UNIVERSITÉ ÉVRY VAL D'ESSONNE, CEA

École doctorale S&I
Laboratoire Ibisc – Équipe Cosmo

THÈSE

présentée et soutenue publiquement le 7 juillet 2015
pour l'obtention du grade de

Docteur de l'Université d'Evry Val d'Essonne

Discipline ou Spécialité : Informatique

par

Jordan DE LA HOUSSAYE

MODÈLE DE STOCKAGES DISTRIBUÉS APPLIQUÉ AUX CACHES HIÉRARCHIQUES

COMPOSITION DU JURY

Directeurs :	M. Pommereau Franck
	M. Deniel Philippe
Rapporteurs	Mme Choppy Christine
	M. Sens Pierre
Examinateurs	Mme Klaudel Hanna
	M. Nominé Jean-Philippe

Je tiens à remercier tout particulièrement Mme Choppy et M. Sens pour avoir accepté d'être rapporteurs malgré des contraintes de temps difficiles.

Je remercie également Mme Klaudel et M. Nominé d'avoir accepté d'être membres du jury, ainsi que mes directeurs, MM. Deniel et Pommereau.

Merci à Mme Atmaca et M. Lucas pour m'avoir recommandé auprès du CEA.

Je remercie enfin ma famille et mes amis de m'avoir supporté durant ma thèse.

Résumé

Les systèmes de cache sont omniprésents dans les systèmes informatiques modernes, qu'ils soient embarqués dans le matériel (processeurs, contrôleurs de disques, ...) ou le logiciel (systèmes de fichiers, bases de données, serveurs web, ...). Une grande quantité de travaux ont été réalisés pour concevoir, implémenter et optimiser ces systèmes. Cependant il n'existe pas à notre connaissance de conception universelle de ce qu'est et réalise un cache, ni donc d'outil permettant la conception et la simulation de système définis ainsi. Dans cette thèse nous proposons un tel point de vue générique et universel sur les caches et nous développons un nouveau modèle de cache allant dans cette direction. Un simulateur implémentant ce modèle est réalisé et des exemples sont développés afin d'illustrer l'usage du modèle et d'en valider la pertinence.

Mots-clés : caches, modèle formel, simulation

Abstract

Caches' systems are ubiquitous in modern computer systems, either embedded in hardware (processors, disks controlers, ...) or in software (file systems, databases, web servers, ...). A large amount of work has been dedicated to design, implement and optimise those systems. However there exists to the best of our knowledge no universal conception of what is and what achieves a cache, and therefore no tool allowing the conception and the simulation of systems defined this way. In this thesis we propose such a generic and universal point of view on caches and we develop a new cache model toward this direction. A simulator implementing this model is realised and examples are developped in order to illustrate the usage of the model and validate its relevance.

Keywords: caches, formal model, simulation

Table des matières

Résumé 3

I Introduction 7

1 *Introduction* 9

II Contribution 25

2 *Les éléments structurants* 27

3 *Modèle formel* 35

4 *Expériences* 53

III Ouvertures 69

5 *Discussions* 71

6 *Conclusion* 83

Annexes 85

A *Rappels sur les réseaux de Petri* 87

B *Implémentation* 93

Bibliography 97

Première partie
Introduction

1 *Introduction* 9
1.1 Généralités 9
1.2 Problématiques fondamentales 13
1.3 Problématiques distribuées 19
1.4 Problématiques latentes 22
1.5 Ambitions de la thèse 23

« *Nous sommes donc forcés de reconnaître la possibilité de construire une hiérarchie de mémoires, chacune ayant une plus grande capacité que la précédente mais étant moins rapidement accessible.*

. . . . . .

We are therefore forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible. »

A. W. Burks, J. H. Goldstine, J. von Neumann, *Preliminary Discussion of the Logical Design of Electronic Computing Instrument,* Part I, Vol. I, report prepared for U.S. Army Ord. Dept., 28 June 1946

1
Introduction

L'étude des caches est un champ de développement actif depuis plusieurs décennies qui s'est développé à partir des travaux sur la mémoire virtuelle dans les années soixante. Aujourd'hui les caches sont omniprésents dans les environnements informatiques, qu'ils soient embarqués dans du matériel (processeurs, contrôleurs de disque, ...) ou du logiciel (systèmes de fichiers, bases de données, serveurs web, ...).

La littérature sur le sujet des caches est très prolifique et en même temps disparate. À mesure qu'elle s'intéresse à des aspects différents du sujet, nos connaissances se spécialisent et se fragmentent. Il n'existe pas à notre connaissance de manière standard de traiter de l'ensemble des problématiques soulevées par les caches. Bien que les problématiques s'y répètent, à chaque contexte particulier sont attachés des termes et des méthodes adaptées. Si cette situation est entièrement justifiée, elle empêche à notre avis l'existence d'une vision d'ensemble sur le sujet.

Dans ce chapitre nous verrons ce que recouvre cette notion, les problématiques que ces systèmes tentent de résoudre et celles qu'ils rencontrent.

1.1 *Généralités*

Des mémoires virtuelles aux caches

Historiquement, les grandes mémoires sont complexes et lentes, mais bon marché. Les mémoires rapides quant à elles sont simples mais chères et de petite taille.

C'est la raison pour laquelle on a commencé à adjoindre à une petite mémoire principale une grande mémoire auxiliaire dans les premières générations de calculateurs. Hélas la gestion des transferts entre ces deux mémoires se faisait de manière manuelle par le programmeur, ce qui est fastidieux.

C'est pour cela que les premières mémoires virtuelles ont été conçues[1]. Le sujet était d'automatiser ces transferts, recréant ainsi un système de mémoire à un seul niveau du point de vue du programmeur, mais étant composé de deux niveaux en réalité [1].

[1] Kilburn et al.: "One-Level Storage System", 1962.

1 Les systèmes de mémoires virtuelles ont beaucoup évolué depuis, et ils recouvrent bien d'autres fonctionnalités que nous ne traiterons pas ici.

Plus tard, on a cherché à faire le chemin inverse : intercaller entre un processeur et sa mémoire une petite mémoire rapide où stocker et manipuler ses données avant de les re-transferer dans la mémoire principale. On a appelé cette mémoire intermédiaire un *cache*.

Bien que la forme soit différente, le principe des caches est similaire à celui des mémoires virtuelles des origines. Une mémoire de taille modeste s'intercalle entre un processeur et une mémoire de taille importante ; les transferts entre ces deux mémoires se font automatiquement.

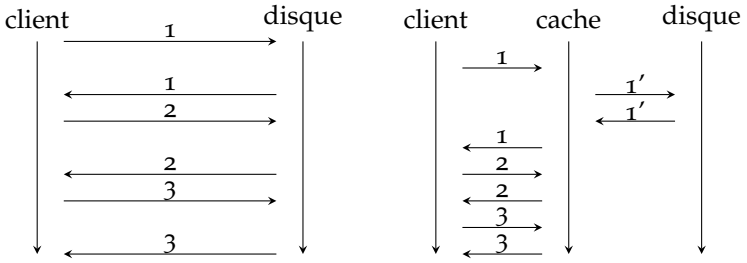
On a ensuite repris ce principe au sein des contrôleurs de disques, des bases de données et des systèmes de fichier, etc. On appelle généralement ces caches des *caches de données*[2]. Dans la suite nous ne traiterons des caches que du point de vue des caches de données, c'est-à-dire d'un point de vue logique.

[2] ROBINSON and DEVARAKONDA: "Data cache management using frequency-based replacement", 1990.

Caches et tampons

Un *cache* est un dispositif comprenant une mémoire lui permettant de répondre à des requêtes normalement destinées à un autre dispositif à la place de celui-ci.

On espère généralement que le cache sera plus rapide que le dispositif à la place duquel il répond, et ainsi accélerer le système. Il est bien sûr nécessaire que ces deux dispositifs communiquent entre eux afin que les résultats soient corrects. Considérons les schémas suivants :

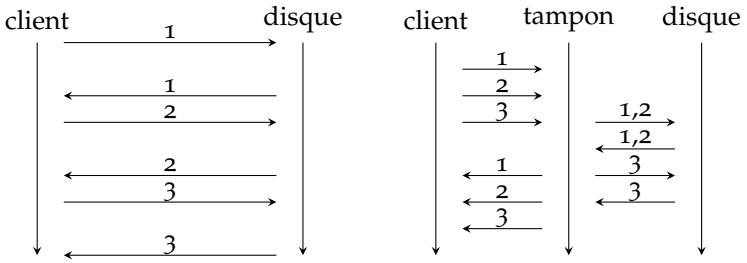


À gauche on a représenté un système dans lequel un client envoie des requêtes à un disque qui lui envoie des réponses. À droite on a représenté un système similaire dans lequel on a ajouté un cache entre le client et le disque. Afin de représenter la capacité du cache à répondre plus vite que le disque, on l'a disposé à mi-distance du client sur le dessin. Ce n'est bien sûr qu'une représentation graphique et ne doit pas être pris littéralement.

D'une manière générale un cache cherchera à garder dans sa mémoire les informations les plus utiles à court terme afin de répondre le plus possible à la place du dispositif qu'il cache. Dans cette illustration le cache a permis de répondre à deux requêtes sur trois à la place du disque. On utilise généralement les caches pour compenser une lenteur, que ce soit celle du dispositif qu'on cache ou celle du réseau.

Un *tampon* (ou *buffer* en anglais) est comme le cache un dispositif intermédiaire. La différence est qu'il régule le rythme et/ou l'ordre

dans lequel les requêtes sont transmises au disque. Contrairement au cache il n'aura donc pas tendance à réduire le nombre de requêtes transmises.



Dans cette illustration, le tampon a compacté les requêtes 1 et 2 pour le disque, ce qui a effectivement réduit la durée nécessaire pour répondre aux trois requêtes du client.

D'une manière générale, un tampon cherchera à optimiser les requêtes de manière à obtenir un gain sur la durée globale d'exécution. Il peut se permettre pour cela d'être plus lent localement. On utilise généralement les tampons pour compenser une différence de rythmes entre des dispositifs et/ou pour maximiser des débits.

Dans les systèmes on peut trouver des tampons sans cache ou des caches sans tampon. Cependant on les rencontre souvent ensemble, à tel point que les deux concepts, pourtant clairement différents, sont souvent amalgamés dans la littérature.

Par exemple le site d'IBM nous propose la définition suivante pour le terme *cache* (ci-contre en version originale [2]) :

> *[Un cache est une] mémoire rapide ou un dispositif utilisé pour réduire le temps effectif requis pour lire une donnée de ou écrire une donnée vers une mémoire plus lente ou un dispositif. Un cache de lecture retiens de la donnée en anticipant sa requête par un client. Un cache d'écriture retiens de la donnée écrite par un client jusqu'à ce qu'elle puisse être stockée de manière sûre sur un stockage plus permanent tel qu'un disque ou une bande.*

C'est une définition assez large qui recouvre en fait les deux notions de *cache* et de *tampon*. En effet ce qui est appelé ici «cache d'écriture» ressemble plutôt à ce que nous avons défini plus haut comme un tampon, puisque la requête est maintenue dans le dispositif «jusqu'à ce qu'elle puisse être stockée de manière sûre» sur son dispositif cible. Ceci étant, les caches d'écriture ont une particularité que n'ont pas les tampons : d'abord ils utilisent la nouvelle valeur correspondant aux données écrites pour répondre aux lectures[3]; ensuite si plusieurs écritures surviennent sur les mêmes données avant que les requêtes n'aient été transmises, seule la dernière d'entre elles le sera. De cette manière on aura bien réduit le nombre d'écritures sur le dispositif caché en plus d'avoir réorganisé le rythme et/ou l'ordre des écritures.

Ainsi s'il est important de savoir faire la distinction entre caches et tampons, il faut bien noter que ces dénominations ne sont pas toujours pertinentes dans le cadre qui nous intéresse.

2 [A cache is a] high-speed memory or storage device used to reduce the effective time required to read data from or write data to lower-speed memory or a device. Read cache holds data in anticipation that it will be requested by a client. Write cache holds data written by a client until it can be safely stored on more permanent storage media such as disk or tape.
— `http://www-01.ibm.com/support/knowledgecenter/STPVGU/com.ibm.storage.svc.console.510.doc/mlt_glossary_1b1crp.html?lang=ko`

[3] Gill et al.: "STOW : a spatially and temporally optimized write caching algorithm", 2009.

Un problème de langage

Nous voudrions nous arrêter un instant sur le point qui vient d'être abordé. Les lignes ci-dessus illustrent un problème que nous avons identifié dans la littérature : il ne semble pas exister de concensus au sujet de la terminologie utilisée dans le domaine des caches, ni de conception uniforme de ce qu'est un cache. Par exemple un même mot sera utilisé pour désigner des concept différents ou au contraire des mots différents désigneront le même concept, comme c'est le cas pour les termes tampons et caches. Un autre exemple est la distinction systématique qui est faite entre cache d'écriture et cache de lecture, comme dans la définition d'IBM plus haut. Ce genre de distinction échoue à expliquer ce qui est commun à ces deux types de cache, et donc à expliquer ce qu'est un cache.

Pourtant on rencontre des caches dans virtuellement tous les systèmes et sous-systèmes informatiques modernes. On pourrait donc s'attendre à ce que ces concepts soient depuis longtemps clairement définis. Or, lorsqu'on cherche à rendre compte de ce qu'est un cache sans être réducteur on se trouve contraint d'écrire un compte rendu exhaustif de la littérature. On perd à cette occasion la clarté et le recul nécessaire à l'objectif initial : répondre à la question «Qu'est-ce qu'un cache ?». De plus il est difficile de comparer les approches, de les mettre en opposition, ou de voir comment elles s'articulent, faute d'avoir une conception homogène du domaine.

Ce phénomène s'explique précisément par le caractère ubiquitaire des caches. D'abord c'est une technique employée par tous les systèmes qu'on peut rencontrer. Le domaine des caches est donc en fait partagé entre des communautés très diverses ayant des contraintes et des objectifs très variés. De plus ces contraintes et objectifs doivent être adaptés constamment à l'évolution rapide des technologies. Il est donc naturel dans ce contexte que le langage et les techniques ne soient ni complètement partagées ni complètement homogènes dans le temps.

Dans la suite nous allons justement tenter de donner au lecteur la vision globale et cohérente de l'ensemble des problématiques associées aux caches que nous avons cherché en vain lorsque nous avons débuté cette thèse. De cette manière ce chapitre est simultanément une introduction au domaine des caches, une tentative de mise en cohérence du domaine, et le point de départ de la problématique développée dans ce mémoire.

Objectif et nature des caches

On s'accorde généralement à dire que l'objectif numéro 1 des caches est de réduire le temps moyen de résolutions des requêtes pour les utilisateurs. Dans le cas des caches de lecture cela passe par l'augmentation de l'utilité marginale des ressources qu'ils contiennent [4,5]. Autrement dit, un cache cherche à maximiser l'utilisation de chacune des ressources qu'il gère. Dans le cas des caches d'écriture, cela passe par la recherche de l'ordre et du rythme idéal d'écritures

[4] KIM et al.: "A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references", 2000.

[5] GILL and BATHEN: "AMP : adaptive multi-stream prefetching in a shared cache", 2007.

sur le stockage [3,6]. Autrement dit, le cache assume en plus un rôle de tampon afin de mieux répondre à sa problématique principale.

Cela nous évoque deux choses. La première c'est qu'afin de remplir son objectif, le cache a à sa disposition plusieurs régimes d'exploitation possibles. La seconde est que ces régimes d'exploitation cherchent d'une part à capturer une certaine forme de régularité dans la distribution des accès, et d'autre part à faire conserver au cache une forme de régularité dans son comportement.

Le point précédent est important car il suggère l'existence de plans stratégiques sous-jacent aux techniques de gestion de caches. Or si la littérature est familière avec ces éléments, elle ne les considère pas en tant qu'objet de premier ordre. D'une manière générale, on ne traîte que rarement des caches en tant qu'objet et on ne s'intéresse souvent qu'à l'aspect technique des solutions. Ce faisant, il nous semble que la nature des caches reste à définir et qu'alors de nouvelles classes de solutions seront accessibles.

[6] Gill and Modha: "WOW : wise ordering for writes - combining spatial and temporal locality in non-volatile caches", 2005.

La section suivante tente d'approcher une telle définition en proposant une vue globale et stratégique du fonctionnement des caches.

