Olivier Savry
Thomas Hiscock
Mustapha El Majihi

SÉCURITÉ MATÉRIELLE DES SYSTÈMES

Vulnérabilités des processeurs et techniques d'exploitation

DUNOD

Illustration de couverture : matejmo/istockphoto.com

Cet ouvrage a été réalisé avec le soutien d'Investir l'avenir et de Nanoelec

© Dunod, 2019
11 rue Paul Bert, 92240 Malakoff
www.dunod.com
ISBN 978-2-10-079096-8

Table des matières

Introduction

Les cyber-attaques en tout genre ont envahi les actualités quotidiennes. La dépendance accrue de nos sociétés aux solutions automatisées, aux algorithmes et au numérique en général ouvre la voie à des menaces de plus en plus prégnantes. Des données récentes laissent à penser que la croissance des cyber-attaques n'est plus linéaire, mais qu'elle a pris une trajectoire exponentielle. Un phénomène poussé par l'ingéniosité toujours renouvellée des attaquants. Ces derniers évoluent dans un monde de plus en plus complexe, et donc de plus en plus vulnérable où l'accès à l'information et à la connaissance est facilité par nos modes de communications ultra-rapides.

L'intrication et l'interopérabilité de tous les systèmes qui nous entourent – transports (trains, avions, voitures…), énergie, gestion des ressources, habitat (domotique), gestion de la ville (*smart cities*), outils de production (usines) – où le maillon le plus faible est susceptible de présenter des failles, ouvrant la porte à des risques toujours plus probables de black-out total. Si on remonte l'arbre de causes de ces attaques, le cœur du problème se révèle : les processeurs. Ils sont la partie active de ces CPS (*cyber physical systems*) et de l'IoT (*internet of things*) relié par l'Internet. Nous nous appuyons tous les jours sur ces solutions pour nous simplifier la vie, au prix d'un risque d'aliénation que l'on ne mesure pas toujours quand leur sécurité n'est pas assurée.

Une étude du cycle de développement de ces systèmes à base de processeurs montre que les failles ont trois raisons essentielles :

1. La sécurité des processeurs et des logiciels embarqués n'est pas pensée en amont du développement du système. Bien souvent, on réalise après coup (souvent à la suite d'une attaque) que des vulnérabilités structurelles sont présentes. La réaction est alors de « patcher », avec le risque de déplacer le problème et de dégrader les performances.
2. L'implémentation de fonctions de sécurité est un travail difficile nécessitant l'avis d'experts. Malheureusement, il y aura toujours plus de développeurs et de concepteurs que de spécialistes de la sécurité (qui sont rares et coûteux).
3. Les ingénieurs, les développeurs et même les experts n'ont pas toujours les outils pour répondre au défi de la sécurité de systèmes toujours plus complexes.

> Outre le fait que, souvent, on ne peut adapter l'implémentation de la sécurité qu'à un certain niveau défini de risques, il est à regretter qu'on n'ait pas sur le marché de solutions accessibles à tous pour développer des produits sûrs.

C'est dans ce contexte que cet ouvrage prend tout son sens. Nous espérons qu'il apportera aux développeurs, aux étudiants en informatique et aux électroniciens numériques des moyens d'appréhender les vulnérabilités des processeurs, de prendre conscience des nombreuses attaques qu'elles rendent possibles et de leur exploitation. Nous avons voulu concevoir un ouvrage d'accès facile et qui se suffit à lui-même, tout en permettant au lecteur d'approfondir les sujets abordés avec de nombreuses références vers des articles scientifiques plus pointus, rédigés par des experts du domaine. Toute cette bibliographie est aisément accessible en ligne. Ainsi, ce livre pourra servir avantageusement de guide à des concepteurs de processeurs voire de systèmes complexes qui se soucient un tant soit peu de leur sécurité.

Lors de la rédaction, il n'a pas été simple de définir un cadre de travail sans laisser échapper de vulnérabilité capitale. Nous avons pris le parti de regarder le synoptique d'une architecture moderne de processeur avec tous ses composants et de déterminer les menaces qui pesaient sur chacun d'eux. La Figure I.1 est une architecture générique fusionnant des concepts venant à la fois d'Intel, d'AMD et d'ARM qui a constitué le plan de ce document.

Une grande partie du schéma est couverte par les différents éléments de la hiérarchie mémoire : registres, caches, DRAM, flash ou disque dur, qui sont les principales portes d'entrée des attaques. Le premier chapitre est entièrement dédié à l'étude de ces menaces. Nous plongerons ensuite au plus profond de l'architecture. Nous voyons sur la figure les multiples cœurs incontournables des puces actuelles, avec de l'exécution parallèle et désordonnée d'instructions et de la prédiction de branchements. Ces éléments nécessaires à l'augmentation des performances sont, depuis les attaques récentes et très médiatisées Spectre et Meltdown, des accès aisés pour les hackers. Le seond chapitre dressera un état des lieux des vulnérabilités qui apparaissent à ce niveau de conception (la microarchitecture) et abordera en détails les attaques *Spectre* et *Meltdown*. Nous ferons ensuite une incursion vers les vulnérabilités des programmes comme le fameux *buffer overflow* qui exploite les structures des processus en mémoire. Nous y verrons également d'autres possibilités pour détourner le flot de contrôle des programmes. Toutefois, on se limitera à l'étude des failles qui sont la conséquence de vulnérabilités du matériel. Ainsi, une injection SQL ou une attaque XSS ne seront pas abordées car ce sont des attaques purement software qui laisse l'attaquant confiné aux logiciels attaqués comme la base de données ou le serveur

