. *Micron Technical Note 29-19 NAND Flash 101*, 2006. 5, 6

[Miller 2006] Jason E. Miller and Anant Agarwal. *Software-based instruction caching for embedded processors*. SIGOPS Oper. Syst. Rev., vol. 40, no. 5, pages 293–302, October 2006. 22, 23, 30, 67

[Mutlu 2003] Onur Mutlu, Jared Stark, Chris Wilkerson and Yale N. Patt. *Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors*. In Proceedings of the 9th International Symposium on High-Performance Computer Architecture, HPCA '03, pages 129–, Washington, DC, USA, 2003. IEEE Computer Society. 34

[Mutlu 2005] Onur Mutlu, Hyesoon Kim and Yale N. Patt. *Techniques for Efficient Processing in Runahead Execution Engines*. SIGARCH Comput. Archit. News, vol. 33, no. 2, pages 370–381, May 2005. 34

[Necula 1997] George C. Necula. *Proof-carrying code*. In Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '97, pages 106–119, New York, NY, USA, 1997. ACM. 12

[O'Neil 1993] Elizabeth J. O'Neil, Patrick E. O'Neil and Gerhard Weikum. *The LRU-K page replacement algorithm for database disk buffering.* SIGMOD Rec., vol. 22, no. 2, pages 297–306, June 1993. 46

[O'Toole 1993] James O'Toole, Scott Nettles and David Gifford. *Concurrent compacting garbage collection of a persistent heap.* In Proceedings of the fourteenth ACM symposium on Operating systems principles, SOSP '93, pages 161–174, New York, NY, USA, 1993. ACM. 22

[Panda 2000] Preeti Ranjan Panda, Nikil D. Dutt and Alexandru Nicolau. *On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems.* ACM Trans. Des. Autom. Electron. Syst., vol. 5, no. 3, pages 682–704, July 2000. 30

[Park 2003a] Chanik Park, Jaeyu Seo, Sunghwan Bae, Hyojun Kim, Shinhan Kim and Bumsoo Kim. *A low-cost memory architecture with NAND XIP for mobile embedded systems.* In Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, CODES+ISSS '03, pages 138–143, New York, NY, USA, 2003. ACM. 25, 30

[Park 2003b] Chanik Park, Jaeyu Seo, Dongyoung Seo, Shinhan Kim and Bumsoo Kim. *Cost-Efficient Memory Architecture Design of NAND Flash Memory Embedded Systems.* In Proceedings of the 21st International Conference on Computer Design, ICCD '03, pages 474–, Washington, DC, USA, 2003. IEEE Computer Society. 25

[Park 2004] Chanik Park, Junghee Lim, Kiwon Kwon, Jaejin Lee and Sang Lyul Min. *Compiler-assisted demand paging for embedded systems with flash memory.* In Proceedings of the 4th ACM international conference on Embedded software, EMSOFT '04, pages 114–124, New York, NY, USA, 2004. ACM. 30, 33, 67

[Pozo 2005] Roldan Pozo and Miller Bruce. *Scimark 2.0 benchmark.* In http://math.nist.gov/scimark2/, 2005. 39

[Pucheral 2001] Philippe Pucheral, Luc Bouganim, Patrick Valduriez and Christophe Bobineau. *PicoDBMS: Scaling down database techniques for the smartcard.* The VLDB Journal, vol. 10, pages 120–132, 2001. 48

[Rose 1998] Eva Rose. *Lightweight* bytecode *Verification.* In In OOPSALA Workshop on Formal Underpinnings of Java, 1998. 11, 12

[Schoeberl 2010] Martin Schoeberl, Thomas B. Preusser and Sascha Uhrig. *The embedded Java benchmark suite JemBench.* In Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES '10, pages 120–127, New York, NY, USA, 2010. ACM. 39

[Simon 1999] Doug Simon, Doug Simon, Antero Taivalsaari, Antero Taivalsaari, Antero Taivalsaari, Bill Bush, Bill Bush and Bill Bush. *The Spotless System: Implementing a Java System for the Palm Connected Organizer.* Rapport technique, Sun Microsystems, Inc, 1999. 39, 79

[Simon 2006] Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels and Derek White. *Java™ on the bare metal of wireless sensor devices: the squawk Java virtual machine.* In Proceedings of the 2nd international conference on Virtual execution environments, VEE '06, pages 78–88, New York, NY, USA, 2006. ACM. 40, 79