1.2 *Problématiques fondamentales*

Dans cette section nous allons tenter d'organiser logiquement les différents aspects que recouvrent les politiques. Il y a eu par le passé des articles allant dans ce sens[3,7–9]. Cependant ils s'intéressaient la plupart du temps à comparer des performances par rapport à des approches plutôt qu'à caractériser le fonctionnement d'un cache. Ici nous proposons une grille de lecture générique afin d'en obtenir une vue d'ensemble. Cette grille sera augmentée par la suite quand on s'intéressera aux problématiques distribuées.

Nous divisons notre grille de lecture en trois axes : 1. les stratégies (ou les régimes) d'exploitation, 2. les stratégies de capture des régularités, et 3. les moyens de la capture.

Notons que cette approche ne se substitue pas à l'étude des techniques employées dans les caches. Au contraire elle cherche à donner un nouvel éclairage sur les-dites techniques dans l'optique de mieux en comprendre les enjeux. Pour cette raison nous nous intéressons à la stratégie globale qu'une politique de cache peut adopter. Nous ignorons volontairement les techniques employées par ces politiques car, bien qu'elles soient importantes, elles sont selon nous l'arbre qui cache la forêt. Le troisième axe présenté ne sera donc pas développé ici.

Notons de plus qu'une politique peut très bien se situer à différents points de chacun de ces axes. Il ne faut donc pas les comprendre de manière linéaire.

[7] Biswas, Ramakrishnan, and Towsley: "Trace driven analysis of write caching policies for disks", 1993.

[8] Karedla, Love, and Wherry: "Caching strategies to improve disk system performance", 1994.

[9] Chen et al.: "Empirical evaluation of multi-level buffer cache collaboration for storage systems", 2005.

Stratégies d'exploitation

Dans la stratégie d'exploitation on peut s'intéresser à trois temps :

1. le moment où la donnée entre dans le cache ;
2. le moment où la donnée sort du cache ;
3. le moment où la donnée est synchronisée, ou mise en cohérence, avec le stockage.

Nous allons classer les stratégies d'exploitation selon ces trois axes. Afin de s'y référer plus tard, on associera à chaque stratégie un nom dans la marge. Lorsque ces stratégies correspondent à des catégories connues, on précisera leur nom commun.

On leur associera des politiques que l'on rencontre dans la littérature. Si ces politiques implémentent selon nous les stratégies auxquelles on les associe, leurs auteurs respectifs ne les ont pas nécessairement pensées comme telles. De plus la liste des politiques citées ne sera pas exhaustive.

Entrée de la donnée dans le cache On peut adopter plusieurs stratégies pour faire entrer la donnée dans le cache [3].

EI1 La stratégie la plus courante est de faire rentrer la donnée dès qu'une requête cherche à y accéder. Cela suit l'hypothèse selon laquelle un accès à une donnée implique un accès prochain à cette même donnée (voir plus bas la localité temporelle). Cette stratégie porte le nom de «pagination à la demande» (ou *demand paging* en anglais). C'est la stratégie de base appliquée par défaut ; s'il n'existe pas de raison de penser que la donnée ne sera pas réutilisée (cf. EI3) on applique EI1.

EI2 Une deuxième stratégie consiste à faire rentrer une donnée en même temps qu'une autre donnée qui lui soit liée (voir plus bas la localité spatiale). Cela suit l'hypothèse selon laquelle l'accès à la première implique un accès prochain à la seconde. Cette stratégie porte le nom de «pré-pagination» (ou *prefetching* en anglais). On peut citer SARC [10] et AMP [5] pour cette stratégie.

EI3 Une troisième statégie, complémentaire à EI1, est de réaliser un filtre à l'entrée. S'il existe une manière de supposer à l'avance qu'une donnée ne sera pas réutilisée, cette stratégie dicte de ne pas la faire entrer dans le cache. On peut citer DEMOTE [11] et PROMOTE [12] par exemple, ou bien les politiques à base d'indices (cf. CU6).

Sortie de la donnée du cache À l'instar de l'entrée, on peut adopter plusieurs stratégies concernant la sortie des données du cache [4].

EO1 La stratégie la plus courante est de faire sortir la donnée (on dit «évincer») dès lors que l'espace qu'elle occupe est nécessaire pour une donnée jugée plus utile. C'est la stratégie par défaut employée dans les caches puisqu'elle répond à une nécessité : faire de la place. La notion d'utilité est définie par la stratégie de capture. On parle dans la litérature de «remplacement» (*replacement* en anglais).

EO2 Une deuxième stratégie consiste à évincer une donnée lorsque le coût de cette éviction est le plus bas. Cette question se pose lorsqu'on se place dans un cache d'écriture et qu'une donnée est en état «asynchrone» par rapport au stockage. Dans ce cas le principe de cohérence impose qu'elle soit synchronisée avant d'être évincée, hors

3 Les stratégies seront nommées EI*n*. E pour *Exploitation*, I pour *Input* et *n* est son numéro.

[10] GILL and MODHA: "SARC : sequential prefetching in adaptive replacement cache", 2005.

[11] WONG and WILKES: "My Cache or Yours ? Making Storage More Exclusive", 2002.

[12] GILL: "On multi-level exclusive caching : offline optimality and why promotions are better than demotions", 2008.

4 Les stratégies seront nommées EO*n*. E pour *Exploitation*, O pour *Output* et *n* est son numéro.

cette synchronisation à un coût. Cette stratégie est donc intimement liée à la stratégie de synchronisation des données ci-après. Elle est donc aussi liée à une notion de coût de synchronisation capturé par la stratégie de capture.

E03 Une troisième stratégie consiste à maintenir dans le cache une certaine proportion d'espace libre, de manière souple. De cette manière on espère être capable d'absorber des variations dans le rythme des requêtes. Cela permet au stockage de ne pas subir ces variations, et donc à l'utilisateur de ne pas subir de ralentissement lié à ces variations. Cette stratégie devra donc capturer la notion d'espace disponible. On peut citer WOW [6] et STOW [3].

Synchronisation de la donnée Une fois encore, on peut adopter plusieurs stratégies concernant le temps de la synchronisation des données entre le cache et son stockage [5].

5 Les stratégies seront nommées ES*n*. E pour *Exploitation*, S pour *Synchronize* et *n* est son numéro.

ES1 Une première stratégie est de synchroniser les données au fur et à mesure qu'elles sont modifiées. Autrement dit le cache écrit d'abord sur le stockage avant de modifier sa copie locale. Cette stratégie porte le nom d'«écriture au travers» (ou *write through* en anglais). Elle a le mérite d'assurer le principe de cohérence, puisque le cache ne peut jamais avoir de donnée non synchronisée avec le stockage. Cependant, c'est la stratégie la plus coûteuse puisque toutes les écrtures sont transmises au stockage.

ES2 Une deuxième stratégie consiste à synchroniser uniquement lors de l'éviction. C'est la stratégie la plus courante lorsqu'on a accès à un espace non volatile qui permette de récupérer la dernière valeur des données en cas d'arrêt du système. Elle porte le nom de *write back*. Cette stratégie à le mérite de réduire au maximum le nombre d'écritures qu'il est nécessaire de transmettre au stockage. Nous reviendrons sur cette réduction dans une section ultérieure.

ES3 Ces deux stratégies sont simples. Ce sont celles que l'on rencontre dans la littérature. Une troisième stratégie consiste à synchroniser la donnée lorsque le coût de cette synchronisation est le plus bas. Par exemple en même temps que la synchronisation d'une donnée voisine dans le cas d'un stockage sur disque. Cela porte le nom d'«écriture à coté» (ou *write behind* en anglais). Cette stratégie nécessite de capturer la notion de localité spatiale. On peut se référer aux politiques *write behind* et *write behind with threshold* dans [7].

Stratégies de capture

Les stratégies d'exploitation nous ont déjà permis d'évoquer les trois axes stratégiques suivants :

1. l'utilité des données ;
2. le coût d'opportunité ;
3. le rythme d'accès au stockage.

Le positionnement de la politique sur ces axes stratégique sont liés avec son positionnement sur les axes de la stratégie d'exploitation.

C'est ce qui permet à la politique soit de cibler des ressources (qui doit rentrer, sortir, ou être synchronisée), soit de cibler des moments (quand faire rentrer, sortir ou synchroniser).

Chacun de ces axes nécessitera de capturer les éléments suivants : 1. la structure des accès, 2. la structure des données, et 3. l'espace utilisé.

Utilité des données Le cache ayant généralement à disposition un espace plus limité que son stockage, il doit faire un choix quant aux données qu'il maintient (et donc aussi quant à celles qu'il ne maintient pas). Sa stratégie numéro 1 consiste donc naturellement à ne garder que les données les plus utiles.

On peut définir l'utilité d'une donnée pour le cache de différentes manières. Cependant la logique derrière cette notion est qu'il existe la plupart du temps une certaine forme de régularité dans les accès aux données. Plus précisément il existe un principe nommé «localité temporelle» selon lequel il existe une relation entre les différents accès à une même donnée dans le temps.

La littérature identifie deux archétypes : 1. les distributions de type SDD (*Stack Depth Distribution*) et 2. les distributions de type IRM (*Independent Reference Model*).

Dans une distributions de type SDD un bloc a une probabilité de subir un nouvel accès inversement proportionnelle à l'âge de son dernier accès. Autrement dit, un bloc récemment lu (ou écrit) aura plus de chance d'être relu (ou réécrit) qu'un bloc lu précédemment[3].

Les distributions de type IRM sont un schéma d'accès dans lequel les références sont des variables aléatoires indépendantes suivant une distribution fixe. Autrement dit, il n'est pas possible d'exhiber de localité temporelle dans ce cas. En revanche la notion de fréquence capture parfaitement ce schéma[13,14].

Entre les distributions SDD et IRM il existe une infinité de cas [15]. Cela fait de ces deux distributions deux extrêmes entre lesquels on peut se placer pour déterminer la distribution représentant le mieux la réalité.

CU1 La première stratégie[6] consiste à capturer la localité temporelle sur l'axe SDD-IRM. Celà peut aller de la limite SDD (ce que fait LRU : *Least Recently Used* [16]) à la limite IRM (ce que fait LFU : *Least Frequently Used* [14]) en passant par tout le spectre (ce que fait LRFU : *Least Recently/Frequently Used* [15,17]). Dans cette stratégie l'utilité est définie explicitement. Les données sont ordonnées de manière à pouvoir cibler à tout moment celles qui doivent sortir ou rester. Nous avons déjà cité LRU et LFU ; nous pouvons ajouter toutes leurs variantes telles que FBR [2], EELRU [18], LRU/K [19], etc.

CU2 La deuxième stratégie est de capturer la localité spatiale. Dans les systèmes à base de blocs (mémoire, disques, ...) les données sont physiquement structurées de manière linéaire. Lorsque l'application ou l'utilisateur cherche à accéder à ses données (par exemple un fichier) il accède en réalité à une séquence de blocs plus ou moins

[13] COFFMAN and DENNING: *Operating Systems Theory*, 1973.

[14] AHO, DENNING, and ULLMAN: "Principles of Optimal Page Replacement", 1971.

[15] LEE et al.: "On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies", 1999.

6 Les stratégies seront nommées CU*n*. C pour *Capture*, U pour *Usefulness* et *n* est son numéro.

[16] BELADY: "A study of replacement algorithms for a virtual-storage computer", 1966.

[17] LEE et al.: "LRFU : A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies", 2001.

[18] SMARAGDAKIS, KAPLAN, and WILSON: "EELRU : simple and effective adaptive page replacement", 1999.

[19] O'NEIL, O'NEIL, and WEIKUM: "The LRU-K page replacement algorithm for database disk buffering", 1993.

contigus dans la mémoire ou sur le disque. C'est ce qu'on appelle la «localité spatiale». Autrement dit dans ce schéma d'accès si un bloc b est lu, on peut facilement prédire que le bloc $b+1$ sera le prochain bloc lu. Ce schéma nous permet d'intégrer dans notre connaissance des données prochainement utiles de nouvelles données. La stratégie d'exploitation EI2 devient accessible. La plupart des caches d'écritures cherchent à captuer cette localité, en particulier lorsqu'ils sont destinés à être embarqués dans des contrôleurs de disque. On peut citer DULO [20], SARC [10], WOW [6], AMP [5], ou encore STOW [3].

[20] JIANG et al.: "DULO : an effective buffer cache management scheme to exploit both temporal and spatial locality", 2005.

CU3 Une troisième stratégie liée à la précédente consiste à considérer les accès séquentiels comme ayant une faible localité temporelle. Autrement dit on peut supposer qu'une lecture séquentielle à de fortes chances d'être à usage unique, comme on peut l'observer dans des usages de lecture de vidéo par exemple. Dans ce cas il peut être judicieux de ne pas s'encombrer trop longtemps des blocs déjà lus. Cette stratégie cherchera donc à classer rapidement ces schémas d'accès dans la catégorie «faible utilité». C'est ce que font les politiques LIRS [21], ARC [22], UBM [4]ou encore DULO [20].

[21] JIANG and ZHANG: "LIRS : an efficient low inter-reference recency set replacement policy to improve buffer cache performance", 2002.

[22] MEGIDDO and MODHA: "ARC : A Self-Tuning, Low Overhead Replacement Cache", 2003.

Les accès séquentiels vus ci-dessus sont une schéma d'accès courant. Les boucles d'accès en sont un autre. Dans ce dernier un bloc ou une séquence de blocs subit des accès répétés de manière régulière dans le temps. Cette régularité peut être courte ou longue.

CU4 Dans le cas court, cela peut entraîner un obscurcissement des autres schémas. La quatrième stratégie consiste donc à détecter ces boucles courtes afin de mitiger leur effet négatif sur la notion d'utilité. On peut classer dans la catégorie des boucles courtes les accès successifs à une même donnée dans le cadre de la même opération. Dans ce cas la localité temporelle détectée peut être considérée comme un faux positif et être ignorée. Les politiques FBR [2], LRU/K [19], UBM [4], LIRS [21], ou encore ARC [22], suivent cette logique.

CU5 Dans le cas long, ce sont les autres schémas qui obscurcissent ce dernier. Afin de prendre en compte cette localité temporelle, la cinquième stratégie consiste à détecter les boucles longues et à leur donner une chance. On rencontre ce schéma dans des applications de type systèmes de fichiers ou bases de données, au sein desquels il est fréquent de parcourir la racine du système. Ces accès fréquents peuvent rapidement être cachés par des accès plus récents mais de moindre utilité. Ce sont des points traîtés dans LRU/K [19], EELRU [18], UBM [4], LIRS [21], et ARC [22].

CU6 Il existe une autre manière pour le cache d'obtenir l'utilité des données. La sixième stratégie consiste à recevoir ces informations de la part de l'utilisateur lui-même sous la forme «d'indices» (*hints* en anglais). Cela suppose que l'utilisateur soit 1. capable de transmettre ces informations lors de ses requêtes et 2. en mesure de connaître précisément l'ensemble de ses accès. Du fait de ces difficultés cette stratégie est moins courante. On peut tout de même citer les politiques suivantes : *informed prefetching and caching*[23], TQ [24], CLIC [25], et HINTK [26].

[23] PATTERSON et al.: "Informed prefetching and caching", 1995.

[24] LI et al.: "Second-tier cache management using write hints", 2005.

[25] LIU et al.: "CLIC : client-informed caching for storage servers", 2009.

[26] WU et al.: "Hint-K : An Efficient Multi-level Cache Using K-Step Hints", 2010.

Coûts d'opportunités Dans le deuxième axe de la stratégie de capture on s'intéresse aux opportunités d'accès au stockage[7].

7 Les stratégies seront nommées CT*n*. C pour *Capture*, T pour *Timeliness* (opportunité) et *n* est son numéro.

CT1 La première stratégie de cet axe est de profiter de la lecture d'un bloc pour en lire deux ou plus. Elle est liée à la capture des accès séquentiels vue précédemment et à la stratégie EI2. Ici ce qui nous intéresse c'est l'existence de l'opportunité et non l'existence de la structure séquentielle.

CT2 La seconde stratégie consiste à tirer parti de l'écriture d'un bloc pour en synchroniser (voire évincer) deux ou plus. Elle est là encore liée à la capture de la localité spatiale, mais ici on veut profiter de l'opportunité d'un accès pour réaliser d'autres accès qui lui sont proches. Cette stratégie se fonde sur le fait qu'un accès unique sur une séquence de blocs est moins coûteux qu'une séquence d'accès sur chacun de ces blocs, et beaucoup moins coûteux qu'autant d'accès mais sur des blocs non liés par la localité spatiale.