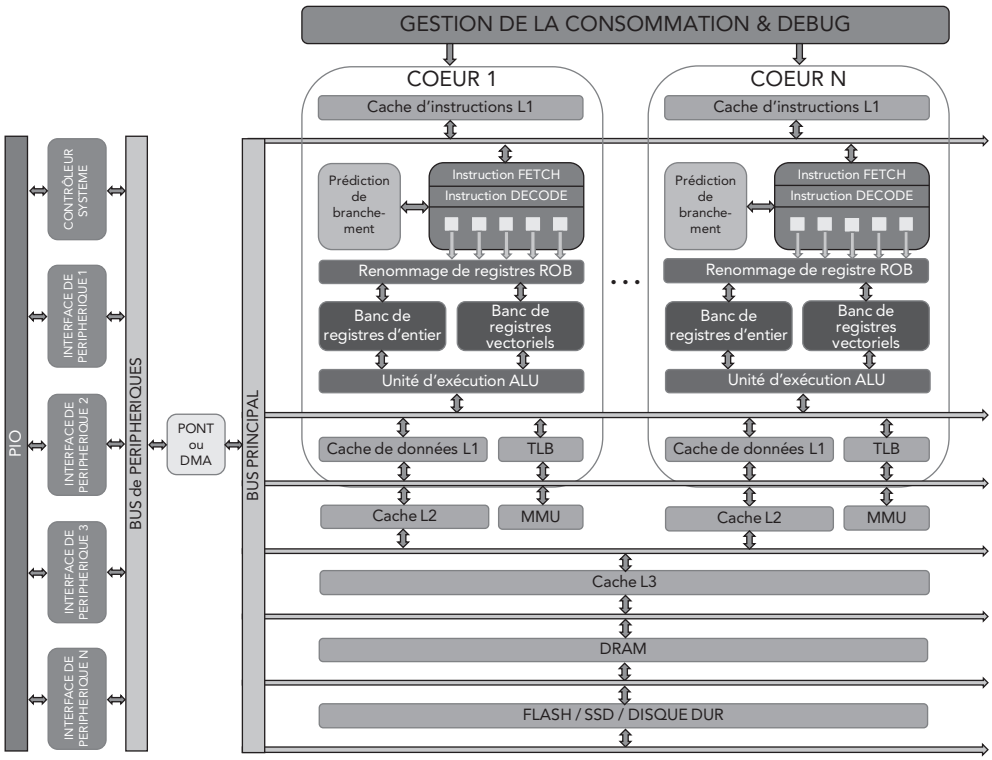

Figure I.1 Synoptique d'un processeur moderne

web PHP, contrairement à un *buffer overflow* qui mettra en défaut l'intégrité de la mémoire du processeur. Le chapitre suivant sera consacrée aux *secure boot* dont la conception très contrainte est une source de failles. En suivant notre synoptique de la Figure I.1, le cinquième chapitre étudiera les bus, les interfaces de debug et d'administration à distance (Intel ME), les périphériques, le DMA (*direct memory access*) et la gestion de l'énergie qui apportent leur lot de menaces. Enfin, l'ouvrage s'achèvera sur deux chapitres importants : les attaques par canaux cachés comme la SPA (*simple power analysis*) et la DPA (*differential power analysis*) et les attaques en faute avec entre autres l'exemple de la DFA (*differential fault analysis*).

Pour chaque attaque ou vulnérabilité étudiée, nous nous efforcerons de spécifier son cadre précis, ses causes, ses conséquences, les composants du processeur affectés, les biens mis en danger et son opportunité, c'est-à-dire la probabilité qu'elle apparaisse. Nous nous focaliserons bien sûr sur les principes de fonctionnement de ces vulnérabilités, ce qui donnera au lecteur l'occasion de revisiter ou de découvrir l'architecture des processeurs.

Nous conclurons en dérivant de toutes ces attaques des objectifs de sécurité à même de faire prendre conscience à chacun des exigences d'un processeur sécurisé et du chemin qui reste à faire pour y parvenir.

Une dernière remarque avant d'entamer cette plongée dans l'architecture des processeurs : nous avons pris le parti de garder les termes anglo-saxons pour un certain nombre de désignations d'attaques afin de préparer les novices à la lecture d'ouvrage majoritairement en anglais, mais aussi pour faciliter la compréhension des lecteurs déjà compétents dans le domaine.

Enfin, nous tenons à remercier l'IRT NanoElec et tous ses membres pour avoir rendu possible cet ouvrage.

1 Vulnérabilités des mémoires

Dans ce chapitre, nous allons introduire dans un premier temps des notions préliminaires sur la hiérarchie mémoire en l'occurrence, la mémoire cache, la mémoire virtuelle, la mémoire SRAM, la mémoire DRAM et la mémoire Flash. Dans un deuxième temps, nous présenterons les vulnérabilités potentielles de ces composants matériels à travers l'explication et l'analyse des principales attaques qui profitent de ces failles de sécurité.

1.1 Préliminaires

1.1.1 Hiérarchie mémoire

La hiérarchie mémoire d'un système numérique est constituée de différents types de mémoires. Ces dernières se distinguent par leurs vitesses d'accès, leurs débits, leurs capacités et leurs coûts. La rapidité de la mémoire est dépendante de sa technologie de fabrication et par conséquent de son prix. Depuis la fin des années 1980, les performances des processeurs se sont améliorées bien plus rapidement que celles des mémoires DRAMs. Il y a ainsi aujourd'hui une différence très importante entre la vitesse d'exécution d'un processeur et la latence d'un accès en mémoire DRAM (phénomène du « *memory wall* », Figure 1.1).

Pour combler cette différence, la mémoire est structurée en une hiérarchie de mémoires de différentes latences, tailles, débits et coûts. La conception d'une hiérarchie mémoire est un sujet de recherche complexe et très actif. La Figure 1.2 illustre de manière simplifiée la hiérarchie mémoire d'un serveur et celle d'un téléphone mobile.