[Smith 1982] Alan Jay Smith. *Cache Memories.* ACM Comput. Surv., vol. 14, no. 3, pages 473–530, September 1982. 19, 27, 32

[Smith 2001] L. A. Smith, J. M. Bull and J. Obdrzálek. *A parallel java grande benchmark suite.* In Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM), Supercomputing '01, pages 8–8, New York, NY, USA, 2001. ACM. 39

[Smith 2005] James E Smith. *Virtual machines: versatile platforms for systems and processes.* 2005. 112

[SPEC 1998] SPEC. *JVM98 Benchmarks.* In www. spec. org/osg/jvm98/, 1998. 39

[Stärk 2001] Robert Stärk, Joachim Schmid and Egon Börger. Java and the java virtual machine - definition, verification, validation. 2001. 11

[Tanenbaum 2001] Andrew S. Tanenbaum. Modern operating systems. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd édition, 2001. 24

[Toshiba 2003] Toshiba. *NAND vs. NOR Flash Memory, Technology Overview*, 2003. 5, 6

[Verma 2004] Manish Verma, Lars Wehmeyer and Peter Marwedel. *Dynamic overlay of scratchpad memory for energy minimization.* In Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, CODES+ISSS '04, pages 104–109, New York, NY, USA, 2004. ACM. 22

[Wilson 1992] Paul Wilson. *Uniprocessor garbage collection techniques.* In Yves Bekkers and Jacques Cohen, editeurs, Memory Management, volume 637 of *Lecture Notes in Computer Science*, pages 1–42. Springer Berlin / Heidelberg, 1992. 10.1007/BFb0017182. 22

[Wilson 1995] Paul R. Wilson, Mark S. Johnstone, Michael Neely and David Boles. *Dynamic Storage Allocation: A Survey and Critical Review.* In Proc. Int. Workshop on Memory Management, Kinross Scotland (UK), 1995. 21

Appendice A

Compléments cartographiques

Cette section reprend une série d'exemples de cartographies des points chauds. Dans ces figures, chaque pixel représente un octet dans le binaire exécutable du programme, et une ligne correspond à 512 octets. Un point bleu matérialise un octet lu et exécuté moins de 100 fois, un point vert moins de 1000 fois, un point orange moins de 10000, un point rouge moins de 100000 et un point noir plus de 100000 fois. Tous les points blancs correspondent à des instructions présentes dans le binaire mais qui ne sont jamais exécutées. Le nombre d'octets par tranche de chaleur est rappelé dans le tableau suivant :

Programme	bleu, <100	vert, <1000	orange, <10000	rouge, <100000	noir, >100000
Rijndael	11 589	41	4 677	2	14
GSM	15 391	13 801	4 785	549	0
SHA	8 240	27	374	522	78
KVM	21 193	1 433	620	754	40
Richards V7 C++	10 274	65	1 227	808	92