Autrement dit, pour les deux stratégies précédentes, c'est l'hypothèse du moindre coût d'un accès sur une séquence qui justifie l'usage des lectures et écritures anticipées (respectivement) de la stratégie d'exploitation. C'est ce point qui justifie l'existence de cet axe stratégique. Notons que la première stratégie ci-dessus se justifie en conjonction avec l'hypothèse de l'utilité des blocs liés dans l'axe précédent (CU2).

[3]comporte une très bonne synthèse de cette approche.

CT3 La troisième stratégie consiste à réordonner les écritures de manière à exhiber un comportement plus séquentiel pour le stockage. Cette stratégie à pour effet de réduire le coût moyen de ces synchronisations. Si elles sont accompagnées d'évictions, cela vient contredire la stratégie d'utilité. Cette possibilité n'est donc valable que pour les systèmes dans lesquels des accès aléatoires sont très coûteux. WOW [6]et STOW [3]sont deux politiques très avancées sur ce point.

Rythme d'accès au stockage Dans cet axe, on s'intéresse au rythme d'arrivée des requêtes sur un cache par rapport au rythme de sortie des requêtes du cache. Cet axe est lié à la stratégie EO3[8].

8 Les stratégies seront nommées CR*n*. C pour *Capture*, R pour *Rythm* et *n* est son numéro.

CR1 La première stratégie consiste à absorber ce qu'on pourrait appeler des séquences d'accès frénétiques. Il sagit de périodes dans la vie du cache dans lesquelles le rythme des accès augmente violemment pour une période courte. Cela peut se produire dans des systèmes partagés auxquels les utilisateurs ont tendance à accéder au même moment. Afin de réaliser cette absorption, la stratégie consiste à maintenir une certaine proportion du cache vide. De cette manière les accès en échec ne déclencheront pas d'éviction, et donc pas de synchronisation qui s'ajouterait au coût total de la requête. Ici c'est donc l'espace libre du cache qu'il faut capturer. Une fois encore on citera STOW [3], qui est une politique de cache très avancée.

CR2 La deuxième stratégie consiste à lisser les écritures de type synchronisation sur le stockage. De cette manière on tente de lui éviter la situation décrite pour la stratégie CR1. À cette fin la stratégie consiste à adapter le rythme des écritures au taux d'occupation du cache. Plus

le cache est rempli, plus le rythme doit être élevé. Parcequ'on commence à synchroniser avant que le cache n'y soit contraint, le rythme des écritures sur le stockage est lissé. On citera AWOL [27], et encore un fois STOW.

[27] BATSAKIS et al.: "AWOL : an adaptive write optimizations layer", 2008.

Synthèse et ouvertures

Les stratégies présentées ci-dessus couvrent une grande partie de la littérature, cependant il existe probablement des stratégies nous ayant échappées. Ce qui nous importait était de présenter au lecteur un point de vue global sur l'ensemble de ce que cherchent à réaliser les caches.

Ceci étant il reste quelques zones d'ombre que nous n'avons pas encore traîtées. La section suivante s'attache à compléter cette grille de lecture avec le cas des caches distribués.

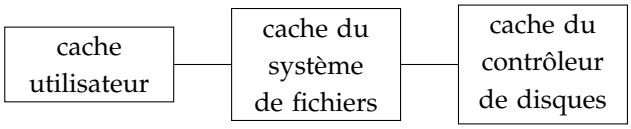
1.3 *Problématiques distribuées*

La section précédente nous a permis de traîter de ce que nous avons appelé les «problématiques fondamentales» du cache. La première d'entre elles était la réduction du temps moyen de résolution des requêtes. Nous avons vu que cette problématique entraînait d'une part la recherche de la maximisation de l'utilité de ce que le cache contient, et d'autre part la recherche de la régularisation du comportement du réseau pour le stockage. Nous avons vu que cela passait par l'existence de plans stratégiques sous-jacents à la capture et à l'exploitation de structures dans les accès et les données.

Topologies

Jusqu'ici nous ne nous sommes intéressé au cache que comme un objet isolé, coincé entre un utilisateur et un stockage. Cependant la littérature s'intéresse à deux autres formes, ou topologies : les topologies que nous appelerons «linéraires» et celles que nous appellerons «arborescentes».

Topologies linéraires C'est le type de topologie que l'on obtient lorsqu'on empile les services dans un système. Par exemple si un utilisateur à accès à un système de fichier sur le réseau, on aura une topologie qui ressemble à ceci :

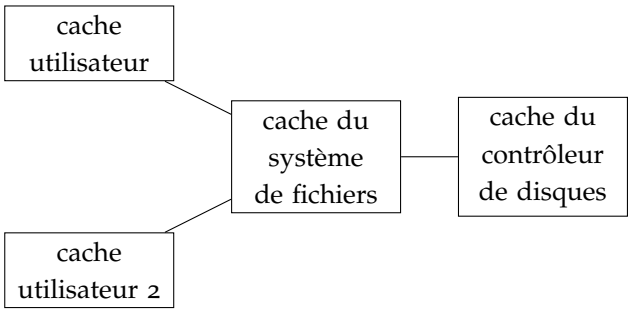


Notez qu'on peut tout aussi bien représenter les différents niveaux de cache d'un processeur monocœur :

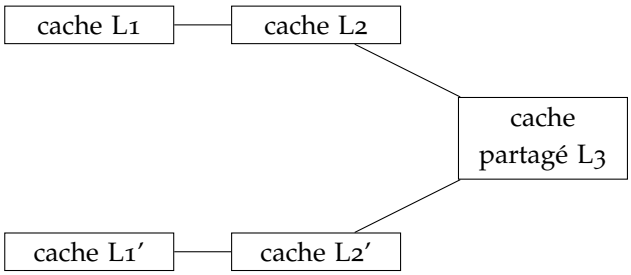


Dans ce dernier exemple nous avons inclus la mémoire principale du système dans la topologie.

Topologies arborescentes Si plusieurs utilisateurs se connectent, on obtiendra une topologie arborescente comme suit :



Là encore, on peut représenter les niveaux de cache d'un processeur, cette fois-ci bicœur (par exemple) :



Notez que dans ces deux cas, la modification d'une valeur dans les niveaux hauts nécessite une gestion particulière de la cohérence. Nous y reviendrons ci-après.

Stratégies d'exploitation

Entrée de la donnée dans le cache Les topologies vues ci-dessus permettent de nouvelles stratégies d'exploitations sur l'axe de l'entrée des données dans le cache.

EI4 La quatrième stratégie consiste à faire des caches supérieurs (c'est-à-dire proches des utilisateurs) des sous-ensembles des caches inférieurs. On appelle cela l'inclusivité.

EI5 La cinquième stratégie consiste à rendre les différents étages de cache exclusifs. Autrement dit on veut limiter ou supprimer la redondance de données afin de profiter d'un espace de cache global correspondant à la somme des espaces de tous les caches. L'idée date de [28], mais ce n'est que dans les années deux mille qu'elle commence à être exploitée avec DEMOTE [11], PROMOTE [12], EV [29], KARMA [30]ou encore HINTK [26].

Sortie de la donnée du cache Elles entraînent de plus l'existences d'une nouvelle stratégie sur l'axe de la sortie des données du cache.

EO4 La quatrième stratégie de cet axe consiste à faire sortir une donnée du cache si une autre copie de la même donnée est mise à jour dans un cache voisin. Cela nécessite de capturer la notion de *cohérence*.

[28] MUNTZ and HONEYMAN: "Multi-level Caching in Distributed File Systems -or- Your cache ain't nuthin' but trash", 1992.

[29] CHEN, ZHOU, and LI: "Eviction based cache placement for storage caches", 2003.

[30] YADGAR, FACTOR, and SCHUSTER: "Karma : know-it-all replacement for a multilevel cache", 2007.

Synchronisation de la donnée Avec les topologies arborescentes, une quatrième stratégie sur l'axe de la synchronisation des données voit le jour.

ES4 La quatrième stratégie consiste à mettre à jour une donnée du cache si une autre copie de la même donnée est mise à jour dans un cache voisin. C'est le pendant de la stratégie E04 et nécessite de même de capturer la notion de cohérence.

Note sur l'inclusivité Notons au sujet de la stratégie EI4 que le cas fondamental dans lequel un cache est présent en façade d'un stockage correspond déjà à cette stratégie. En effet le cache ne gère jamais aucune donnée qui ne soit pas présente dans le stockage. Autrement dit si le cas distribué nous force à l'expliciter, ce n'est en aucun cas une question spécifique à ce cas.

Stratégies de capture

Les stratégies de capture dans les systèmes distribués sont essentiellement celles décrites dans la section 1.2. Le défi dans le cas EI5 consiste donc à partager ou à distribuer les notions capturées entre les niveaux de cache de manière à maintenir l'exclusivité. Par exemple DEMOTE cherche à maintenir un LRU global sur une topologie linéaire.

Les nouvelles stratégies d'exploitation définies précédemment impliquent l'existence d'un nouvel axe dans les stratégies de capture : la capture de la cohérence.

Cohérence Ce nouvel axe comprend deux stratégies[9].

CC1 La première stratégie consiste à rendre le cache responsable de la notification des modifications qu'il réalise aux autres caches. Il sait en effet lorsqu'il modifie une donnée qu'elle devient nécéssairement non cohérente avec les caches voisins.

CC2 La seconde stratégie est de passer par une structure intermédiaire partagée par les caches pour réaliser les modifications sur les données. C'est cette structure qui se charge de notifier les caches concernés. On parle de «répertoire» (ou *directory* en anglais).

9 Les stratégies seront nommées CC*n*. C pour *Capture*, C pour *Coherency* et *n* est son numéro.

Note sur la cohérence On peut croire que le problème de la cohérence des données est créé par les topologies arborescentes. Cependant, et nous y avons déjà fait référence auparavant, ce problème intervient dès lors qu'il existe deux copies de la même donnée. Autrement dit dès qu'il existe un cache de la donnée, donc déjà dans le cas fondamental. C'est pour cette raison que la stratégie ES2 n'est normalement appliquée que lorsque l'on a accès à un espace de stockage non volatile de type NVRAM (*Non-Volatile Random-Access Memory*).

En fait les topologies linéaires (dont le cas de base n'est qu'un cas particulier) résolvent le problème de la cohérence de manière triviale grâce aux stratégies de synchronisation ES1 à ES3. Il n'est donc pas nécessaire d'en parler spécifiquement. Nous aurions pu appeler les

stratégies de synchronisation des stratégies de cohérence.

Note sur les stratégies du cas distribué La note précédente et celle concernant l'inclusivité nous amène au point suivant : si le cas distribué nous force à expliciter certaines stratégies, cela ne signifie pas que ces problèmes n'existent pas dans le cas de base. On peut même affirmer maintenant qu'en terme de stratégie les cas distribués ne changent pas grand chose. Pour autant ce sont des cas difficile pour lesquels il n'existe que peu de solutions. Cela nous laisse entendre que le point de vue global et stratégique que nous avons adopté ici peut être une bonne piste à suivre afin de dépasser les limites auxquelles nous sommes confrontés lorsque nous travaillons sur les caches.

1.4 *Problématiques latentes*

Un certain nombre de points reste encore à éclaircir. Certains de ces points sont des limites déclarées comme telles dans la littérature. D'autres sont présents «en filigrane» mais ne sont pas encore développés. Nous les avons regroupés sous le terme de «problématiques latentes» afin de compléter le tableau brossé dans les deux sections précédentes.

Systèmes de données complexes

Jusqu'ici toutes les stratégies que nous avons présentées sont plus ou moins explicitement destinées à des systèmes de données de type «mémoire», dans laquelle des blocs, ou des pages, sont structurées linéairement et auxquelles on accède grâce à leur adresse. Pourtant on s'intéresse aussi aux caches pour des systèmes de fichiers[2], des bases de données[19], etc. Or ces structures ne sont pas du tout linéaires, contrairement aux disques ou à la mémoire principale des systèmes.

On parvient tout de même à s'intéresser à ces autres structures car elles sont concrètement implémentées *par dessus* des systèmes de type bloc. Les caches qui y sont intégrés sont donc en fait des caches de blocs, potentiellement informés. C'est pour cela que ça fonctionne.

Mais que ce passe-t-il si on n'a pas accès aux informations de type bloc mais véritablement aux structures de système de fichier (arborescent) ou bien toute autre structure de niveau logique supérieur ? Généralement on se contente d'en cacher les contenus en lecture seule, et on perd toute capacité à appliquer les stratégies normalement applicables aux caches d'écriture.

La première problématique latente que nous rencontrons est donc celle-ci : «Comment appliquer des algorithmes de caches pensés pour des sytèmes simples tels que des blocs à des systèmes complexes tels que des systèmes de fichier ou autre ?».

Hiérarchies sémantiques

La problématique précédente nous entraîne sur la voie d'une autre moins évidente. Concrètement les systèmes informatiques sont souvent organisés en couches d'abstractions successives. Chacune de ces couches tire partie de la couche inférieure et propose de nouveaux services pour les couches supérieures.

Pour ce qui nous intéresse ici, les données, on peut reprendre l'exemple précédent. Une base de données est conçue pour fonctionner soit directement avec les disques en mode blocs soit avec un système de fichier tiers lui-même conçu pour des systèmes en mode bloc. On a donc affaire à une hiérarchie dont chaque étage est une abstraction distincte de l'étage précédent. Nous appellerons ce type d'architecture des «hiérarchies sémantiques» ou «hiérarchies logiques».

La deuxième problématique que nous rencontrons dans cette série est donc : «Dans une architecture de type "hiérarchie logique", comment gérer de manière commune les caches présents à chaque niveau de la hiérarchie (à des fins d'exclusivité par exemple) ?».

Dans [31], les auteurs se posent une question similaire : comment un cache de disque peut-il inférer les structures de données supérieurs afin d'en retirer des connaissances sur l'utilité des blocs ? Si cette question est distincte de la notre, le problème sous-jacent est bien celui des hiérarchies logiques.

[31] Li et al.: "C-Miner : Mining Block Correlations in Storage Systems", 2004.

Topologies complexes

Nous avons présenté plus haut les topologies linéraires et arborescentes. Les auteurs de politiques de caches ont depuis longtemps identifié comme limite à ces politiques la gestion efficaces de topologies arborescentes [11,12,26,30], en particulier dans le cas des caches d'écriture.

Ainsi la conception d'une politique de cache est-elle systématiquement adossée au choix d'une topologie cible. Il est donc, au mieux, très difficile d'appliquer une politique sur une topologie pour laquelle elle n'a pas été prévue, empêchant ainsi la réutilisation de techniques éprouvées dans d'autres situations. Le problème est donc similaire à celui des systèmes de données complexes.

1.5 *Ambitions de la thèse*

Nous avons maintenant fait le tour des problématiques que nous avons identifié concernant les caches. Nous sommes parvenus à les organiser logiquement sous la forme de stratégies que peuvent tenter de suivre les politiques de gestion de cache puis nous en avons montré les limites dans les problématiques latentes.

Objectif général Le fait d'être parvenu à réorganiser les problématiques de cache comme nous l'avons fait nous suggère la possibilité d'articuler logiquement des stratégies auparavant appliquées à

des systèmes sans rapport, et donc à priori irréconciliable. Cela nous laisse entrevoir la possibilité de construire un nouveau genre de système de cache. Ce système serait générique, dans le sens où il permettrait d'identifier clairement l'articulation des stratégies les unes par rapport aux autres, qu'il suffirait de brancher. Il permettrait de plus de se détacher au maximum des spécificités de topologies ou de structures de données.

On pourrait alors 1. réutiliser les techniques développées pour les appliquer à d'autres systèmes, et 2. homogénéiser le traitement des caches.

Cet objectif général est ambitieux et nécessitera des travaux complémentaires à ceux présentés dans ce mémoire. En particulier un travail de reformulation et de formalisation sera certainement nécessaire.

Objectif de la thèse Ce mémoire tente de réaliser un premier pas vers cet objectif en s'attachant à séparer les concepts normalement intriqués de structure de données, de topologies et de politiques de cache. Ce travail semble en effet le premier pas à réaliser afin de pouvoir exprimer les stratégies d'exploitation et de capture en des termes réutilisables.

On pourra alors envisager d'exprimer les politiques plus en intention et ainsi éviter de tomber dans le piège d'une trop forte spécifité.