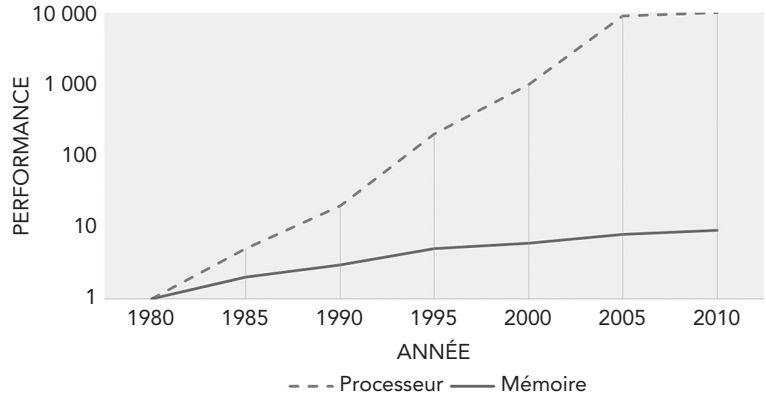

Figure 1.1 Évolution de la différence entre la performance d'un processeur et la latence d'un accès en mémoire DRAM

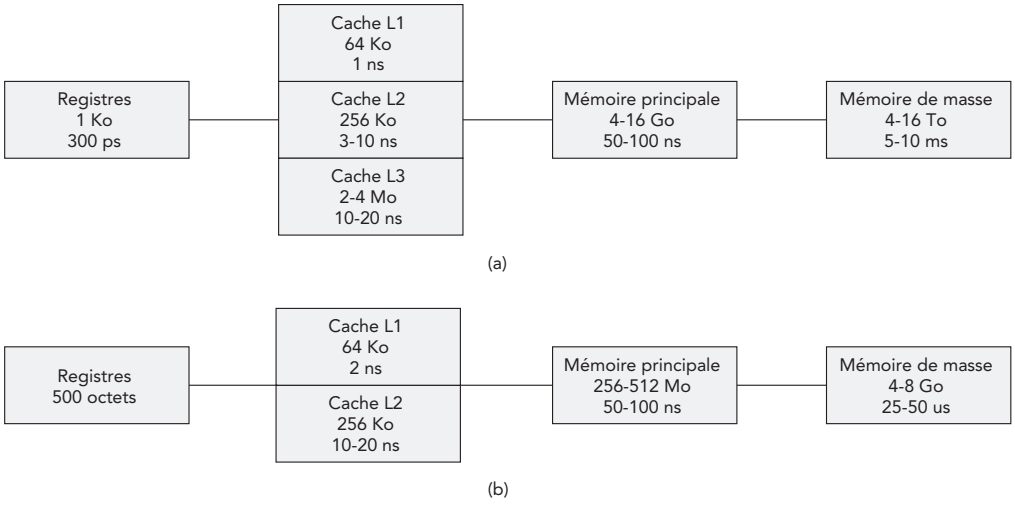

Figure 1.2 Hiérarchies mémoire d'un serveur (a) et d'un téléphone mobile (b)

Tout en bas de la hiérarchie mémoire se trouvent les registres, qui fonctionnent à la même vitesse que le processeur et servent de stockage interne pour une très faible quantité de données. Au-dessus, sont ajoutés un ensemble de mémoires cache, moins rapides que les registres, mais plus volumineuses. Enfin, la mémoire principale de type DRAM est placée avant le stockage de masse qui présente une forte latence.

1.1.2 Mémoire cache

Les caches ou mémoires cache sont des mémoires très rapides mais de faible capacité de stockage (souvent implémentées à l'aide de technologies SRAMs) servant à stocker des données temporaires. Les caches ont été instaurés dans

les architectures des processeurs afin de limiter l'impact des temps d'accès à la mémoire depuis le processeur vers la mémoire principale et vice-versa. Le processeur essaie d'accéder à une donnée ou une instruction d'abord dans le cache avant de passer à la mémoire principale. En cas d'échec (*miss*), la donnée ou l'instruction est chargée et gardée dans le cache pour un accès futur. En cas de succès (*hit*), la mémoire principale n'est pas accédée. Les caches sont conçus autour de deux principes fondamentaux :

- La localité temporelle : lorsqu'une donnée est accédée par une application, il y a une grande chance qu'elle soit accédée à nouveau dans un futur proche.
- La localité spatiale : lorsqu'une donnée est accédée, il y a une grande chance qu'une donnée proche en mémoire soit accédée dans un futur proche.

Pour répondre à ces principes, les caches stockent et gardent en mémoire rapide les données accédées récemment (localité temporelle). De plus, ils stockent les données par blocs (localité spatiale). Enfin, pour améliorer le taux de succès du cache, des algorithmes de *pre-fetching* (chargement en avance des données dont le CPU devrait avoir besoin) basés sur ces principes de localité sont utilisés.

Politique d'écriture d'un cache

Les données accédées par le processeur sont stockées dans le cache et dans la mémoire principale. Quand une donnée est modifiée localement, le processeur doit la mettre à jour dans le cache et dans la mémoire principale. Pour ce faire, plusieurs politiques de mise à jour existent :

- Écriture immédiate (*write-through*) : le processeur écrit la donnée à la fois dans le cache et dans la mémoire principale.
- Écriture différée (*write-back*) : le processeur n'écrit la donnée dans la mémoire principale que lorsque la ligne qui la contient dans le cache est évincée. Chaque ligne de cache contient un bit (*bit dirty*) qui indique la modification de la ligne et la nécessité de mise à jour dans la mémoire principale.

Le cache est divisé en blocs de même taille appelés lignes de caches. La correspondance entre une ligne de cache et un bloc de la mémoire principale se fait de différentes manières. On distingue trois types de caches : les caches à correspondance directe, les caches complètement associatifs et les caches associatifs par ensemble.

Caches à correspondance directe

Dans les caches à correspondance directe, chaque bloc de la mémoire principale est mappé à une unique ligne de cache. Généralement pour ce type de cache, la correspondance se fait de cette manière :

Indice de cache = (adresse du bloc dans la mémoire principale) modulo (nombre de lignes dans le cache)

Pour les caches dont le nombre de lignes n est de puissance 2 (par exemple $n = 2^p$), l'opération modulo revient à choisir les p premiers bits de l'adresse du bloc dans la mémoire principale comme adresse de la ligne de cache. La Figure 1.3 présente un exemple d'une mémoire principale de 16 blocs et un cache de 4 lignes. Les blocs 0001, 0101, 1001 et 1101 de la mémoire principale correspondent tous à la même ligne de cache indexée par 01.