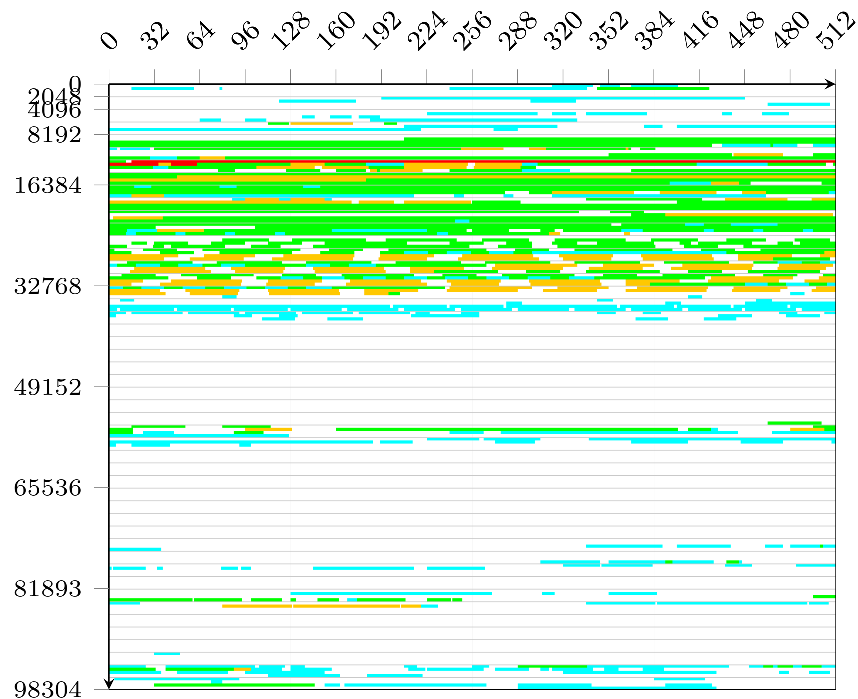


Figure A.1: GSM

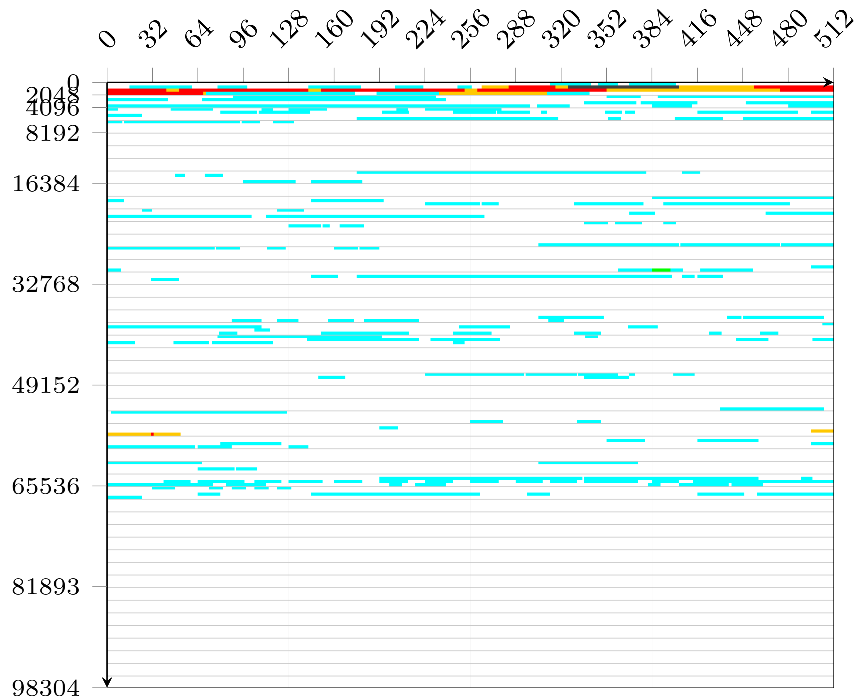


Figure A.2: SHA