Résultats présentés

La partie suivante développe les contributions de cette thèse à l'atteinte des objectifs présentés ci-dessus. Elles comportent :

— un modèle de stockage distribué qui sépare les aspects «structures de données», «topologies» et «politiques». On verra que cela implique la présence d'autres éléments ;
— un simulateur permettant d'exécuter des instances de ce modèle dans un but d'analyse ;
— deux applications représentant des politiques connues (LRU [16] et DEMOTE [11]) ;
— leur validation en confrontant leur comportement à celui d'une implémentation directe de ces cas (validant par la même le modèle général au sein duquel ils sont définis).

Plan La partie II se découpe comme suit :

Chapitre 2 présente les réflexions et choix ayant présidé à la forme du modèle présenté ;

Chapitre 3 présente le modèle développé durant la thèse ;

Chapitre 4 développe des exemples d'utilisation destinés à valider le modèle.

La partie III commence par une discussion systématique de tous les points abordés dans le modèle (chapitre 5) puis conclut ce mémoire (chapitre 6).

Deuxième partie
Contribution

2 *Les éléments structurants* 27
- *2.1 Réflexions sur les données* 27
- *2.2 Réflexions sur les topologies* 29
- *2.3 Schéma synoptique du modèle* 31

3 *Modèle formel* 35
- *3.1 Modèle de données* 35
- *3.2 Modèle de topologies* 43
- *3.3 Interprétations* 45
- *3.4 Le moteur d'exécution* 46
- *3.5 Modèle de politiques* 49
- *3.6 Le gestionnaire de requêtes* 51
- *3.7 Simulateur* 52

4 *Expériences* 53
- *4.1 Définition des mesures* 54
- *4.2 Opérations, stockage et acteur* 55
- *4.3 Définition des cas* `ACS`, `ACCS` *et* `ADDS` 59
- *4.4 Résultats* 62

« *(...) le but de l'abstraction n'est* pas *d'être vague, mais de créer un nouveau niveau sémantique dans lequel on peut être absoluement précis.*

. . . . . .

(...) the purpose of abstracting is not *to be vague, but to create a new semantic level in which one can be absolutely precise.* »

E.G. Dijkstra, *"The Humble Programmer". 1972 ACM Turing Award Lecture. in : Communications of the ACM 15 (10), p.864*, October 1972

2
Les éléments structurants

Dans ce chapitre, nous allons détailler les éléments de réflexions qui ont le plus structurés le modèle présenté au chapitre 3.

2.1 *Réflexions sur les données*

Notion d'asynchronisme et absorption

On adresse ici les caches d'écriture en particulier. Nous verrons plus bas comment cette logique finit par s'appliquer de même aux caches de lecture.

Le premier élément que nous avons retenu des systèmes de cache est leur nécessité de générer de l'*asynchronisme* par rapport au système de stockage sous-jacent. Par *asynchronisme*, nous entendons la capacité à traiter les données en cache pour éviter de devoir transmettre les opérations d'écriture systématiquement.

Implicitement, cela revient à maintenir en mémoire des opérations en plus des données. Cela peut se traduire par le maintient d'un état synchrone ou asynchrone associé à la donnée (cf. exemple figure 2.1). En maintenant cet état en mémoire, on permet à de multiples écritures en cache de se traduire ultérieurement (lors de l'éviction par exemple) en une seule écriture sur le stockage. On observe alors un phénomène d'*absorption* des écritures par le cache.

	État		
nœud	initial	après read(1)	après write(1,b)
C	—	$1 : a$, ✓	$1 : b$, ✘
S	$1 : a$ $2 : b$	$1 : a$ $2 : b$	$1 : a$ $2 : b$

Figure 2.1: Le cache (C) associe à chaque donnée maintenue un état synchrone (✓) ou asynchrone (✘) par rapport à l'état du stockage (S). Dans ce scénario, le cache subit une lecture de la ressource 1 (read(1)) puis une écriture de la ressource 1 avec une valeur b (write(1, b)).

En terme de lectures, le maintient en cache des données (synchrones ou non) met en œuvre exactement le même phénomène. Si la donnée accédée est dans le cache, elle peut y être lue directement et on évite ainsi de devoir lire sur le stockage à chaque appel. Ici, l'absorption se produit donc dans l'autre sens : on lit une fois pour ne plus lire ensuite, là ou pour l'écriture on n'écrit pas encore pour n'écrire qu'une fois ensuite.

Le fait d'observer le phénomène sous cet angle, qu'on appelle ici *absorption*, nous permet de concevoir qu'implicitement, des opérations sont stockées en même temps que les données, pour être simultanément simplifiées en une seule opération, préalable ou ultérieure.

Ce phénomène d'absorption est le pendant du *hit rate* si souvent utilisé pour mesurer l'efficacité des politiques de remplacement. En effet plus l'absorption est grande, plus les opérations produiront de

hits. Vu comme cela, «*absorption*» n'est guère plus qu'un autre terme pour «*hit rate*». Mais comme dit précédemment, la notion d'absorption met en jeu la notion d'opérations et leur gestion en mémoire, ce que ne fait pas la notion de *hit*.

Systèmes de données complexes

On arrive donc à l'idée selon laquelle il existe de manière éventuellement implicite un gestionnaire d'opérations au sein des systèmes de cache. Voyons maintenant comment on retrouve cette notion en travaillant cette fois sur la structure des données.

Une limite très nette des systèmes de cache existant est leur spécialisation aux données de type *bloc*, et donc leur incapacité à traîter de systèmes de données plus complexes tels que des arbres ou des graphes.

Par exemple, on trouve des caches dans les processeurs. Ce sont des caches de la mémoire, qu'on peut représenter sous la forme d'un ensemble de blocs reliés entre eux par leurs adresses si elles sont adjacentes. On ne peut traîter ces blocs qu'avec les opérations *read* et *write*, et le travail sur un bloc n'a pas d'impact sur les autres. On omet ici de distinguer la notion de *page* utilisée dans la mémoire pour regrouper des blocs, puisque ça n'en change pas le principe.

Un autre exemple : on trouve des caches dans les contrôleurs de disques, qui sont basiquement là encore des systèmes à base de blocs indépendants sur lesquels on peut lire ou écrire sans impacter d'autres blocs.

Même dans les caches intégrés aux systèmes de fichier, on se contente généralement de mettre en cache le contenu des fichier (des suites d'éléments contigües) et non les meta-données. Ces dernières suivent une structure d'arbre et il existe un nombre bien plus grand d'opérations qui peuvent en plus s'appliquer sur deux voire trois fichiers simultanément [1]. Lorsque parfois certains caches gèrent ces meta-données, ils ne le font qu'en lecture et chaque modification est transmise au stockage [2].

Si l'on veut produire un modèle de cache générique, il faut être capable de gérer ces systèmes de données plus complexes. De manière informelle, supposons que l'on aie affaire à un système de données mettant en jeu des opérations complexes sur plusieurs données en même temps, ces données étant liées entre elles au niveau du système de donnée. Si on veut gérer ces opérations en caches, les opérations successives sur ce système peuvent rapidement créer des dépendances entre ces opérations. Cela rendrait nécessaire l'existence d'un gestionnaire d'opérations, explicite, capable de gérer ces dépendances lors de l'éviction des données du cache (type *write-back*).

Un modèle de cache générique implique donc au minimum trois choses : 1. l'existence d'un modèle de donnée permettant de rendre compte des systèmes de données complexes et des opérations qu'ils gèrent, 2. la capacité d'instancier un de ces systèmes selon le modèle

1 L'opération *rename* par exemple, modifie les méta-données du fichier renommé, du répertoire parent source ainsi que du répertoire parent cible s'il est différent du répertoire source.

2 cf. par exemple le serveur de fichiers Ganesha : `https://github.com/nfs-ganesha/nfs-ganesha`

défini, et 3. l'existence d'un gestionnaire d'opérations capable de tenir compte des dépendances entre ces opérations. Dans la suite, une instance du système de données sera appelée un *état*.

Notons que le gestionnaire d'opérations devient l'objet en charge de l'absorption, étant le seul élément ayant une vue globale de ces opérations. L'absorption deviendrait alors un phénomène explicite impliquant l'existence au sein du modèle de données de la capacité à transformer une séquence d'opérations en un autre séquence équivalente.

Impact sur les politiques

Un effet avantageux de l'existence d'un modèle de données générique capable de présenter les relations entre les données est que la politique peut en tirer partie. On peut par exemple imaginer une politique réalisant du *prefetch* sur la seule base de l'existence de relations entre les ressources et plus sur la forme spécifique des adresses de ces données. On peut aussi imaginer typer les relations et décider de types standards étant interprétés comme des liens permettant la navigation dans le système de données. Toute politique de cache le souhaitant pourrait alors prendre une décisions informée sans avoir à connaître les spécificités de tel ou tel système de données.

On commence donc à concevoir des politiques qui adressent des classes de systèmes et non plus un système particulier comme c'est le cas dans les solutions actuelles.

2.2 *Réflexions sur les topologies*

On cherche à traîter de systèmes de caches distribués, c'est à dire d'éléments logiques communiquant et agissant dans un but commun : le maintient sûr d'informations dans une mémoire intermédiaire dans un réseau. Il apparaît nécessaire de réfléchir à la forme de ce réseau ainsi qu'à ce qu'implique la présence d'un système de cache dans ce réseau.

Notion d'interprétation

On nomme usuellement la forme d'un réseau une *topologie*. La notion usuelle de topologie intègre deux aspects : la structure du réseau et le traîtement de cette structure. Par exemple, la topologie d'un réseau en forme d'arbre est généralement appelée une hiérarchie. Or la notion de hiérarchie suppose l'identification d'une racine, à partir de laquelle on peut identifier des niveaux correspondant à la distance des nœuds de l'arbre à la racine. Mais la même forme de réseau peut tout à fait être comprise sans notion d'ordre. Il existe donc une notion secondaire au sein de la notion de topologie qui vient conditionner la manière dont on traîte les structures.

Nous appelons cette notion une *interprétation* et la séparons de la notion de topologie dans cette thèse.

On appellera donc *topologie* un graphe représentant les relations de voisinage entre nœuds du réseau et *interprétation* l'ensemble des concepts et relations adossés à une topologie permettant de lui faire subir des traîtements plus complexes.

Interprétation dans le cadre des caches

Reprenons les notions d'asynchronisme et d'absorption traîtées dans la section précédente sur les données.

> Le premier élément que nous avons retenu des systèmes de cache est leur nécessité de générer de l'*asynchronisme* par rapport au système de stockage sous-jacent.

Implicitement, ce paragraphe nous impose pour faire du cache l'existence d'un système de stockage «sous-jacent» ainsi que l'existence d'un autre nœud du réseau à l'origine des requêtes. On impose de plus l'existence d'une relation hiérarchique entre ces nœuds, une relation d'ordre.

Un système de cache ne se conçoit donc que dans une interprétation spécifique : une hiérarchie.

Notions d'état distribué et d'état global

La topologie telle que nous sommes en train de la définir dépasse déjà largement le seul domaine des caches. En effet pour traiter de ces derniers on a dû leur adjoindre des stockages et des acteurs.

Intuitivement, les nœuds de stockages sont chacun associés à un état. Les nœuds de caches sont aussi associés à un état, bien que plus volatile. De manière moins intuitive, les acteurs sont eux aussi associés à un état. Bien que cette piste ne sera pas explorée dans cette thèse, on peut argumenter que les acteurs étant à l'origine de tous les traitements sur le stockage ils finissent par obtenir une vue partielle de l'état du stockage. Dans ce document, on supposera que l'état des acteurs sera vide puisqu'on se focalisera sur le maintient des données dans le cache et dans le stockage.

On choisira donc d'associer de manière explicite à chaque nœud de la topologie un état. On aura donc systématiquement affaire à un état de fait distribué, même lorsqu'on considérera les systèmes les plus simples possibles. À cet état distribué correspondra un état dit «global» qui en est intuitivement une vue consolidée.

La manière de composer l'état distribué ne peut être définie que par l'interprétation dans laquelle on s'inscrit. On intègre ainsi dans la notion d'interprétation la fonction qui permet de projeter un état distribué d'une topologie sur un état global correspondant.

Impact sur les politiques

À l'instar du modèle de données intuité dans la section précédente, la séparation de l'interprétation et de la topologie nous permet de relier une politique de cache non plus à la forme d'un réseau mais à l'interprétation qu'on en fait ; nœud voisin, fils ou père sont des

concepts utilisables par une politique venant directement d'une interprétation de type hiérarchique par exemple.

Là encore, on commence à concevoir des politiques qui s'adressent à des classes de réseaux et plus seulement à des formes de réseaux spécifiques.

Aparté sur les stockages

En intégrant comme nous venons de le faire acteurs, caches et stockages à travers la notion d'interprétation, nous avons franchi une barrière. Nous venons en effet d'aborder ces trois notions comme étant de même nature. Si l'intégration de la notion d'acteur aux deux autres tient plus de l'artefact que d'une véritable unification, il nous semble important de nous arrêter sur les deux autres.

Cela répond en effet à une intuition que nous n'avons pas pu développer dans l'introduction : l'idée que caches et stockages sont finalement de même nature. Si par stockage, on entend tout objet capable d'adresser des données, de les organiser et de répondre à des requêtes, c'est à dire tout objet fournissant un moyen d'appliquer des traitements à des données, alors le cache devient lui-même une forme de stockage. Il agira comme un proxy, toujours subordonné à un autre stockage, et formera avec ce dernier un stockage de fait distribué.

Bien qu'informel, c'est déjà un résultat important car 1. c'est une première définition génériquei du cache ; et 2. il permet d'envisager le cache dans les mêmes termes que les stockages, ce que nous venons de toute façon de faire et ce qui suggère une unité de concept. De plus cela implique que les stockages peuvent être composés les uns avec les autres, ce que réalise la notion d'interprétation. Enfin, cela confirme que les problématiques distribuées sont bien présentes dès lors qu'il existe un cache, et pas seulement lorsque ce cache est distribué.

2.3 *Schéma synoptique du modèle*

Le modèle que nous proposons peut se représenter en six parties :

1. Le modèle de données, qui permet de définir formellement les structures de données et les opérations d'un système ;
2. Le modèle de topologie, qui permet de représenter un réseau dans lequel les nœuds sont associés à des états ;
3. Le modèle d'interprétation, qui permet de définir les éléments propres au système modélisés, tels que les notions de hiérarchie par exemple ;
4. Le modèle de politique, qui permet de définir le comportement des nœuds dans le système ; il se repose sur l'interprétation et le modèle de données ;
5. Le gestionnaire d'opérations, qui permet d'apporter l'asynchronisme et l'absorption utiles à des systèmes de données distribués ;

6. Le moteur d'exécution, qui fait appel à tous ces éléments et les fait interagir entre eux.

La figure 2.2 représente ces éléments ainsi ques les relations qui existent entre eux. Détaillons ces relations.

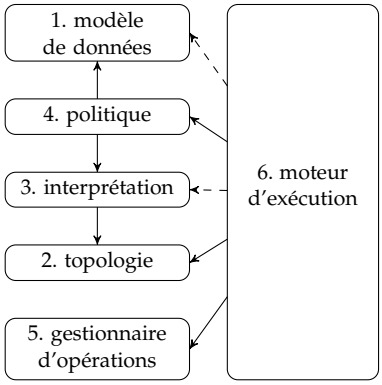


FIGURE 2.2: Schéma synoptique du modèle

Politique et modèle de données La politique est susceptible de faire appel au système de données afin de calculer l'espace disponible, de prendre une décision se basant sur la présence ou l'absence d'une ressource spécifique, ou encore de prendre une décision se basant sur la présence de relations entre plusieurs ressources. Lorsque la politique est rafraichie par le moteur d'exécution (cf. le paragraphe *Moteur d'exécution et politique*), elle peut avoir besoin d'éléments du modèle de données (notamment les définitions d'opérations afin de calculer le domaine d'une requête).

Politique et interprétation La politique est susceptible de faire appel à l'interprétation si elle a besoin de réaliser un calcul faisant appel à certains concepts n'existant que dans l'interprétation. Par exemple, la politique peut être amenée à choisir un nœud de la topologie qui ait une relation hiérarchique avec le nœud courant.