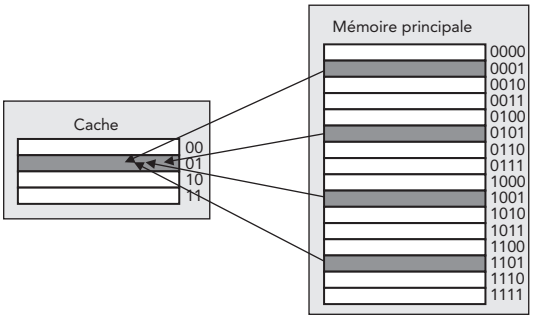

Figure 1.3 Exemple d'un cache à correspondance directe

Plusieurs blocs de la mémoire principale sont ainsi associés à la même ligne de cache. Pour les distinguer, une étiquette est utilisée. Elle spécifie précisément avec l'indice de la ligne de cache l'adresse du bloc dans la mémoire principale. Au démarrage du processeur, le cache contient des données non valides et même après l'exécution de plusieurs instructions certaines lignes de cache demeurent vides. C'est pour cette raison qu'un bit de validité est rajouté pour indiquer si les données indexées sont valables ou non. La Figure 1.4 présente l'exemple d'un cache à correspondance directe pour une architecture MIPS 32 bits. La taille de l'étiquette pour cette configuration est égale à $32 - (n + m + 2)$, où 2^n est le nombre de lignes dans le cache et 2^m est le nombre de mots dans un bloc. Pour cet exemple, les adresses et les données sont de taille 32 bits, un bloc de mémoire principale contient un mot de 4 octets et la taille du cache est égale à $2^{10} = 1\,024$ lignes. Les deux premiers bits d'une adresse permettent de choisir parmi les 4 octets d'un mot, les 10 bits suivant indexent la ligne de cache et, dans le cas où le bit de validité est égal à 1, une

comparaison est faite sur l'étiquette de taille 20 bits pour savoir s'il y a un échec ou succès d'accès. Si le bit de validité est égal 0, un échec d'accès est remonté au processeur.

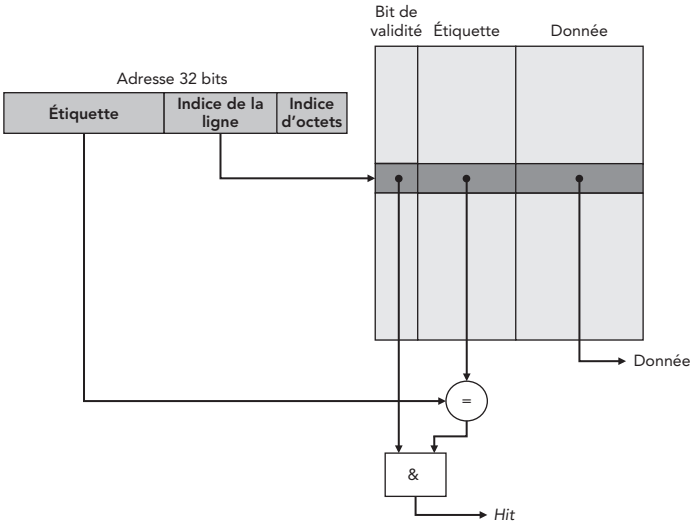

Figure 1.4 Exemple d'un cache à correspondance directe

Cette méthode a l'avantage de permettre un accès rapide à la ligne de cache car on sait directement où elle se trouve. Cependant, elle est peu efficace en pratique car elle crée un comportement conflictuel plus récurrent. La solution est de permettre à un bloc de la mémoire principale d'être mappé à un nombre limité de lignes de caches (les caches complètement associatifs ou associatifs par ensemble).

Caches complètement associatifs

Dans les caches complètement associatifs, chaque bloc de la mémoire principale peut être associé à n'importe quelle ligne de cache. Pour trouver un bloc, il faut parcourir toutes les lignes du cache. Cette recherche doit se faire en parallèle pour optimiser le temps d'accès. Cela implique l'association coûteuse d'un comparateur à chaque entrée de cache. Cette méthode d'association ne demeure utilisable en pratique que dans les cas où le nombre de blocs mémoire est petit.

Caches associatifs par ensemble

C'est un compromis entre les deux types précédents de cache. Le cache est subdivisé en ensembles (*sets*) de *N* lignes de cache, chaque ensemble est subdivisé lui-même en *p* blocs *ways*. On parle d'un cache à *N-sets* et *p-way*. Chaque bloc

de la mémoire principale peut être affecté à un ensemble unique selon la formule suivante :

Indice de cache = (adresse du bloc dans la mémoire principale) modulo (nombre de *sets* dans le cache)

Mais il va être placé aléatoirement dans un *way* de cet ensemble. Il faut chercher toutes les étiquettes d'un ensemble pour trouver un bloc mémoire. Augmenter le nombre d'associativité a pour but de diminuer le taux d'échec d'accès, mais en contrepartie cela augmente le temps d'accès à une donnée déjà présente dans le cache. La Figure 1.5 donne un exemple d'un cache associatif par ensemble (256 *sets* et *4-way*). Son implémentation nécessite quatre comparateurs et un multiplexeur.

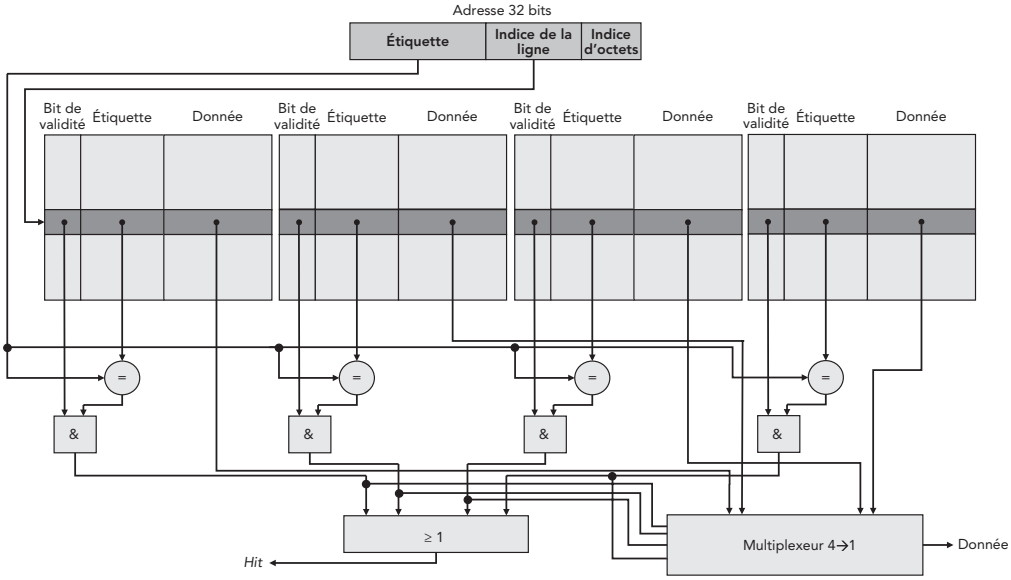

Figure 1.5 Exemple d'implémentation d'un cache associatif par ensemble

Le choix d'un cache à correspondance directe, d'un cache complètement associatif ou d'un cache associatif par ensemble va dépendre du coût du taux d'échec d'accès *versus* le coût associé à l'implémentation du cache associatif ou du cache associatif par ensemble. Ce coût intègre le temps d'accès et le silicium ajouté.