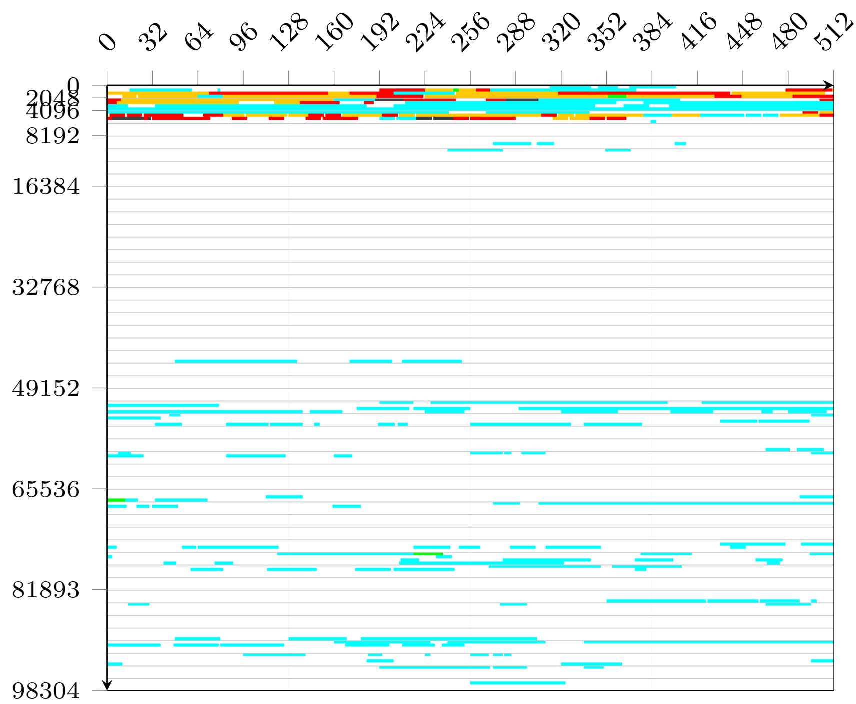


Figure A.3: Richards V7 C++

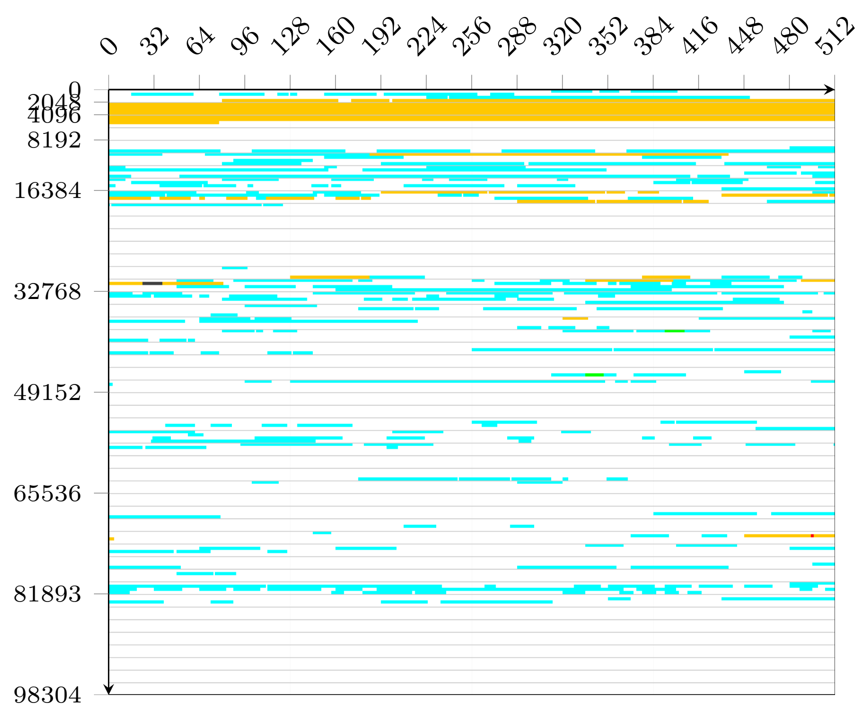