Interprétation et topologie L'interprétation est toujours adossée à une topologie afin de donner à la politique une réponse précise. Dans l'exemple précédent, l'interprétation doit rendre à la politique un nœud. Ce nœud n'étant connu que de la topologie, il doit y avoir cette relation proche entre interprétation et topologie.

Moteur d'exécution et politique Le moteur d'exécution fait appel à la politique à plusieurs occasions : mise à jour de la politique avec des événements en cours ou passés, demande d'un module adapté pour traîter un événement standard. Dans ces cas, selon l'implémentation du modèle, le moteur d'exécution peut avoir à passer à la politique l'interprétation et le modèle de données.

Moteur d'exécution et topologie Le moteur d'exécution fait appel à la topologie lorsqu'il doit gérer des communications entre les nœuds de cette topologie.

Le système que nous sommes en train de définir fait explicitement appel à la fois à l'interprétation et à la topologie. Dans un soucis d'expliciter tous les aspects intriqués de la gestion d'un système de stockage distribué, on garde ces deux objets séparés au lieu de faire de l'interprétation une extension de la topologie.

Moteur d'exécution et gestionnaire d'opérations Le moteur d'exécution fait appel au gestionnaire d'opérations à trois occasions : 1. l'arrivée de nouvelles requêtes dans le système, 2. la prise en charge d'une requête à réaliser et 3. la finalisation d'une requête par le système. La figure 2.3 représente les différents états que traverse le système concernant cet aspect.

	moteur d'exécution		gestionnaire de requêtes	
actions	avant	après	avant	après
ajout	req	—	—	req
exécution	—	req	req	req*
finalisation	req	—	req*	—

FIGURE 2.3: Cycle de vie d'une requête req du point de vue du moteur d'exécution et du gestionaire de requêtes. On observe ici les échanges de responsabilités entre ces deux éléments. Pour le gestionnaire de requêtes, req* signifie en cours de traitement.

3
Modèle formel

Notre modèle est composé de six parties :

1. un modèle de *données* qui définit une notion d'état et d'opérations utilisateurs sur ces états ;
2. un modèle de *topologie* qui définit une notion de nœuds stockant des données, formant ainsi un état distribué, ainsi qu'un modèle de communication pour échanger des requêtes ;
3. le modèle de topologie mène à une notion d'interprétation qui définit la notion d'état global, d'intégration, et les éléments propres du système ;
4. un modèle de *politiques* qui décrit le comportement des nœuds de la topologie ;
5. un gestionnaire de requêtes, au cœur de l'asynchronisme et de l'absorption ;
6. un moteur d'exécution permettant l'intégration des éléments précédents.

Ce chapitre formalise ces différentes parties.

3.1 Modèle de données

Ce modèle se divise en deux parties : la définition des états et la définition des opérations agissant sur ces états. Ensembles, ces deux parties permettent à un utilisateur de définir et d'étudier les types de données qui lui sont spécifiques. Une troisième partie viendra en présenter un exemple.

États

Définitions de base On veut être capable de représenter le plus grand nombre de type de données possibles. Il faut pour cela considérer deux éléments : la valeur des données et leur structure, c'est-à-dire les relations (éventuellement typées) qui existent entre ces données.

On doit donc considérer un ensemble de *clés*, un ensemble de *valeurs* à leur associer et un ensemble d'*étiquettes* pour typer les relations.

Un *état* sera définit comme une paire : le premier élément reliera des clés à des valeurs, le deuxième reliera des ensembles de clés par des étiquettes. On définiera ainsi la valeur des ressources de notre système ainsi que les relations existant entre ces ressources.

DÉFINITION 1. *Soient K, V, L trois ensembles disjoints deux à deux, K et V non vides. Soient $H_{K,V} \stackrel{\text{df}}{=} 2^{K \times V}$ et $R_{K,L} \stackrel{\text{df}}{=} 2^{L \times 2^K \times 2^K}$.*

Un état *σ est une paire $(\sigma.h, \sigma.r)$ telle que $\sigma.h \in H_{K,V}$ et $\sigma.r \in R_{K,L}$.*

On note $\Sigma_{K,V,L}$ l'ensemble de tous les états exprimables à partir des ensemble K, V et L selon cette définition.

Par exemple, pour modéliser un système de fichier Unix, on pourrait choisir K un ensemble de chemins, V l'ensemble des valeurs qu'un fichier peut prendre et $L = \{\texttt{dir}\}$ pour symboliser la relation *répertoire parent* entre un fichier et un ensemble de fichiers [1].

Autre exemple, pour modéliser une mémoire, on choisirait K l'ensemble des adresses mémoires, V l'ensemble des valeurs que peut prendre un bloc mémoire, et L serait vide.

1 Ce serait évidemment une modélisation grossière sans commune mesure avec la complexité des systèmes de fichiers.

On appellera *domaine d'un état σ* l'ensemble des clés incluses dans l'une ou l'autre de ses composantes $\sigma.h$ ou $\sigma.r$.

DÉFINITION 2. *On définit* $\mathsf{dom}(\sigma.h) \stackrel{\text{df}}{=} \{k \mid \exists v \in V, (k, v) \in \sigma.h\}$ *et* $\mathsf{dom}(\sigma.r) \stackrel{\text{df}}{=} \bigcup_{(l,K_1,K_2)\in\sigma.r} K_1 \cup K_2$.

On définit $\mathsf{dom}(\sigma) \stackrel{\text{df}}{=} \mathsf{dom}(\sigma.h) \cup \mathsf{dom}(\sigma.r)$.

Propriétés On peut maintenant définir quelques propriétés pour nos états.

On dira qu'un état est *bien formé* s'il n'associe qu'une seule valeur à chacune de ses clés dans $\sigma.h$ et s'il ne construit pas de relations dans $\sigma.r$ entre des clés qu'il ne connait pas dans $\sigma.h$.

DÉFINITION 3. *Un état $\sigma \in \Sigma_{K,V,L}$ est dit* bien formé *si et seulement si il satisfait les conditions suivantes :*

— *$\sigma.h$ est une fonction : $\forall k \in \mathsf{dom}(\sigma.h), |\{(k, v) \in \sigma.h\}| = 1$;*

— *les relations de $\sigma.r$ n'existent que pour des clés de $\sigma.h$:* $\mathsf{dom}(\sigma.r) \subseteq \mathsf{dom}(\sigma.h)$.

Il est donc possible de définir des états qui n'auraient aucun sens dans des systèmes réels. C'est donc une propriété importante qu'on cherchera à vérifier, à rapprocher de la notion de cohérence.

On dira qu'un état a est inclus dans un état b si chacune des composantes de a est incluse dans la composante correspondante dans b.

DÉFINITION 4. *On définit un ordre partiel $\preceq$ sur $\Sigma_{K,V,L}$:*

$$\sigma_a. \preceq \sigma_b \Leftrightarrow \sigma_a.h \subseteq \sigma_b.h \wedge \sigma_a.r \subseteq \sigma_b.r$$

Opérations sur les états On dote nos états des opérations d'union, d'intersection et de différence, qui sont les généralisations des opérations du même nom dans les ensembles.

DÉFINITION 5. *On équipe $\Sigma_{K,V,L}$ de trois opérateurs $\cup$, $\cap$ et $\setminus$ qui sont les extensions composante par composante de leur équivalent ensembliste respectif :*

$$\begin{array}{rcl} \sigma_a, \sigma_b \in \Sigma_{K,V,L}, \quad \sigma_a \cup \sigma_b & \stackrel{\text{df}}{=} & (\sigma_a.h \cup \sigma_b.h, \sigma_a.r \cup \sigma_b.r) \\ \sigma_a \cap \sigma_b & \stackrel{\text{df}}{=} & (\sigma_a.h \cap \sigma_b.h, \sigma_a.r \cap \sigma_b.r) \\ \sigma_a \setminus \sigma_b & \stackrel{\text{df}}{=} & (\sigma_a.h \setminus \sigma_b.h, \sigma_a.r \setminus \sigma_b.r) \end{array}$$

On dote également nos états d'un opérateur de division qui permet de séparer un état σ des éléments de $\sigma.h$ et $\sigma.r$ mettant en jeu une clé $k \in K$.

DÉFINITION 6. *Soient $\sigma \in \Sigma_{K,V,L}$ et $k \in K$. On note σ/k la restriction de σ à ses éléments ne mettant pas en jeu k :*

$$\sigma/k \stackrel{\text{df}}{=} (\sigma'.h, \sigma'.r) : \begin{cases} \sigma'.h & = & \{(k', v) \in \sigma.h \mid k' \neq k\} \\ \sigma'.r & = & \{(l, K_1, K_2) \in \sigma.r \mid k \notin K_1 \cup K_2\} \end{cases}$$

Effets On introduit en prévision de la définition des opérations une extension des états qu'on appelle *effet*. Un effet est un couple d'états $(e.plus, e.minus) \in \Sigma^2_{K,V,L}$ dont le premier élément sera interpreté comme ce qu'il faudra ajouter à l'état et le second comme ce qui lui sera retiré. Un effet s'applique sur un état grâce à l'opérateur $\gg$, appelé projection.

DÉFINITION 7. *Soient $\sigma_{in} \in \Sigma_{K,V,L}$ et $e \in \Sigma^2_{K,V,L}$.*

$$e \gg \sigma_{in} \stackrel{\text{df}}{=} (\sigma_{in} \setminus e.minus) \cup e.plus$$

On dote également les effets d'un opérateur de division. Ce sera la généralisation de $/$ à $\Sigma^2_{K,V,L}$.

DÉFINITION 8. *Soit $e \in \Sigma^2_{K,V,L}$ un effet et $k \in K$. On note e/k la restriction de e à ses éléments ne mettant pas en jeu k :*

$$e/k \stackrel{\text{df}}{=} \begin{pmatrix} e.plus/k & e.minus/k \end{pmatrix}$$

Nous aurons besoin de doter $\Sigma^2_{K,V,L}$ de deux autres opérateurs pour composer les effets entre eux. Ce sera utile lorsqu'on abordera les topologies et les interprétations.

Commençons par la séquence d'effets.

DÉFINITION 9. (Séquence d'effets) *On définit l'opérateur de séquence « ; » entre deux effets e et f. Les deux définitions suivantes sont équiva-*

lents :

$$\sigma \in \Sigma_{K,V,L}, e, f \in \Sigma^2_{K,V,L},$$
$$(e; f) \gg \sigma \stackrel{\text{df}}{=} f \gg (e \gg \sigma)$$
$$(e; f) \stackrel{\text{df}}{=} \begin{pmatrix} (e^+ \setminus f^-) \cup f^+ & e^- \cup f^- \end{pmatrix}$$

PREUVE. Soient A, B et C trois ensembles, on rappelle les propriétés ensemblistes suivantes :

$$(A \cup B) \setminus C = (A \setminus C) \cup (B \setminus C)$$
$$(A \setminus B) \setminus C = A \setminus (B \cup C)$$

Partons de la seconde définition.

$$\begin{aligned}
(e; f) &\stackrel{\text{df}}{=} \begin{pmatrix} (e^+ \setminus f^-) \cup f^+ & e^- \cup f^- \end{pmatrix} \\
(e; f) \gg \sigma &= \begin{pmatrix} (e^+ \setminus f^-) \cup f^+ & e^- \cup f^- \end{pmatrix} \gg \sigma \\
&= [\sigma \setminus (e^- \cup f^-)] \cup (e^+ \setminus f^-) \cup f^+ \\
&= [(\sigma \setminus e^-) \setminus f^-)] \cup (e^+ \setminus f^-) \cup f^+ \\
&= ([(\sigma \setminus e^-) \cup e^+] \setminus f^-) \cup f^+ \\
&= f \gg [(\sigma \setminus e^-) \cup e^+] \\
(e; f) \gg \sigma &= f \gg (e \gg \sigma)
\end{aligned}$$

Continuons avec l'opérateur de parallélisme.

DÉFINITION 10. *On définit l'opérateur de parallélisme « || » entre deux effets e et f dont le domaine est disjoint. Les deux définitions suivantes sont équivalentes :*

$$\sigma \in \Sigma_{K,V,L},\ e, f \in \Sigma^2_{K,V,L},\ \mathsf{dom}(e) \cap \mathsf{dom}(f) = \emptyset,$$
$$(e||f) \gg \sigma \stackrel{\text{df}}{=} e \gg (f \gg \sigma) = f \gg (e \gg \sigma)$$
$$(e||f) \stackrel{\text{df}}{=} \begin{pmatrix} e^+ \cup f^+ & e^- \cup f^- \end{pmatrix}$$

PREUVE. Partons de la première définition :

$$\begin{aligned}
(e||f) \gg \sigma \stackrel{\text{df}}{=} e \gg (f \gg \sigma) &= f \gg (e \gg \sigma) \\
e \gg [(\sigma \setminus f^-) \cup f^+] &= f \gg [(\sigma \setminus e^-) \cup e^+] \\
([(\sigma \setminus f^-) \cup f^+] \setminus e^-) \cup e^+ &= ([(\sigma \setminus e^-) \cup e^+] \setminus f^-) \cup f^+ \\
[(\sigma \setminus f^-) \setminus e^-] \cup [f^+ \setminus e^-] \cup e^+ &= [(\sigma \setminus e^-) \setminus f^-] \cup [e^+ \setminus f^-] \cup f^+ \\
[\sigma \setminus (f^- \cup e^-)] \cup f^+ \cup e^+ &= [\sigma \setminus (e^- \cup f^-)] \cup e^+ \cup f^+ \\
\begin{pmatrix} f^+ \cup e^+ & f^- \cup e^- \end{pmatrix} \gg \sigma &= \begin{pmatrix} e^+ \cup f^+ & e^- \cup f^- \end{pmatrix} \gg \sigma \\
(e||f) &= \begin{pmatrix} e^+ \cup f^+ & e^- \cup f^- \end{pmatrix}
\end{aligned}$$

Représentations Un état σ se compose de deux ensembles $\sigma.h$ et $\sigma.r$. Un effet e se compose de deux états $e.plus$ et $e.minus$. On pourra les représenter de la manière suivante :

$$\sigma = \begin{pmatrix} \{\sigma.h\} \\ \{\sigma.r\} \end{pmatrix} \qquad e = \begin{pmatrix} {}^+\{e.plus.h\} & {}^-\{e.minus.h\} \\ {}^+\{e.plus.r\} & {}^-\{e.minus.r\} \end{pmatrix}$$

Opérations

Les états nous permettent de représenter une large gamme de données. Il faut maintenant les rendre manipulables. Pour cela on définit des *opérations*.

Définitions préalables Avant de définir les opérations nous avons besoin de définir deux notions : les *expressions* et les *valuations*.

Une *expression* est une chaîne de caractères qu'on peut évaluer. Elle peut faire référence à des variables auxquelles il faudra associer des valeurs. Cette association est une *valuation*, ou encore *binding* en anglais.

DÉFINITION 11. *Une* valuation *est une fonction qui associe à un ensemble de* variables *un ensemble de* valeurs *(attention à ne pas confondre ces valeurs avec les valeurs de l'ensemble V du modèle de données).*

Dans notre cas, les expressions seront écrites en langage mathématique. Dans une implémentation elles pourront être écrites dans un langage de programmation quelconque.

Pour des raisons pratiques on se donne quelques notations.

DÉFINITION 12. *Soit β une valuation. On note* $\mathsf{keys}(\beta) \stackrel{\text{df}}{=} \mathsf{img}(\beta) \cap K$ *l'ensemble des clés référencées dans β, avec* img *l'image (ou le codomaine) de la valuation.*

On peut composer les valuations.

DÉFINITION 13. *Soient $\beta_a, \beta_b \in \mathcal{B}_{op,K,V,L}$ deux valuations dont les domaines sont disjoints. On définit leur composition $+$ comme suit :*

$$\mathsf{dom}(\beta_a) \cap \mathsf{dom}(\beta_b) = \emptyset, \forall x \in \mathsf{dom}(\beta_a) \cup \mathsf{dom}(\beta_b),$$
$$\beta_a(x) + \beta_b(x) \stackrel{\text{df}}{=} \begin{cases} \beta_a(x) & \textit{si } x \in \mathsf{dom}(\beta_a), \\ \beta_b(x) & \textit{sinon, i.e., si } x \in \mathsf{dom}(\beta_b) \end{cases}$$

Définitions de base Une opération se compose de quatre éléments : un nom, une garde, un effet et un ensemble de variables.

La garde est une expression booléenne qui vérifie qu'un état est en mesure de subir cette opération. Elle devra donc être vérifiée avant chaque tentative d'appliquer l'opération à l'état courant.