1.1.3 Mémoire virtuelle

Les premiers processeurs n'exécutaient qu'un programme simple qui avait accès à toute la mémoire du système. Très rapidement s'est fait ressentir le besoin de faire cohabiter plusieurs tâches. La motivation principale était de combler les temps élevés des opérations d'entrées-sorties en entrelaçant l'exécution de plu-

sieurs programmes (processus). Aussi, les premiers systèmes multitâches travaillaient directement sur la mémoire physique. Chaque processus avait une zone de la mémoire qui lui était attribuée. Un programme particulier, le système d'exploitation (OS) était chargé de partager le temps d'exécution du processeur entre les différents processus (Figure 1.6).

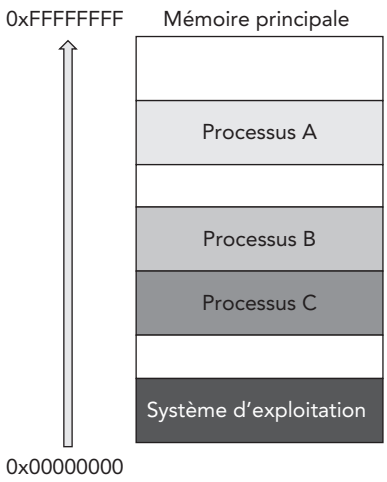

Figure 1.6 Mémoire physique partagée sans utilisation de mémoire virtuelle

Ce partage de la mémoire et du temps du processeur était déjà un net progrès par rapport aux systèmes monotâches. Cependant, cette approche présente un certain nombre de limites importantes :

- La plus importante est l'impossibilité d'accéder à tout l'espace mémoire disponible.
- Il est difficile de gérer les droits d'accès, par exemple empêcher le processus A d'accéder à la zone mémoire du processus B. Une première solution avait été d'ajouter un registre supplémentaire (*bound register*) pour borner les zones accessibles. Mais cela ne permet qu'une gestion grossière des droits.
- Enfin, il faut compiler les programmes en connaissant l'adresse à laquelle ils seront dans le système. Ce qui d'une part limite la portabilité des applications, et, d'autre part, prohibe toute forme de partage de code (librairies partagées).

Le mécanisme de mémoire virtuelle apporte une réponse à ces limitations en ajoutant un niveau d'indirection dans les adresses. Chaque processus dispose ainsi d'un espace d'adressage virtuel. Un exemple de plan d'adressage virtuel est représenté sur la Figure 1.7. Un mécanisme de traduction spécifique à chaque processus est configuré par le système d'exploitation et permet d'obtenir l'adresse physique associée à chaque adresse virtuelle.

Cette indirection supplémentaire au décodage d'adresse permet à n'importe quel processus d'accéder à tout l'espace d'adressage, voire même à plus de mémoire que le système n'en dispose réellement. La mémoire principale est alors utilisée comme une mémoire cache pour le stockage de masse (mémoire flash, disque magnétique…). Aussi, on trouvera dans les mécanismes de mémoire virtuelle de nombreuses similarités de conception avec les mémoires caches. Les mécanismes de mémoire virtuels utilisés aujourd'hui effectuent un découpage de la mémoire physique en pages de taille fixe, en général 4 ko. Ces pages sont utilisées comme les lignes d'un cache et sont chargées en mémoire principale seulement à la demande : si l'on accède à une page non chargée (l'équivalent d'un *miss*) un défaut de page est généré (*page fault*) et la page est chargée en mémoire à ce moment. Lorsqu'un processeur utilise la mémoire virtuelle, il émet des adresses virtuelles qui sont transformées en adresses physiques par la MMU (*memory management unit*). Ces dernières permettent ainsi d'accéder à la mémoire principale. Prenons l'exemple d'une traduction simple d'adresse (avec une seule table des pages). Une adresse virtuelle est alors décomposée en deux parties : un numéro de page et un déplacement relatif au début de la page (l'offset de la page). La Figure 1.8 montre un exemple d'une traduction d'adresse. Le nombre de bits de l'offset de la page indique la taille de la page.

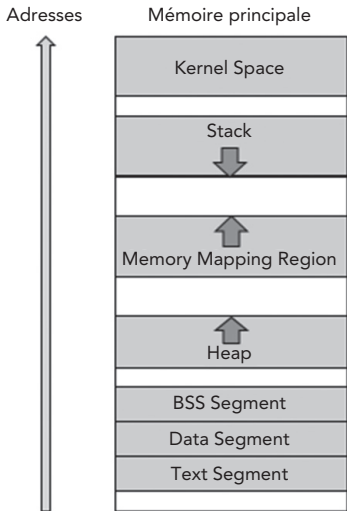

Figure 1.7 Exemple de la vision de la mémoire pour un processus x86 Linux

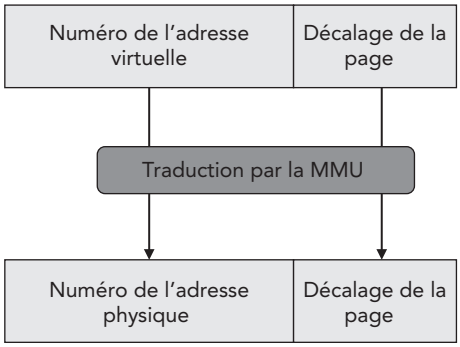

Figure 1.8 Correspondance entre une adresse virtuelle et une adresse physique

Lors de la traduction, il se peut que la page ne soit pas présente en mémoire et donc, n'ait pas d'adresse physique associée dans la table. Dans ce cas, un défaut de page est généré et la page doit être chargée en mémoire par le système d'exploitation. Un défaut de page a un coût important en temps d'accès à cause de la latence des mémoires de masse par rapport à la mémoire principale. Ce constat est pris en compte dans la conception des mémoires virtuelles :