Figure A.4: Rijndael

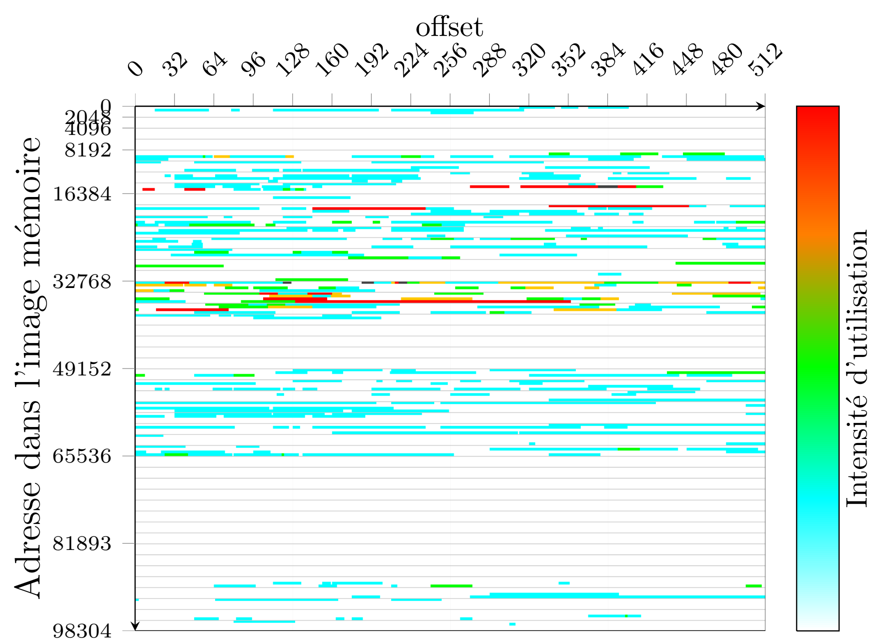
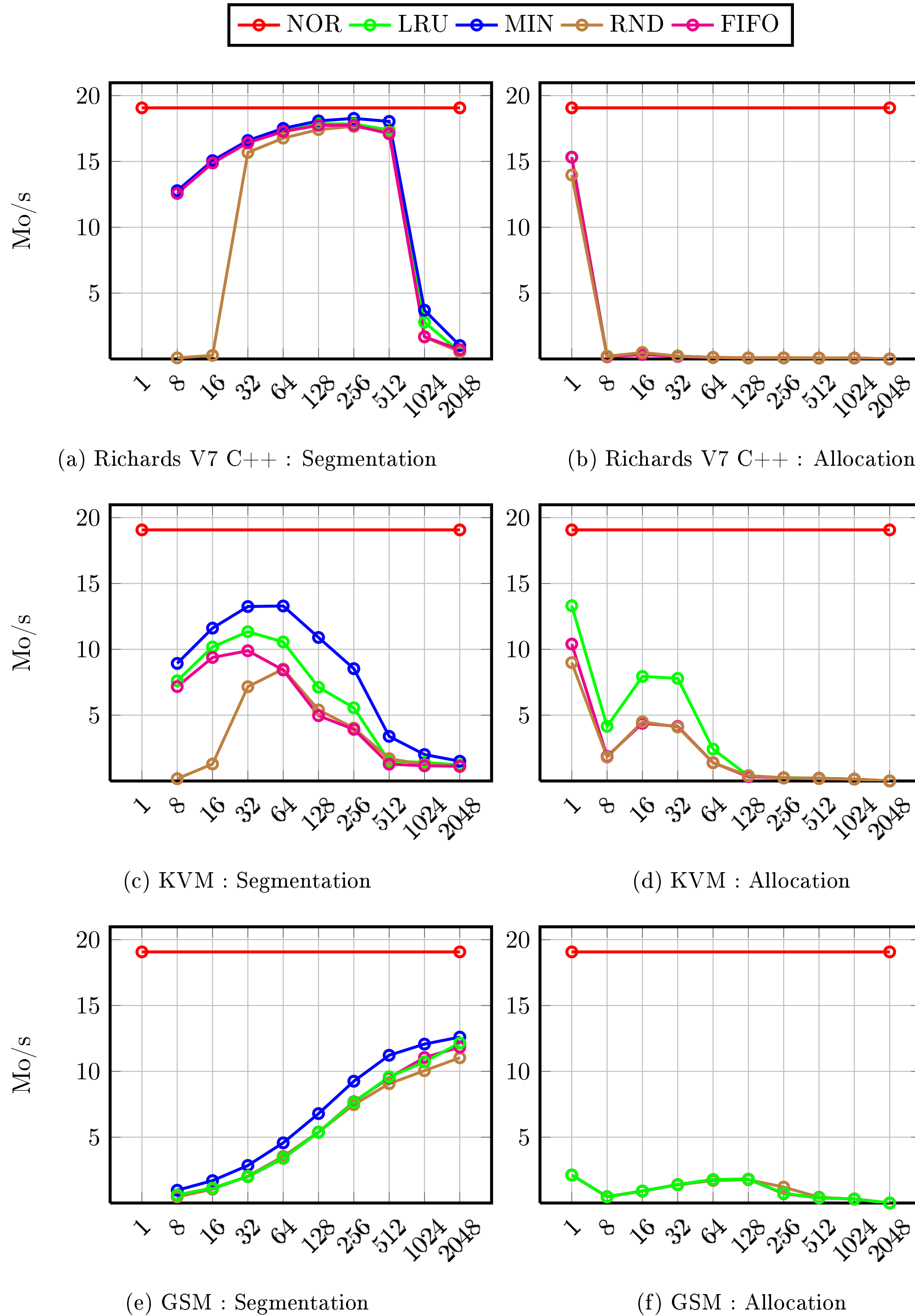