L'effet est une expression qui s'évalue dans $\Sigma^2_{K,V,L}$. C'est l'élément opérant de l'opération.

L'expression de la garde ainsi que l'expression de l'effet mettent en jeu des variables auxquelles il faut associer des valeurs concrètes. Ces valeurs sont les paramètres de l'opération. Elles ne peuvent pas toujours être fournies par l'utilisateur, auquel cas c'est le système qui doit les évaluer [2]. Pour cette raison il est nécessaire de spécifier l'ensemble des variables que l'utilisateur doit valuer. Cet ensemble de variable est le quatrième élément de la définition d'une opération.

2 Par exemple lors de la création d'éléments dans l'état, la clé de l'élément créé ne peut pas toujours être décidée par l'utilisateur. C'est le cas dans les systèmes de fichiers par exemple.

La valuation se fera en deux temps : la valuation de l'utilisateur (valuation d'entrée) et celle du système (en cas de succès, valuation de sortie). Ces deux valuations combinées sont nécessaires et suffisantes pour 1. évaluer la garde et 2. évaluer l'effet. Dans ce modèle, on prend donc le parti de retourner à l'utilisateur la valuation de sortie en cas de succès. *Cela peut créer une différence sensible par rapport aux comportements habituels des systèmes. On renverra en général à l'utilisateur en général plus d'informations que dans un système réel.*

DÉFINITION 14. *On note* $\mathsf{vars}(e)$ *l'ensemble des variables impliquées dans une expression e. Une opération est un 4-uplet* $op \stackrel{\text{df}}{=} (op.name, op.guard, op.effect, op.params)$ *tel que :*

- *op.name est un nom (n'importe quelle chaîne de caractère) ;*
- *op.guard est une expression booléenne ;*
- *op.effect est une expression qui s'évalue comme un couple d'états ;*
- *op.params est un ensemble de variables tel que :* $op.params \subseteq \mathsf{vars}(op.guard) \cup \mathsf{vars}(op.effect) \stackrel{\text{df}}{=} \mathsf{vars}(op)$ *;*
- *on a* $\mathsf{vars}(op.effect) \subseteq op.params \cup \mathsf{vars}(op.guard)$ *;*
- *il existe au moins une valuation telle que op.effect et op.guard sont effectivement calculables* (i.e. *elles sont toutes les deux évaluables).*

On évite volontairement de préciser une syntaxe pour les expressions car on ne veut ni fixer K, V, L ni restreindre la portée de nos définitions.

Le dernier point de cette définition est suffisant pour s'assurer qu'une implémentation de ce modèle fournisse une syntaxe concrète d'expression ainsi qu'une manière efficace de les évaluer.

Notations On se donne quelques notations afin de faciliter les définitions ultérieures.

DÉFINITION 15. *On note* $\mathcal{OPS}$ *l'ensemble des opérations définies pour un système donné.*

Soit $op \in \mathcal{OPS}$*. On note :*

- $\mathcal{B}_{op,K,V,L}$ *l'ensemble des valuations* $\beta : \mathsf{vars}(op) \to K \cup V \cup L$ *;*
- $\mathcal{B}^{in}_{op,K,V,L}$ *l'ensemble des valuations* $\beta : op.params \to K \cup V \cup L$ *;*
- $\mathcal{B}^{out}_{op,K,V,L}$ *l'ensemble des valuations* $\beta : \mathsf{vars}(op) \setminus op.params \to K \cup V \cup L$*.*

Nous pouvons noter que $\mathcal{B}_{op,K,V,L} = \mathcal{B}^{in}_{op,K,V,L} \cup \mathcal{B}^{out}_{op,K,V,L}$.

DÉFINITION 16. *Soient* $\sigma_{in} \in \Sigma_{K,V,L}$ *et* $\beta_{in} \in \mathcal{B}^{in}_{op,K,V,L}$.

On note $op.guard(\sigma_{in}, \beta)$ *l'évaluation de op.guard avec* $\beta + \{`\sigma' \to \sigma_{in}\}$*, où* σ *est une convention d'écriture pour faire référence à l'état courant dans la garde.*

DÉFINITION 17. *On note* $op.effect(\beta)$ *l'évaluation de op.effect avec* β*.*

DÉFINITION 18. *On définit la fonction op.candidates qui retourne l'ensemble des valuations de sorties qui, composées avec β_{in}, permettent de valider la garde et d'évaluer l'effet.*
$op.candidates(\sigma_{in}, \beta_{in}) \stackrel{\text{df}}{=} \{\beta_{out} \in \mathcal{B}^{out}_{op,K,V,L} \mid op.guard(\sigma_{in}, \beta_{in} + \beta_{out}) \wedge op.effect(\beta_{in} + \beta_{out}) \in \Sigma^2_{K,V,L}\}$

Une implémentation du modèle pourra définir explicitement cette fonction afin d'être plus efficace.

Propriétés Une opération est dite *éligible* pour un état σ et une valuation β_{in} si elle admet au moins candidat.

DÉFINITION 19. $\text{eligible}(op) \Leftrightarrow op.candidates(\sigma_{in}, \beta_{in}) \neq \emptyset$

Elle est dite *déterministe* si elle ne peut admettre au plus qu'un seul candidat pour tout état σ et toute valuation d'entrée β_{in}.

DÉFINITION 20. *op est dite* déterministe *si et seulement si, pour tous $\sigma_{in} \in \Sigma_{K,V,L}$ et tous $\beta_{in} \in \mathcal{B}^{in}_{op,K,V,L}$ on a $|op.candidates(\sigma_{in}, \beta_{in})| \leq 1$.*

Usage Quand op est éligible pour un état et une valuation, l'ensemble des états et valuations de sorties est calculé en appliquant op avec chaque valuation candidate, c'est à dire en projetant son effet comme suit.

DÉFINITION 21. *L'application de l'opération $op \in \mathcal{OPS}$ sur l'état $\sigma_{in} \in \Sigma_{K,V,L}$, étant donnée une valuation $\beta_{in} \in \mathcal{B}^{in}_{op,K,V,L}$, renvoie un sous-ensemble de $\mathcal{B}^{out}_{op,K,V,L} \times \Sigma_{K,V,L}$. Elle est définie par $op(\sigma_{in}, \beta_{in}) \stackrel{\text{df}}{=} \{(\beta_{out}, op.effect(\beta_{in} + \beta_{out}) \gg \sigma_{in}) \mid \beta_{out} \in op.candidates(\sigma_{in}, \beta_{in})\}$.*

Ainsi le vrai résultat d'une opération n'est pas une valuation de sortie mais un ensemble de couples valuation de sortie - état. Bien que ce ne soit pas l'objectif principal, cette partie du modèle peut être utilisée indépendamment des autres pour étudier la correction (ou d'autre propriétés) du modèle de données implémenté.

Exemple : système de fichiers simplifié

Voyons maintenant un exemple pour illustrer le modèle présenté plus haut. Ce sera un système de fichier extrêmement simplifié.

Soient K, V, L nos trois ensembles de base.

— K est l'ensemble des entiers naturels ;

— V est l'ensemble des chaînes de caractères possibles grâce à l'alphabet anglais (toutes les lettres, non accentuées) ;

— $L = \{\mathsf{root}, \mathsf{dir}, \mathsf{file}\}$.

On se donne les conventions suivantes :

— il existe un et un seul répertoire racine, qui n'est lié qu'à lui-même par la relation root ;

— tous les autres répertoires, ainsi que les fichiers sont liés à un répertoire parent par la relation dir.

Pour l'exemple, seule une partie des méta-données sera représentée, et aucune donnée associée au contenu des fichiers. C'est-à-dire qu'on ne représentera que la structure de répertoires et de fichiers.

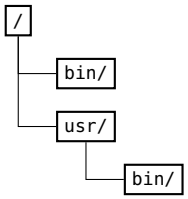


FIGURE 3.1: État initial d'un système de fichiers simplifié

La figure 3.1 représente un extrait de système de fichier. Ce sera notre état initial σ_0. On peut le représenter comme suit :

$$\sigma_0.h \stackrel{\text{df}}{=} \{(0, /), (1, \texttt{bin}), (2, \texttt{usr}), (3, \texttt{bin})\}$$
$$\sigma_0.r \stackrel{\text{df}}{=} \{(\mathsf{root}, \{0\}, \emptyset), (\mathsf{dir}, \{0\}, \{1\}), (\mathsf{dir}, \{0\}, \{2\}), (\mathsf{dir}, \{2\}, \{3\})\}$$

On dote notre système de fichiers de trois opérations : *create*, *makedir*, *delete* (figure 3.2). Par soucis de lisibilité, on introduit quelques fonctions booléennes [3] :

$$isroot(k, \sigma) \stackrel{\text{df}}{=} (\mathsf{root}, \{k\}, \emptyset) \in \sigma.r$$
$$isdir(k, \sigma) \stackrel{\text{df}}{=} isroot(k, \sigma) \vee (\exists K_1 \mid (\mathsf{dir}, K_1, \{k\}) \in \sigma.r)$$
$$exists(k, \sigma) \stackrel{\text{df}}{=} \exists v \in V \mid (k, v) \in \sigma.h$$
$$empty(k, \sigma) \stackrel{\text{df}}{=} \nexists K_2 \in 2^K \mid (\mathsf{dir}, \{k\}, K_2) \in \sigma.r$$

3 Notez que notre implémentation permet de définir de telles fonctions annexes et de les utiliser au sein des opérations.

nom	garde	effet	paramètres
`create`	$exists(k_p, \sigma) \wedge isdir(k_p, \sigma) \wedge \neg exists(k, \sigma)$	$\binom{(k,v)\quad \emptyset}{(\mathsf{file}, \{k_p\}, \{k\})\ \emptyset}$	$\{k_p, v\}$
`mkdir`	$exists(k_p, \sigma) \wedge isdir(k_p, \sigma) \wedge \neg exists(k, \sigma)$	$\binom{(k,v)\quad \emptyset}{(\mathsf{dir}, \{k_p\}, \{k\})\ \emptyset}$	$\{k_p, v\}$
`delete`	$exists(k, \sigma) \wedge \neg isroot(k, \sigma) \wedge empty(k, \sigma)$	$\binom{\emptyset\quad (k,v)}{\emptyset\ (l, \{k_p\}, \{k\})}$	$\{k\}$

FIGURE 3.2: Opérations d'un système de fichiers simplifié

Essayons de compléter l'état σ_0 en créant un fichier `sh` dans le répertoire `/bin/`. On utilisera l'opération *create* à laquelle on doit associer une valuation d'entrée pour k_p et v.

Le répertoire `/bin` est associé à la clé 1 et le nouvel élément que nous voulons créer aura la valeur `sh`.

$$\beta_{in} \stackrel{\text{df}}{=} \{k_p \to 1, v \to \texttt{sh}\}$$

Il manque la valuation de 'k' pour pouvoir évaluer complètement la garde ainsi que l'effet. C'est le rôle de *op.candidates* que de fournir ce complément. Ici, l'ensemble K étant infini, on se limitera à un sous-ensemble fini.

$$create.candidates(\sigma_0, \beta_{in}) = \begin{cases} \{k \to 4\} \\ \{k \to 5\} \\ \dots \end{cases}$$

On peut vérifier que ces candidats sont corrects, c'est à dire que la garde de *create* est vérifiée et que son effet s'évalue bien. Par exemple

pour $\beta_{out} = \{k \to 4\}$:

$$\sigma_0 = \left(\begin{smallmatrix} \{(0,/),\ (1,\texttt{bin}),\ (2,\texttt{usr}),\ (3,\texttt{bin})\} \\ \{(\mathsf{root},\{0\},\emptyset),(\mathsf{dir},\{0\},\{1\}),\ (\mathsf{dir},\{0\},\{2\}),\ (\mathsf{dir},\{2\},\{3\})\} \end{smallmatrix} \right)$$

$$\begin{aligned} \beta &= \beta_{in} + \beta_{out} \\ &= \{k_p \to 1, v \to \mathsf{sh}, k \to 4\} \\ create.guard(\sigma_0, \beta) &= exists(1, \sigma_0) \wedge isdir(1, \sigma_0) \wedge \neg exists(4, \sigma_0) \\ &= true \wedge true \wedge true \\ \boldsymbol{create.guard} &\boldsymbol{= true} \\ create.effect &= \left(\begin{smallmatrix} (k,v) & \emptyset \\ (\mathsf{file},\{k_p\},\{k\}) & \emptyset \end{smallmatrix} \right) \\ &= \left(\begin{smallmatrix} (4,\mathsf{sh}) & \emptyset \\ (\mathsf{file},\{1\},\{4\}) & \emptyset \end{smallmatrix} \right) \\ \boldsymbol{create.effect} &\boldsymbol{\in \Sigma^2_{K,V,L}} \end{aligned}$$

On peut donc utiliser β_{out} pour appliquer l'opération.

$$create.effect(\beta_{in} + \beta_{out}) \gg \sigma_0 = \left(\begin{smallmatrix} \{(0,/),\ (1,\texttt{bin}),\ (2,\texttt{usr}),\ (3,\texttt{bin}),\ (4,\mathsf{sh})\} \\ \{(\mathsf{root},\{0\},\emptyset),(\mathsf{dir},\{0\},\{1\}),\ (\mathsf{dir},\{0\},\{2\}),\ (\mathsf{dir},\{2\},\{3\}),\ (\mathsf{file},\{1\},\{4\})\} \end{smallmatrix} \right)$$

On obtient le système représenté figure 3.3.

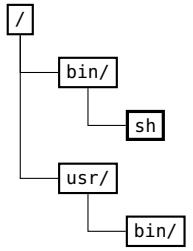


FIGURE 3.3: État final de notre système de fichiers simplifié. En gras le nouvel élément.

Discussions

Le modèle décrit précédemment permet de représenter et de faire évoluer une large gamme de données. Il est important de noter que ce modèle d'état permet de représenter des situations qui ne sont pas souhaitables. Par exemple une resource k peut tout à fait être associée à deux valeurs v_1 et v_2, alors que le type de données modélisé ne le permet peut-être pas.

Il est souhaitable de pouvoir représenter de telles mauvaises situations. Cela permet d'observer les erreurs de conception et de vérifier le comportement du système modélisé.

On dispose maintenant d'une algèbre permettant de modéliser des systèmes de données. S'il est possible de faire du calcul sur cette base, il n'est pas encore possible de traiter des machines qui les portent. C'est le rôle du modèle de topologie présenté dans la section suivante.

3.2 *Modèle de topologies*

Définition de base Un stockage distribué est défini par un ensemble de *nœuds* associés à des états locaux et des opérations locales, et qui communiquent à travers des *bus*. Ils sont formalisés sous la forme d'un hypergraphe comme suit.

DÉFINITION 22. *Soit N un ensemble de* nœuds. *Une* topologie *T sur N est une paire $T \stackrel{\text{df}}{=} (T.nodes, T.buses)$ où $T.nodes \stackrel{\text{df}}{=} N$ est l'ensemble des nœuds et $T.buses \subseteq 2^N \setminus \emptyset$ est l'ensemble des hyperarcs. Pour $i, j \in T.nodes$, on note $T[i,j]$ le fait qu'il existe $b \in T.buses$ tel que $\{i,j\} \subseteq b$.*

Communications

Étant donné une topologie T, les nœuds de $T.nodes$ sont autorisés à communiquer en échanger des *trames* sur les bus de $T.buses$. On suppose qu'un bus ne peut transmettre qu'un message à la fois, *i.e.* un émetteur est bloqué jusqu'à ce qu'un message précedemment envoyé soit reçu. de plus, un récepteur est bloqué jusqu'à ce qu'un message lui soit envoyé. Les trames possibles sont définies dans la figure 3.4. chaque trame est un 4-uplet contenant 1. le bus sur lequel la communication est faite, 2. l'identité de l'émetteur, 3. l'identité du récepteur et 4. le message lui-même.

$$
\begin{array}{rcl}
\langle \texttt{frame} \rangle_T & ::= & (\mathit{bus}, \mathit{source}, \mathit{destination}, \langle \texttt{message} \rangle) \\
\langle \texttt{message} \rangle & ::= & (\texttt{block}, \langle \texttt{request} \rangle) \mid (\texttt{nblock}, \langle \texttt{request} \rangle) \\
& \mid & (\texttt{wait}, \mathit{handler}) \\
& \mid & (\texttt{return}, \langle \texttt{response} \rangle) \\
& \mid & (\texttt{return}, \mathit{handler}, \langle \texttt{response} \rangle) \\
\langle \texttt{request} \rangle & ::= & (\texttt{operate}, \mathit{op}, \beta_{in}) \\
\langle \texttt{response} \rangle & ::= & (\texttt{success}, \beta_{out}) \mid (\texttt{failure}, \mathit{text})
\end{array}
$$

FIGURE 3.4: Les trames échangées entre les nœuds d'une topologie T, où $\mathit{bus} \in T.buses$, $\mathit{source}, \mathit{destination} \in T.nodes$, $\mathit{handler} \in \mathcal{H}$ ($\mathcal{H}$ est un ensemble d'identifiants), $\mathit{op} \in \mathcal{OPS}$, $\beta_{in} \in \mathcal{B}^{in}_{op,K,V,L}$, $\beta_{out} \in \mathcal{B}^{out}_{op,K,V,L}$ et *text* est une chaîne de caractères. Une police spéciale dénote les ⟨non terminaux⟩ et les `symboles` (*i.e.*, les constantes).