- La taille des pages virtuelles doit être assez grande pour amortir le temps d'accès mémoire élevé. La taille typique sur des ordinateurs personnels est de 4 ko. Selon les applications, on utilise parfois des pages de 16 ko voire jusqu'à 32 ko ou 64 ko pour des serveurs récents. Pour les systèmes embarqués, la taille typique est de 1 ko.
- La correspondance complètement associative est privilégiée : une page virtuelle peut être mappée à n'importe quelle page physique. Une table de page est utilisée pour indexer toutes les pages physiques dans la mémoire virtuelle pour éviter de la parcourir entièrement pour trouver une page physique (Figure 1.9).
- L'écriture différée (politique *write-back*) est utilisée à cause de la latence d'écriture dans la mémoire de masse.
- Comme nous l'avons évoqué, les défauts de page sont gérés par la couche logicielle. Pour ce faire plusieurs algorithmes de remplacement de pages existent, en l'occurrence l'algorithme optimal de Belady, l'algorithme FIFO, l'algorithme de remplacement de la page non récemment utilisée (NRU), l'algorithme de la page la moins récemment utilisée et bien d'autres. Ces algorithmes peuvent se restreindre à remplacer juste les pages du processus qui a provoqué le défaut de page, ce qu'on appelle l'allocation locale, comme ils peuvent remplacer toutes les pages en mémoire. On parle d'allocation globale.

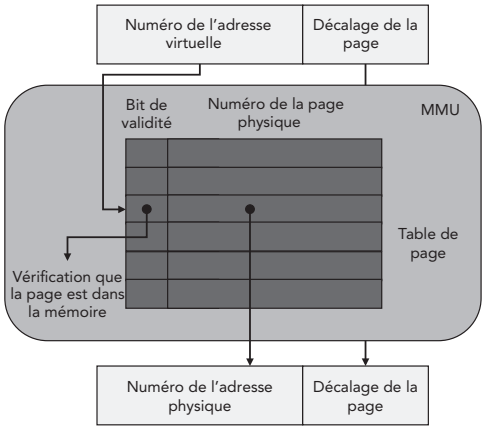

Figure 1.9 Utilisation d'une table de page pour indexer les pages physiques dans la mémoire virtuelle

Table des pages

Le mécanisme de traduction d'adresse de la mémoire virtuelle repose sur l'utilisation d'une table des pages (Figure 1.9). Cette table, au même titre que les registres, fait partie du contexte d'exécution d'un processus et est sauvegardée/restaurée à chaque changement de contexte. C'est le système d'exploitation qui maintient à jour la table des pages de chaque processus. La table de page est stockée dans la mémoire principale, pour chaque processus, le système d'exploitation utilise un registre qui pointe à l'adresse du premier niveau de table des pages. En pratique, plusieurs niveaux de tables des pages sont utilisés : 4 sur x86 et 3 sur des systèmes plus modestes.

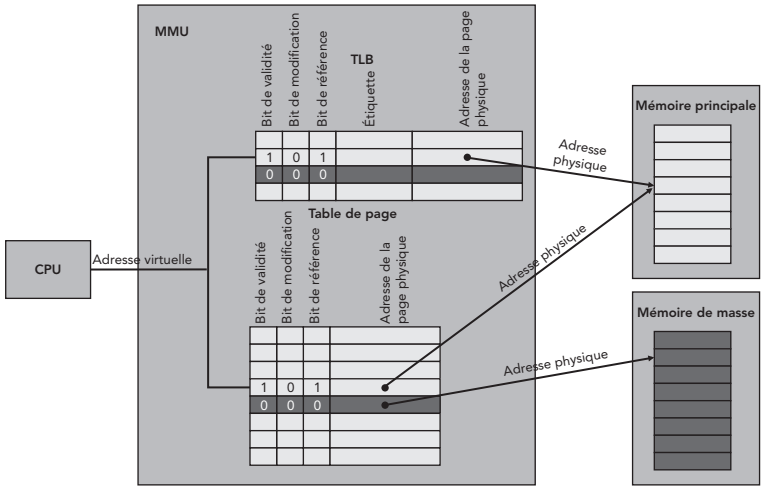

Figure 1.10 Utilisation du TLB comme cache de la table de page

On comprend donc que la traduction d'adresse est une opération lente, car il faut parcourir cet ensemble de tables. Aussi, pour accélérer la translation des pages virtuelles, les processeurs modernes utilisent un cache des traductions récentes d'adresse effectuées appelé TLB. Il mémorise les dernières pages virtuelles utilisées pour une traduction (Figure 1.10). Pour minimiser le taux de *miss*, les systèmes utilisent des TLBs complètement associatifs. Pour la traduction d'une adresse virtuelle, au lieu de traduire directement à l'aide de la table des pages, son numéro est recherché tout d'abord dans le TLB. Dans le cas d'un TLB *hit*, l'adresse physique associée à la bonne étiquette du TLB est utilisée et le bit de référence est mis à la valeur 1. Dans le cas d'un TLB *miss*, le processeur vérifie la table de page à l'entrée indexée par le numéro de page virtuelle. Si l'adresse physique correspondante est dans la mémoire principale, elle est utilisée pour la traduction et chargée dans le TLB et dans le cas contraire, un défaut de page est signalé. Il sera géré par la couche logicielle, en l'occurrence le système d'exploitation.