Figure A.5: KVM

Appendice B

Compléments graphiques sur les débits



(a) Richards V7 C++ : Segmentation

(b) Richards V7 C++ : Allocation

(c) KVM : Segmentation

(d) KVM : Allocation

(e) GSM : Segmentation

(f) GSM : Allocation

Figure B.1: Débit d'un cache de 4096 octets

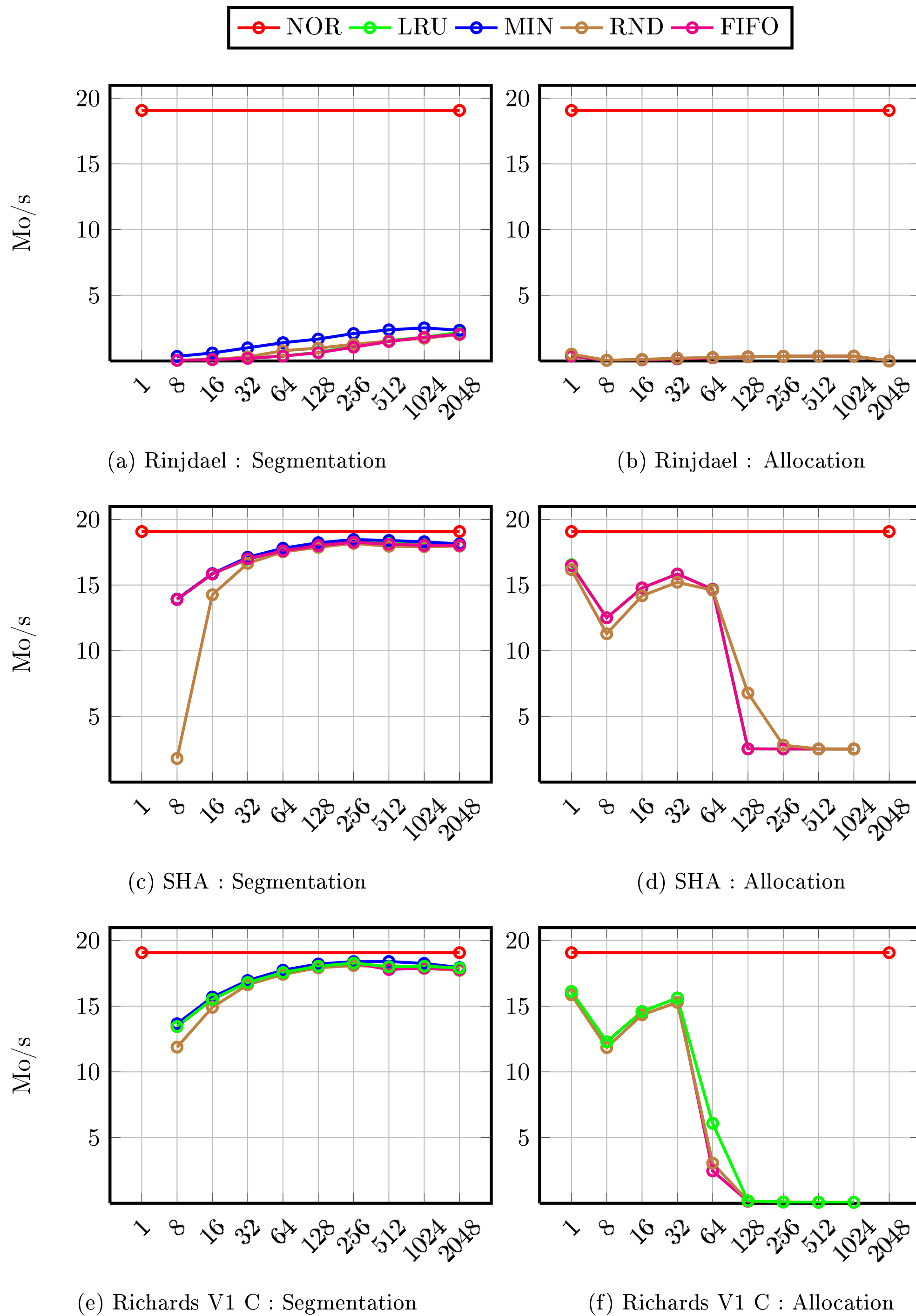


Figure B.2: Débit d'un cache de 4096 octets

Appendice C

Structures de données

Listing C.1: Liste doublement chaînée

```
#define DLIST_PAGECOUNT ((CACHESIZE-8)/(PAGESIZE+12))

typedef struct cache_entry {
   uint8_t page[PAGESIZE];
   addr_t  flash_addr;
}cache_entry_t;

typedef struct list_entry{
   cache_entry_t      entry;
   struct list_entry  *prev;
   struct list_entry  *next;
}list_entry_t;

typedef struct cache_t{
   list_entry_t list[DLIST_PAGECOUNT];
   list_entry_t *first;
   list_entry_t *last;
}dlist_cache_t;
```