Les messages peuvent être de quatre types :

`block` ce type de message transmet une ⟨`request`⟩. Il est bloquant dans la mesure où il ne peut plus y avoir de message entre la source et la destination jusqu'à ce que la destination aie répondu avec un message `return` comprenant la ⟨`response`⟩ attendue ;

`nblock` ce type de message transmet une ⟨`request`⟩. Il est non bloquant dans la mesure où il ne bloque l'émetteur que jusqu'à ce que la destination aie répondu avec un `wait`, la ⟨`response`⟩ arrivant plus tard ;

`wait` ce type de message est une réponse à un message `nblock`. Elle transmet un *handler* (identifiant unique) de manière à ce que le récepteur puisse relier sa requête à la réponse qui lui sera transmise plus tard. On suppose que $\mathcal{H}$ est un ensemble suffisemment grand (éventuellement infini) pour assigner un identifiant unique à chaque message `wait` ;

`return` ce type de message transmet une ⟨`response`⟩ à un message ⟨`request`⟩. Une réponse à une message `block` est une paire (`return`, ⟨`response`⟩) ; une réponse à un message `nblock` est un triplet (`return`, *handler*, ⟨`response`⟩), où *handler* est l'identifiant du message `wait` précédent.

Le modèle ne définit qu'un seul type de ⟨`request`⟩, cependant il est conçu pour être étendu et d'autres type peuvent être créés au besoin.

Ce sera d'ailleurs le cas lors d'une de nos expériences présentées dans le chapitre 4.

Une requête $req \stackrel{\text{df}}{=} (\texttt{operate}, op, \beta)$ est paramétrée par une opération $req.op$ et une valuation d'entrée $req.\beta \in \mathcal{B}^{in}_{op,K,V,L}$ pour cette opération. La réponse correspondante, envoyée de manière synchrone ou asynchrone, est une $\langle \texttt{response} \rangle$ qui peut être un `success` ou une `failure`. Dans le premier cas elle transporte la valuation de sortie (notée $resp.\beta$) choisie par le système ; dans le second cas, elle transporte un message d'échec.

Ici, les envois/réceptions ont une relation 1 : 1. On pourrait étendre ou modifier le modèle pour gérer des protocoles plus raffinés. On ne le fait pas ici parceque ce n'est pas le sujet de cette étude de concevoir ni d'étudier des protocoles de communication efficaces. On défini cependant les protocoles `block` et `nblock` afin d'illustrer cette possibilité.

Informations locales

Soit T une topologie. Tout nœud $n \in T.nodes$ est associé à un état $\sigma_n \in \Sigma_{K,V,L}$ et à un effet cummulé $e_n \in \Sigma^2_{K,V,L}$. Autrement dit l'état distribué sur T est une fonction $T.nodes \to \Sigma_{K,V,L} \times \Sigma^2_{K,V,L}$.

On verra plus tard comment ces éléments sont gérés. Pour le moment retenons que les opérations vont modifier l'effet cumulé si elles sont réalisées localement par le nœud et vont potentiellement transférer de l'effet si elles sont transmises à un autre nœud pour être calculées.

3.3 *Interprétations*

Dès que les états sont distribués sur une topologie, il est nécessaire de définir comment composer ces états locaux pour obtenir un état global. On appelle cela une *interprétation*. Cette fonction doit être définie par l'utilisateur avec la topologie.

De plus il faut définir comment un nœud intègre de l'information sur les états qu'il peut déduire de ses échanges avec les autres nœuds. Par exemple, considèrons une hiérarchie mémoire avec un cache qui reçoit une requête pour lire un bloc k. S'il transmet la requête au niveau suivant et reçoit en retour la valeur v associée à k, il sait que (k, v) peut être ajouté à son état local. Plus généralement, à cause de la manière dont les opérations sont définies, la connaissance de l'opération et des valuations d'entrée et de sortie est suffisante pour évaluer $op.effect$. Ce dernier est un effet qui peut être composé avec l'état local. La manière dont cette composition doit être réalisée dépend de la manière d'interpréter l'état distribué et doit également être définie par l'utilisateur.

DÉFINITION 23. *Une* interprétation I_T *d'une topologie* T *est une paire de fonctions* $(globalview, integrate)$ *ayant les signatures suivantes :*

$$\begin{array}{rrcl} globalview: & (T.nodes \to \Sigma_{K,V,L} \times \Sigma^2_{K,V,L}) & \to & \Sigma_{K,V,L} \times \Sigma^2_{K,V,L} \\ integrate: & T.nodes \times T.nodes & \to & \left((\Sigma_{K,V,L} \times \Sigma^2_{K,V,L}) \times \Sigma^2_{K,V,L} \to \Sigma_{K,V,L} \times \Sigma^2_{K,V,L}\right) \end{array}$$

Interprétations et intégration La fonction *globalview* calcule un état global à partir des états associés aux nœuds de $T.nodes$. Elle prend en entrée une fonction qui à chaque nœud de $T.nodes$ associe un couple $\sigma, e \in \Sigma_{K,V,L} \times \Sigma^2_{K,V,L}$ (autrement dit un état distribué). Elle renvoie un couple $\sigma_g, e_g \in \Sigma_{K,V,L} \times \Sigma^2_{K,V,L}$ (l'état global).

La fonction *integrate* est plus complexe. Elle prend en entrée une paire de nœuds (a, b) et retourne une fonction $(\Sigma_{K,V,L} \times \Sigma^2_{K,V,L}) \times \Sigma^2_{K,V,L} \to \Sigma_{K,V,L} \times \Sigma^2_{K,V,L}$. Cette dernière prend en entrée un couple état-effet et un effet et retourne un couple état-effet. Cela doit être compris comme l'intégration d'un effet calculé sur un nœud a à un état d'un nœud b.

La notion d'interprétation recouvre aussi tous les concepts spécifiques dont une politique pourrait avoir besoin. L'exemple étudié au chapitre 4 définiera une notion de hiérarchie au sein de l'interprétation. Cela ajoutera des méthodes spécifiques auxquelles il n'aurait pas de sens de faire référence ici.

3.4 *Le moteur d'exécution*

L'objectif est de fournir un algorithme générique capable d'interroger la politique pour prendre ses décisions. Ces interrogations suivent deux modalités :

1. Étant donné un évenement, à quel module/algorithme dois-je me référer ?
2. Étant donné un état, quels sont les paramètres pour telle ou telle action ?

Initialement, l'objectif était de fournir une boite à outil complète capable de gérer toutes les politiques possibles. Cet objectif se fondait sur l'existence de schémas récurrent dans les politiques de cache. Cependant, il s'est avéré que certaines portions de ces politiques restaient trop spécifiques pour espérer mener un tel projet à bien. Nous avons donc décidé d'intégrer certaines portions algorithmiques dans la politique, auparavant exclusivement intentionnelle.

Cependant, certains schémas récurrents persistent, nous permettant de fournir des éléments génériques.

Notations et conventions Pour des raisons de lisibilité je présenterai ces éléments en pseudo-code. Cependant, leur implémentation utilise des réseaux de Petri colorés dont vous trouverez les schémas en annexe B.1. Je prends les conventions suivantes :

me est le nœud considéré ;

$jobs_{me}$ est le *gestionnaire de requêtes* associé au nœud me ;

T est la topologie. Elle est partagée par tous les nœuds ;

I_T *and* $P^{me}_{I_T}$ sont respectivement l'interprétation (partagée par tous les nœuds) et la politique associée au nœud me ;

σ_{me} est l'état associé à me ;

e_{me} est l'effet cumulé associé à me ;

`chan` est un canal de communication, représenté par une police à chasse fixe. Il peut être global, comme dans cet exemple ou bien local à un nœud. Dans ce cas on le note : $\texttt{chan}_{me}$.

On note $\texttt{chan}.send(msg)$ l'envoie d'un message msg sur un canal `chan` et $\texttt{chan}.receive(msg)$ la réception de ce message sur ce canal. On note $\texttt{chan}.check(msg)$ un test de l'existence de msg dans ce canal, sans le consommer.

On introduit le mot-clé `atomic` dans les algorithmes suivants afin de rendre compte d'une spécificité des réseaux de Petri. En effet si une transition est tirées, les jetons sont produits, traîtés et consommés de manière atomique. De plus, une partie du calcul se fait par résolution de contrainte, ce que la représentation procédurale ne permet pas. Cette notation rend compte de ces spécificités.

Communications Soient deux canaux `io` et `buses`. `buses` est initialisé de manière à contenir l'ensemble des bus de la topologie T. L'envoie et la réception d'une trame représentant un message msg envoyé sur un bus b, d'un nœud me vers un nœud n se passent comme indiqué ci-dessous (en haut l'envoie, en bas la réception). De cette manière on s'assure qu'un bus n'est utilisé que pour le transfert d'un seul message à la fois.

procedure $\texttt{SND}(b, me, n, \langle \texttt{msg} \rangle)$ **:**
- **atomic :**
  - $\texttt{buses}.receive(b)$ *(* prend le bus b */)*
  - $\texttt{io}.send((b, me, n, \langle \texttt{msg} \rangle))$

procedure $\texttt{RCV}(b, n, me, \langle \texttt{msg} \rangle)$ **:**
- **atomic :**
  - $\texttt{io}.receive((b, n, me, \langle \texttt{msg} \rangle))$
  - $\texttt{buses}.send(b)$ *(* relâche le bus b */)*

Pour implémenter les communications bloquantes on introduit un canal $\texttt{idle}_{me}$ par nœud de la topologie T et un canal $\texttt{swaiter}_{me}$. On initialise $\texttt{idle}_{me}$ avec $\{n \mid T[me, n]\}$.

procedure *sendRequestS(b, n, req, h)* **:**
- **atomic :**
  - $\texttt{idle}_{me}.receive(n)$
  - $\texttt{SND}(b, me, n, (\texttt{block}, req))$
  - $\texttt{swaiter}_{me}.send(h, n)$

procedure *receiveResponseS(h)* **:**
- **atomic :**
  - $\texttt{swaiter}_{me}.receive(h, n)$
  - $\texttt{idle}_{me}.send(n)$
  - $\texttt{RCV}(b, n, me, (\texttt{return}, resp))$
- **return** *resp*

procedure *receiveRequestS()* **:**
  atomic :
    $\texttt{RCV}(b, n, me, (\texttt{block}, req))$
    $\texttt{idle}_{me}.receive(n)$
    $h \leftarrow jobs_{me}.add(req)$
    $\texttt{swaiter}_{me}.send(h, n, b)$
    $P^{me}_{I_T}.update(keys(req.\beta), h, req)$
procedure *sendResponseS(resp, h)* **:**
  atomic :
    $\texttt{idle}_{me}.send(n)$
    $\texttt{swaiter}_{me}.receive(h, n, b)$
    $\texttt{SND}(b, me, n, (\texttt{return}, resp))$

Pour implémenter les communications non-bloquantes, on introduit deux canaux $\texttt{awaiter}_{me}$ et $\texttt{awaiter2}_{me}$ par nœud de la topologie.

procedure *sendRequestA(b, n, req)* **:**
  atomic :
    $\texttt{idle}_{me}.receive(n)$
    $\texttt{awaiter}_{me}.send(h, n)$
    $\texttt{SND}(b, me, n, (\texttt{nblock}, req))$
  $receiveWaitA()$
procedure *receiveWaitA()* **:**
  atomic :
    $\texttt{RCV}(b, n, me, (\texttt{wait}, h'))$
    $\texttt{awaiter}_{me}.receive(h, n)$
    $\texttt{awaiter}_{me}.send(h, n, h')$
    $\texttt{idle}_{me}.send(n)$
procedure *receiveResponseA()* **:**
  atomic :
    $\texttt{idle}_{me}.check(n)$
    $\texttt{awaiter}_{me}.receive(h, n, h')$
    $\texttt{RCV}(b, n, me, (\texttt{return}, h', resp))$
  return *resp*

procedure *receiveRequestA()* **:**
  atomic :
    $\texttt{RCV}(b, n, me, (\texttt{nblock}, req))$
    $\texttt{idle}_{me}.receive(n)$
    $h \leftarrow jobs_{me}.add(req)$
    $\texttt{awaiter}_{me}.send(h, n, b)$
    $P^{me}_{I_T}.update(keys(req.\beta), h, req)$
  $sendWaitA(b, n, h)$
procedure *sendWaitA(b, n, h)* **:**
  atomic :
    $\texttt{idle}_{me}.send(n)$
    $\texttt{awaiter}_{me}.receive(h, n, b)$
    $\texttt{awaiter2}_{me}.send(h, n, b)$
    $\texttt{SND}(b, me, n, (\texttt{wait}, h))$
procedure *sendResponseA(resp, h)* **:**
  atomic :
    $\texttt{idle}_{me}.check(n)$
    $\texttt{awaiter2}_{me}.receive(h, n, b)$
    $\texttt{SND}(b, me, n, (\texttt{return}, h, resp))$

Prise en charge de nouvelles requêtes On définit maintenant deux processus : $listener_{me}$ et $dispatcher_{me}$. Le premier a pour rôle d'écouter l'arrivée de requêtes pour le nœud *me*, puis de gérer les spécificités des communications bloquantes ou non-bloquantes selon le type de

message. La requête est ensuite ajoutée au gestionnaire de requêtes.

processus *listener* **:**

```
io.check(bus, src, me, (kind, req))   /* vérifie si un message (kind, req) existe. */
if kind = nblock :                    /* C'est une requête non-bloquante */
    receiveRequestA()
    returns_me.receive(resp, h)
    sendResponseA(resp, h)
elif kind = block :                   /* C'est une requête bloquante */
    receiveRequestS()
    returns_me.receive(resp, h)
    sendResponseS(resp, h)
```

Le second processus ($dispatcher_{me}$) prend en charge les requêtes présentes dans le gestionnaire de requêtes. Il fait appel à la politique du nœud afin de décider quel traîtement lui appliquer.

processus *dispatcher* **:**

```
atomic :
    req, h ← jobs.next()      /* bloque jusqu'à ce qu'un nouveau job soit disponible */
    proc ← P^me_{I_T}.trigger("do " + req.type)   /* demande à la politique quel procédure utiliser pour ce type de requête */
resp ← proc(req, h)
atomic :
    P^me_{I_T}.close (h, resp[0])   /* informe la politique de la réussite ou de l'échec de la requête, resp[0] ∈ {failure, success} */
    jobs_me.done (h)                /* informe le job-manager que la requête h est traîtée */
returns_me.send(resp, h)            /* renvoie la réponse à l'appelant */
```

Notons $p!$ la réplication infinie du processus p. Chaque nœud fait tourner un processus simple qui consiste à mettre en parallèle ces deux processus, répliqués : listener! ‖ dispatcher!

3.5 *Modèle de politiques*

Dans ce modèle, la première des nécessités est de répondre aux requêtes. Quelque soit la procédure sélectionnée par la politique, elle doit toujours renvoyer une réponse. Si c'est le cas, le *listener* et le *dispatcher* nous assurent qu'elle sera renvoyée à l'appelant.

Stockage local et gestion de l'espace La seconde nécessité consiste à gérer l'espace disponible sur les nœuds. Notons que la gestion de l'espace est une action différente du stockage lui-même. Dans notre modèle, le stockage est réalisé par le sous-modèle d'état et la gestion de l'espace est confiée à la politique.