Intégration de la mémoire virtuelle, TLBs et caches

La mémoire virtuelle et les systèmes de cache fonctionnent ensemble en respectant une hiérarchie précise. Dans un cas idéal, une adresse virtuelle est mappée à une adresse physique (TLB *hit*), cette dernière permet de récupérer la donnée ou l'instruction dans le cache directement (*cache hit*) et de la fournir au CPU. Dans cette organisation, le cache est indexé par une adresse physique et le temps d'accès mémoire est la somme du temps d'accès TLB et du temps d'accès au cache. Une autre alternative est d'enlever le TLB du chemin critique pour réduire la latence du cache. Il s'agit d'indexer le cache par des adresses virtuelles. Le processeur aura besoin de translater l'adresse virtuelle à une adresse physique que dans le cas d'un *cache miss*. Pour cette architecture, quand des pages virtuelles sont partagées par plusieurs processus, il est possible que plusieurs adresses virtuelles pointent sur la même page physique (*aliasing*). Des limitations de conception du cache et du TLB ou le contrôle de la couche logicielle permettent de réduire les risques d'apparition de ce problème. Le compromis entre un cache totalement indexé par des adresses physiques et un cache totalement indexé par des adresses virtuelles est un cache indexé virtuellement et étiqueté physiquement (Figure 1.11).

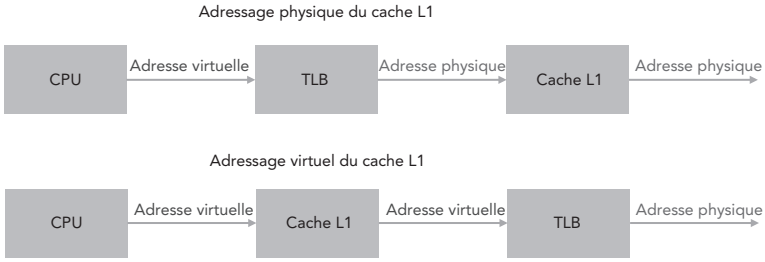

Figure 1.11 Exemple d'indexation virtuelle ou physique du cache

Mécanisme de protection mémoire

Une fonction essentielle de la mémoire virtuelle est la possibilité de gérer finement les droits d'accès et le partage de mémoire avec d'autres processus. La protection fondamentale que doit fournir la mémoire virtuelle est l'isolation de la mémoire entre les processus : assurer qu'un processus partageant la même mémoire principale avec d'autres processus n'a pas la possibilité d'écrire ou de lire dans un espace mémoire adressable par un autre. Comme exemple, le bit de droit d'écriture dans la table des pages permet de marquer des pages en lecture seule. Pour supporter ce modèle d'isolation mémoire, l'architecture matérielle d'un processeur doit prendre en compte plusieurs points qui permettent au

système d'exploitation d'implémenter des protections de la mémoire. Parmi ces points, les trois suivants sont nécessaires :

1. Le processeur doit supporter au moins deux modes qui indiquent si le processus actuel est un processus utilisateur ou processus superviseur (le système d'exploitation, *kernel*). Un processus utilisateur a des droits d'accès limités (i.e., vérifiés par la MMU) à la mémoire physique, alors que le mode noyau à un accès illimité. On a parfois également un niveau supplémentaire (mode hyperviseur) pour isoler différents systèmes d'exploitation exécutés sur la même machine.
2. Une partie de l'état du processeur n'est accessible qu'en lecture pour les processus utilisateurs. Elle inclut le bit de mode (utilisateur/superviseur), le pointeur de la table de page et le TLB.
3. Le passage d'un mode utilisateur à mode superviseur et vice-versa doit être prévu dans la conception matérielle. Par exemple avec l'utilisation d'instructions dédiées.

1.2 Vulnérabilités des SRAMs

La mémoire SRAM (*static random access memory*) ou mémoire vive statique est un type de mémoire volatile utilisant des inverseurs tête-bêche pour mémoriser un bit. La Figure 1.12 présente l'exemple d'une cellule d'une mémoire SRAM. La cellule est sélectionnée par l'adresse du mot WL (*word line*) et par le bit dans le mot BL (*bit line*).

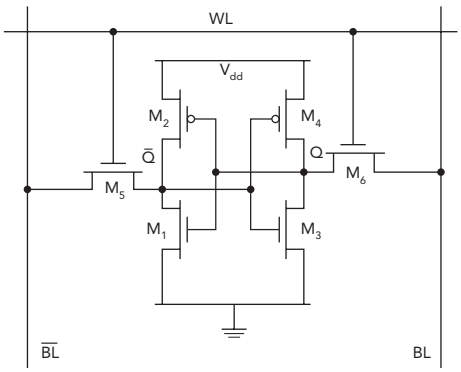

Figure 1.12 Exemple d'une cellule d'une mémoire SRAM en technologie CMOS

À la différence de la mémoire DRAM (*dynamic random access memory*), dont le principe est de mémoriser chaque bit par une charge électrique stockée dans un condensateur, les mémoires SRAM n'ont pas besoin de rafraichissement. Par conséquent, leur temps d'accès est plus rapide (entre 0,5 ns et 2,5 ns) s'appro-

chant de ce fait d'un temps de cycle CPU. La mémoire statique est plus coûteuse, moins dense et plus rapide que la mémoire dynamique. De ce fait, elle est utilisée notamment pour des applications qui nécessitent des temps d'accès rapides ou une faible consommation, en l'occurrence les mémoires caches, les bancs de registres, les tampons et les applications embarquées.

1.2.1 Principe de lecture/écriture

La Figure 1.13 décrit l'opération d'écriture et de lecture pour un schéma classique de cellule à 6 transistors. Pour écrire dans une cellule mémoire, on force la valeur dans la cellule sélectionnée : la *word line* est activée (transistors d'accès AXL et AXR passants), la paire de *bit line* (WBL et son complémentaire WBLB) est forcée à des valeurs CMOS saturées (0 ou 1), ce qui transfère la donnée à l'intérieur de la cellule. Pour la lecture, la paire de *bit line* est laissée flottante et l'ouverture de la *word line* transfère l'information de la cellule sur la paire de *bit line*. Un amplificateur en bout de lignes permet de mesurer la différence de potentiel et de valider la valeur du bit lu. Il est difficile de rendre conjointement ces deux opérations stables parce que les transistors d'accès doivent être surdimensionnés pour une opération d'écriture, alors que l'on cherchera l'inverse pour la lecture. Cette nécessité de compromis peut rendre la cellule susceptible de faire des fautes en lecture ou écriture [79].