Listing C.2: Table de hachage

```
#define HASH_PAGECOUNT ((CACHESIZE-8*BUCKETCOUNT-8)/(PAGESIZE+12))

typedef struct bucket{
   addr_t   flash_addr;
   uint32_t lru_page_timestamp;
   uint8_t  page[PAGESIZE];
   struct bucket *next;
} bucket_t;

typedef struct cache_t{
   bucket_t* current;
   bucket_t* bucket_table[BUCKETCOUNT];
   bucket_t* last_in_list[BUCKETCOUNT];
   bucket_t  pages        [HASH_PAGECOUNT];
   uint32_t  cache_timestamp;
} hash_cache_t;
```

Listing C.3: Arbre rouge et noir

```
#define RBTREE_PAGECOUNT ((CACHESIZE-12)/(PAGESIZE+24))

typedef enum rb_color {
   RB_BLACK,
   RB_RED,
}rb_color;

typedef struct rbtree_node {
   addr_t flash_addr;
   struct rbtree_node *left, *right;
   struct rbtree_node *parent;
   rb_color color;
   uint32_t lru_page_timestamp;
   uint8_t  page[PAGESIZE];
} rbtree_node_t;

typedef struct rbtree_cache{
   rbtree_node_t *root;
   rbtree_node_t *current;
   uint32_t      cache_timestamp;
   rbtree_node_t nodes[RBTREE_PAGECOUNT];
} rbtree_cache_t;
```

Listing C.4: Tirage aléatoire

```
#define RND_PAGECOUNT ((CACHESIZE-4)/(PAGESIZE+4))

typedef struct  page{
   u1     data[PAGESIZE];
   addr_t flash_addr;
} page_t;

typedef struct rnd_cache{
   page_t  pages[RND_PAGECOUNT];
   page_t* currrent;
}rnd_cache_t;
```

Résumé

La dernière génération de cartes à puce permet le téléchargement d'applications après leur mise en circulation. Outre les problèmes de sécurité que cela implique, cette capacité d'extension applicative reste encore aujourd'hui bridée par un espace de stockage adressable restreint. La thèse défendue dans ce mémoire est qu'il est possible d'exécuter efficacement des applications stockées dans la mémoire non-adressable des cartes à puce, disponible en plus grande quantité, et ce, malgré ses temps de latences très longs, donc peu favorables *a priori* à l'exécution de code.

Notre travail consiste d'abord à étudier les forces et faiblesses de la principale réponse proposée par l'état de l'art qu'est un cache. Cependant, dans notre contexte, il ne peut être implémenté qu'en logiciel, avec alors une latence supplémentaire. De plus, ce cache doit respecter les contraintes mémoires des cartes à puce et doit donc avoir une empreinte mémoire faible.

Nous montrons comment et pourquoi ces deux contraintes réduisent fortement les performances d'un cache, qui devient alors une réponse insuffisante pour la résolution de notre challenge. Nous appliquons notre démonstration aux caches de code natif, puis de code et méta-données Java et JavaCard2. Fort de ces constats, nous proposons puis validons une solution reposant sur une pré-interprétation de code, dont le but est à la fois de détecter précocement les données manquantes en cache pour les charger à l'avance et en parallèle, mais aussi grouper des accès au cache et réduire ainsi l'impact de son temps de latence logiciel, démontré comme son principal coût. Le tout produit alors une solution efficace, passant l'échelle des cartes à puce.

Abstract

The latest generation of smart cards allows to download applications after they are released. In addition to new security issues, this ability still remains constrained by a limited addressable storage space. The claim of our thesis is that it is possible to efficiently execute applications stored in the non-addressable memory of smart cards, available in larger quantities, and, despite its very long latency.

Our work is first a study of the strengths and weaknesses of the main solution provided by the state of the art : caching. However, in our context, it can be implemented only in software,then with an additional latency. In addition, the cache must comply with memory constraints of smart cards and must then have on a low memory footprint.

We show how and why these two constraints greatly reduce the performance of a cache, that finally becomes inadequate to fully tackle our challenge. We apply our demonstration to native instructions caches, then to Java/JavaCard2 instructions and metadata caches. Based on our observations, we propose and validate a solution based on a code pre-interpretation, whose purpose is to early detect of missing data in the cache to fetch them in advance, then to group accesses to the cache in order to reduce the impact of its software latency, that we demonstrate to be its main cost. Our result is an effective solution, that fits smart cards size and constraints.