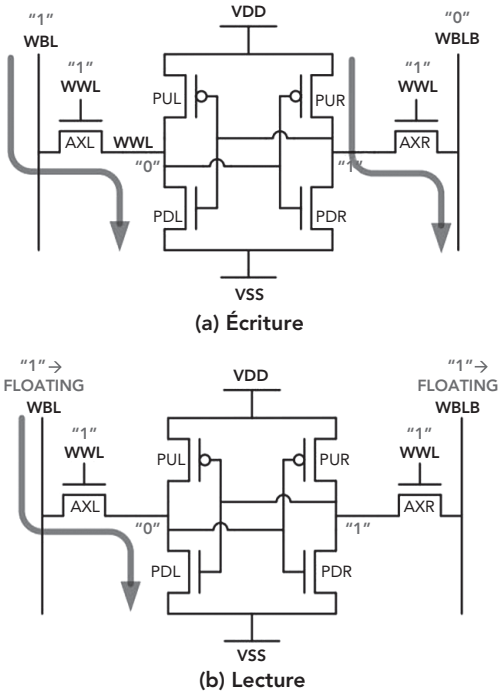

Figure 1.13 Illustrations des opérations de lecture (a) et d'écriture (b) d'une SRAM

1.2.2 Les fautes dans les SRAMs

La marge statique de bruit souvent appelée par son acronyme anglais SNM (*static noise margin*) permet de visualiser la stabilité d'un point mémoire, c'est-à-dire sa capacité à conserver un bit de donnée et à résister aux perturbations. Elle se visualise en traçant sur le même graphique la caractéristique V_{out} (V_{in}) du premier inverseur et la caractéristique V_{in} (V_{out}) du second inverseur. La SNM est alors représentée par la diagonale du plus grand carré qui s'inscrit dans l'une des deux boucles de la courbe en papillon associée au point mémoire (Figure 1.14). L'opération de lecture est plus sensible en matière de conservation de l'information que l'opération d'écriture. De plus, avec la miniaturisation ininterrompue des processus technologiques, on voit désormais émerger des cellules avec des transistors FinFET de largeur de grille de 14 nm voire 7 nm. Ces progrès ont nettement amélioré les performances et la densité des cellules, mais au détriment de leur stabilité et d'un courant de fuite non maitrisé.

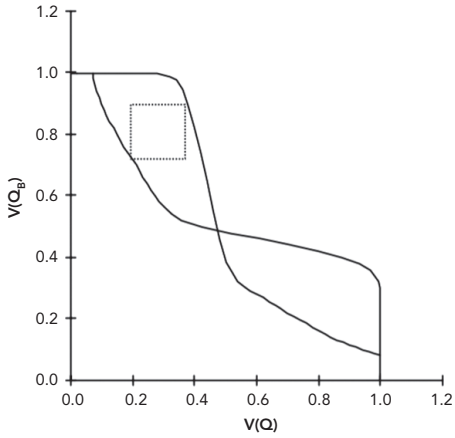

Figure 1.14 Caractéristique d'une cellule SRAM et carré montrant la SNM

Historiquement, les fautes sur les SRAM étaient dans leur grande majorité imputables aux *soft errors,* fautes qui sont des commutations de bits dues à des particules cosmiques de type particules alpha, neutron, rayon cosmique ou radiation externe. Les mémoires cache sont les plus exposées à ce phénomène de par leur densité mais surtout parce qu'elles prennent une grande partie de la surface silicium du processeur [80]. Ces fautes sont d'autant plus présentes que l'altitude de fonctionnement augmente. La densité des SRAMs fait maintenant craindre des fautes multi-bits qui obligent les concepteurs à entrelacer des *bitlines* et à privilégier des architectures à base de cellules à 8 transistors [79 ; 81].

Un article de 2015 [82] a fait la synthèse des fautes qui ont pu se produire dans les mémoires de milliers de CPUs de plusieurs supercalculateurs. Il apparaît en comparant avec des études sous accélérateurs de particules que les *soft errors* sont la grande majorité des fautes que l'on peut observer dans les processeurs de cette génération. Toutefois, cette réalité ne semble à présent plus tout à fait vraie. De plus en plus de fautes sont la conséquence de la miniaturisation qui augmente les couplages entre cellules, de la baisse de la tension d'alimentation et des changements de tension de seuil qui diminue la SNM et donc la stabilité des cellules ainsi que d'une variation plus importante des paramètres de procédés technologiques [83]. Les fautes qui apparaissent alors peuvent prendre les formes suivantes :

- Faute d'accès (*access failure*) : une diminution des courants dans les transistors d'accès AXL et AXR qui déchargent la *bitline* durant l'opération de lecture. Ce mécanisme provoque une moindre différence de potentiel entre les *bit lines* que l'amplificateur ne peut plus évaluer.
- Faute de commutation en lecture (*flipping read failure*) : une perturbation du contenu de la cellule durant la lecture qui provoque le changement de la valeur du bit.
- Faute d'écriture (*write trip voltage WTV fault*) : le retournement de la tension au niveau des inverseurs de la cellule SRAM n'est pas assez important empêchant l'écriture du bit.
- Faute de marge de bruit statique (*static voltage noise margin failure*) : une cellule SRAM peut commuter de façon involontaire avec une marge de bruit trop faible (problème de stabilité).
- Faute de blocage (*stuck-at failure*) : la valeur contenue dans la cellule ne peut être modifiée lors de l'écriture.

Des travaux récents [84] prennent en compte dans les tests de mémoires des fautes statiques ou dynamiques de couplage entre cellules à la fois en lecture et écriture qui peuvent affecter plusieurs bits voire une ligne ou une colonne entière. Elles sont dues à des capacités de couplage entre pistes toujours plus grandes au fur et à mesure que les technologies diminuent d'échelle. Il n'est donc pas exclu de voir apparaître des bugs de type *rowhammer* affectant les DRAMs dans les SRAMs (voir la section 1.5 Attaques sur les mémoires DRAMs) qui pourraient être exploités au niveau des caches.

1.2.3 La gestion des fautes des SRAMs

Afin de gérer ces nouveaux types de fautes, les concepteurs de circuits mettent en place un certain nombre de stratégies. Nous avons déjà vu la nécessité d'effectuer