Il existe quatre éléments nécessaires à toute politique de cache : la recherche d'une solution locale, la recherche d'une solution distante le cas échéant, la prise en compte de la solution dans l'état courant,

la capacité à faire de l'espace si nécessaire.

Remarquons qu'un nœud de type *stockage* est un sous-ensemble de ceci : seule la recherche d'une solution locale et son application font partie de sa politique. Si son espace est plein, il renvoie une erreur ; et s'il n'existe pas de solution locale, il en va de même.

Relations avec l'état et l'interpretation L'une des contraintes initiales pour ce modèle était la capacité à rendre la gestion de cache indépendante du type de donnée ainsi que de la topologie.

Il est maintenant possible de considérer les ressources indépendamment de leur structure, et baser la gestion sur la présence ou non de ces ressources, et éventuellement sur leurs relations internes. Il est aussi possible de baser la politique non plus sur la forme particulière que peut prendre la topologie mais sur l'interprétation qu'on en fait.

On a ainsi créé une abstraction permettant la conception de politiques de caches applicables sur des classes de cas plus grandes.

Interface On suit les mêmes conventions que précédemment.

space $(keys \subseteq K, \sigma_{in} \in \Sigma_{K,V,L}) \to \mathbb{N}$

Retourne le nombre de ressources actuellement stockées sur le nœud *me* devant être supprimées afin de pouvoir stocker localement les valeurs associées à *keys*.

update $(keys \subseteq K, handler \in \mathcal{H})$

Cette méthode ne retourne aucune valeur mais est appelée sur le nœud *me* à chaque fois qu'une requête identifiée par *handler* est reçue. elles est utilisée pour notifier la politique de l'intéret porté aux clés dans *keys*. Par exemple, c'est l'occasion de mettre à jour les clés MRU (*Most Recently Used*, plus récemment utilisées) dans un cache de type LRU (*Least Recently used*).

close $(handler \in \mathcal{H}, outcome \in \{\texttt{success}, \texttt{failure}\})$

Cette méthode est utilisée pour valider (sur un `success`) ou annuler (sur une `failure`) les changement apportés par l'appel à update. Ce mécanisme est la raison pour laquelle on utilise les handlers dans ces deux méthodes.

purge $() \to K$

Supprime et retourne un ressource actuellement stockée sur le nœud *me*. Cette ressource devrait être choisie de manière à être celle qui a la plus petite valeur quand purge est appellée. Par exemple, un cache LRU choisira exactement la ressource la moins récemment utilisée.

trigger $(\texttt{str}) \to PROC$

Retourne une procédure associée à un nom d'évennement. Permet à la politique de sélectionner un traîtement particulier pour une action dont le nom est standardisé.

Les méthodes update et close fonctionnent ensemble : update permet d'augmenter l'importance d'un ensemble de clés ; puis appeler close permet de valider ou d'annuler update. La raison d'un tel mécanisme

est que la plupart des opérations sur un nœud ne peuvent être réalisée de manière atomique et peuvent requérir des communications. Pendant ce processus, le nœud peut recevoir et traîter d'autres requêtes pouvant être complétées localement. On ne peut donc pas se baser sur un mécanisme qui verrouillerait tout le nœud pendant qu'on traite une requête.

Notons que lors de notre définition du processus *dispatcher* dans la section précédente, on a défini un événement standard "do " + *req.type*. Le seul type de requête défini pour le moment est `operate`, on obtient alors l'événement "do operate". Toute politique devra définir une procédure à associer à chaque type de requête qu'elle saura gérer.

3.6 *Le gestionnaire de requêtes*

Dans le cadre des systèmes de caches, on veut générer de l'absorption. Autrement dit, on cherche à réaliser le maximum d'opérations à ce niveau pour en transférer le minimum au niveau inférieur. Afin de générer de l'absorption, il est nécessaire de maintenir en mémoire les opérations, dans l'espoir de réduire le nombre de communications nécessaire à la gestion du système.

Dans un cadre plus général, la gestion d'une file d'attente par un serveur de données est un moyen classique pour gérer un grand nombre de clients simultanément. Dans notre cas, cette file d'attente est tout à fait assimilable à un gestionnaire d'opérations.

Pour ces raisons on ajoute à notre système un *gestionnaire de requêtes* dont le rôle recouvre à la fois celui d'une file d'attente, de *scheduler* et permet aussi l'absorption.

Interface Le *gestionnaire de requêtes* est en relation étroite avec l'algèbre d'effets, mais distinct de celle-ci. En effet afin de permettre la concurrence maximale au sein des nœuds de la topologie, on cherche à maintenir un graphe de requêtes qui respecte les dépendances de celles-ci. Il est pour cela nécessaire de pré-évaluer le domaine de ces requêtes.

Lorsqu'une requête est reçue par un nœud de la topologie, elle est d'abord stockée dans le *gestionnaire de requêtes* et associée à un identifiant (qu'on nommera par la suite *handler*) unique pris dans un ensemble $\mathcal{H}$; elle est maintenue jusqu'à ce qu'elle soit complètement gérée. Des dépendances peuvent exister entre les requêtes : deux requêtes r_1 et r_2 sont indépendantes si $\mathsf{keys}(r_1.\beta) \cap \mathsf{keys}(r_2.\beta) = \emptyset$. Le *gestionnaire de requêtes* gère ces dépendances et est accessible par les méthodes suivantes :

$\mathsf{last}\,(\mathit{key} \in K) \to \mathcal{H} \cup \{\text{✘}\}$
: Retourne le handler de la dernière requête ajoutée dont le domaine comprend *key* s'il y a lieu, ou une valeur spéciale ✘ si aucune requête n'est associée à *key*.

$\mathsf{add}\,(\mathit{request} \in \langle \texttt{request} \rangle) \to \mathcal{H}$

Ajoute *request* dans le gestionnaire et retourne un nouvel handler *handler* qui lui sera associé. La requête ajoutée dépend de la dernière requête de chaque clé dans keys(*request*).

next $() \to \langle \texttt{request} \rangle \times \mathcal{H}$

Retourne une paire $(request, handler)$ qui est prête à être traitée (pas de dépendance non traitée). L'appelant est bloqué jusqu'à ce qu'une telle paire soit disponible.

deps $(handler \in \mathcal{H}) \to (\langle \texttt{request} \rangle \times \mathcal{H})^*$

Retourne une liste de paires (r, h) correspondant à toutes les requêtes r et handlers h dont la requête $r_{handler}$ associée à *handlers* dépend. Cette liste est triée de manière cohérente avec les dépendances, le dernier élément étant $(r_{handler}, handler)$, et est calculé de manière déterministe.

done $(handler \in \mathcal{H})$

Marque toutes les informations associées à *handler* comme dispensables et supprime toute information dispensable du *gestionnaire de requêtes*.

3.7 *Simulateur*

Afin de réaliser des expériences avec notre modèle nous avons développé un simulateur l'implémentant.

On utilise le framework SNAKES[32] qui implémente des algèbres de réseaux de Petri, c'est-à-dire des réseaux de Petri qui peuvent être composés comme des termes dans une algèbre de processus[4]. Cela permet de définir de petits réseaux avec des *places tampons* identifiées, puis en composant ces réseaux, les places tampons sont automatiquement fusionnées.

[32] POMMEREAU: "Quickly prototyping Petri nets tools with SNAKES", 2008.

4 Vous pouvez vous référer à l'annexe A pour un rappel sur les réseaux de Petri ainsi que sur la composition de réseaux.

Le simulateur est écrit en python et permet de spécifier tous les éléments présentés dans ce chapitre de manière indépendantes. Il permet de plus de préparer des scénarii qui seront joués séquentiellement.

Une exécution du modèle est une trace du réseau généré par le simulateur et paramétré par les éléments du modèle. Les états successifs de cette trace sont calculés, à partir de l'état initial, en choisissant la transition suivante de manière aléatoire parmi les transitions en mesure de s'exécuter à ce moment.

Le simulateur est équipé d'un module d'inspection avec lequel on peut vérifier des propriétés ou bien réaliser des mesures à la volée. C'est ce simulateur que nous avons utilisé pour réaliser nos expériences dans le chapitre suivant.

4
Expériences

Nous allons maintenant réaliser quelques expériences. Elles serviront d'une part d'exemples sur la manière dont on peut utiliser le modèle présenté au chapitre précédent, et d'autre part à le valider. On pourra observer que le modèle :

1. est modulaire : il ne sera pas nécéssaire de tout redéfinir pour chaque cas ;
2. est opérationel : cette manière de définir nos cas nous permet bien de les exécuter ;
3. se comporte comme on s'y attend : les mesures que l'on réalisera seront conformes à celles qu'on pourra faire sur une implémentation directe de ces cas (appelée ci-après exécution témoin).

À ces fins nous définirons trois cas auxquels nous nous réfererons sous les termes `ACS`, `ACCS` et `ADDS` et qui correspondent respectivement à :

`ACS` un acteur A qui envoie des requêtes à un cache C géré selon la politique LRU qui se réfère lui-même à un stockage S ;

`ACCS` le même acteur A qui envoie des requêtes à un cache C1 géré selon la politique LRU qui se réfère à un autre cache C2, toujours géré selon LRU, qui se réfère à son tour au stockage S précédent ;

`ADDS` notre acteur A qui envoie des requêtes au cache C1 précédent, mais qui cette fois est géré selon la politique DEMOTE ; il se réfère toujours au cache C2 qui est lui aussi géré selon la politique DEMOTE et qui se réfère à notre stockage S.

On peut constater que ces cas sont choisis de manière à montrer la modularité du modèle. Ils partagent tous les nœuds A et S, qu'on ne définira donc qu'une fois. Les nœuds C, C1 et C2 des cas `ACS` et `ACCS` sont tous gérés selon la même politique LRU. Les cas `ACCS` et `ADDS` partagent la même topologie.

Ces cas sont également choisis de sorte qu'ils soient de plus en plus complexes. `ACCS` a une topologie plus complexe qu'`ACS`. `ADDS` à une politique plus complexe qu'`ACCS`.

Ce chapitre sera divisé comme suit :

1. définitions des mesures qui seront réalisées ;

2. définition du modèle de données, du stockage S et de l'acteur A, communs à tous les cas ;
3. définition des cas `ACS`, `ACCS` et `ADDS` ;
4. étude des résultats.

4.1 *Définition des mesures*

Afin d'étudier ces systèmes, nous allons définir quelques critères de performances. Ils nous permettront d'observer les effets des changements de conditions initiales entre les cas étudiés. D'autre part ils nous permettront de comparer le comportement de l'exécution de notre modèle à celui de l'exécution témoin.

Taux de succès

Autrement appelé *hit ratio*, ce taux se définit comme le rapport du nombre de succès d'un cache sur la somme de ses succès et échecs. Un succès se définit comme la capacité d'un cache à répondre à une requête sans en référer à son stockage. Un échec en est bien sûr l'inverse.

Le taux de succès peut être mesuré globalement dans le cas où le stockage associé à un cache est lui-même un cache, comme ce sera le cas dans `ACCS` et `ADDS` entre C1 et C2.

Taux d'absorption

Ce taux mesure le rapport entre le nombre de reqûetes reçues par un stockage S et le nombre de requêtes envoyées par un acteur A. Autrement dit, il mesure la capacité d'un éventuel cache (ou groupe de caches) C situé entre A et S à absorber des requêtes, c'est à dire à les résoudre à la place de S.

On peut penser que le taux d'absorption est similaire au taux de succès. C'est vrai dans les cas simples comme `ACS` ou `ACCS` mais pas nécessairement dans les cas plus complexes comme `ADDS`, comme on le verra lorsqu'on analysera les résultats.

Charge réseau

Cela mesure la quantité de requêtes transmises sur un arc de la topologie.

Redondance

Ce taux mesure la proportion de ressources présente en commun dans deux nœuds. Cela nous sera utile pour expliquer des différences dans les taux de succès entre les cas `ACCS` et `ADDS`.

Temps de réponses

Cette mesure est un peu particulière. Primo, on ne la propose que pour l'exécution de notre modèle et non pour l'exécution témoin.

Cela est dû au fait que le temps dont il est question est un temps simulé qui n'est pas comparable entre ces deux types d'exécutions. Secundo, ce temps simulé ne prétends pas être représentatif du temps réel d'exécution d'une implémentation. Pour la même raison que précédemment, les temps de la simulation et du système réel ne sont pas comparables.

Dans les limites établies ci-dessus, on pourra utiliser ces mesures pour comparer des exécutions du même type. On pourra observer ainsi *qualitativement* l'effet d'un changement de politique (ou de tout autre élément du modèle) sur les performances générales du système.

Ici, on s'intéressera au temps moyen que met le système à renvoyer une réponse aux requêtes d'un acteur. Il faut noter que le temps dont il est question est présenté en unités arbitraires.

En bref, à chaque événement interne du système est associé un coût. Cette association est appelée fonction de coût et est un paramètre de la mesure. Chaque événement est ajouté à un graphe et associé d'une part au nœud de la topologie sur lequel il a lieu, et d'autre part à tous les événements dont il dépend. Ces événements sont le dernier événement associé au nœud s'il y en a, et l'événement correspondant à l'envoie d'un message si l'événement ajouté est la réception de ce message.

Le temps de réponse d'un message est la somme des coûts des événements du chemin le plus long de l'envoie d'une requête à la réception de sa réponse.

Afin que les effets soient bien visibles on choisit la fonction de coût suivante : chaque événement se voit associé un coût de 1 multiplié par le rang dans la topologie du nœud sur lequel il survient, en commençant par l'acteur dont le rang est 0. Les événements correspondant à des communications voient leur coût multiplié par 10.

4.2 *Opérations, stockage et acteur*

Dans cette section nous allons définir les éléments qui seront communs à toutes nos expériences. Le modèle de données nous permettra de définir les opérations que devra gérer notre système, donc le scénario que jouera l'acteur A. Il nous permettra aussi de définir les états initiaux de tous les nœuds de la topologie, en particulier celui du stockage S. On définira ensuite une fois pour toute le stockage : son état initial et sa politique. Enfin on définira l'acteur A : son état initial, sa politique et le générateur de scénario.

Opérations

Bien que les politiques présentées plus bas soient indépendantes du modèle de données utilisé, l'activité générée en sera, elle, complè-

tement dépendante. Notre scénario sera une séquence de requêtes `operate` forgées à partir de deux opérations, *read* et *write* définies comme suit.

L'opération *read* récupère la valeur v associée à une clé donnée k. L'opération *write* remplace la valeur v_1 associée à la clé k par la valeur v_2 donnée en argument. Plus formellement :

nom	garde	effet	paramètres
`read`	$(k, v) \in \sigma.base.h$	$\left(\begin{smallmatrix}(k,v) & \emptyset \\ \emptyset & \emptyset\end{smallmatrix}\right)$	$\{k\}$
`write`	$(k, v_1) \in \sigma.base.h$	$\left(\begin{smallmatrix}(k,v_2) & (k,v_1) \\ \emptyset & \emptyset\end{smallmatrix}\right)$	$\{k, v_2\}$

Stockage S

État initial Pour nos exemples, on génère l'état initial du système en même temps que le scénario qui sera exécuté. Le stockage sera la base de l'état distribué. On le génère à partir d'un nombre de clés *numkeys*. Sur cette base on peut générer un ensemble K de clés en prenant par exemple les *numkeys* premiers entiers de $\mathbb{N}$: $K \stackrel{\text{df}}{=} \{n \in \mathbb{N} | n < numkeys\}$. On prendra également soin de générer un ensemble non vide V de valeurs.

On se donne comme état initial un effet e nul et un état s dont la composante $s.R$ est nulle et la composante $s.H$ est une fonction de $K \to V$. De cette manière, que les requêtes soient des lectures ou des écritures, le stockage sera toujours en mesure de répondre.

Les ensembles K et V seront réutilisés pour générer le scénario.

Politique La politique P_S du stockage est assez simple. Pour se faciliter la tâche, on considérera soit que le stockage à une taille infinie, soit qu'il ne recevra que des requêtes concernant des ressources qu'il possède déjà et n'aura donc pas besoin de plus d'espace (ce qui sera le cas en pratique ici). La politique n'aura donc pas besoin de gérer les ressources.

Grossièrement, le stockage devra tenter de répondre aux requêtes `operate` qu'il recevra. Dans l'impossibilité de définir une réponse adéquate, il renverra un échec.

Plus formellement, P_S se définit comme suit :

space $(keys, \sigma_{in})$: 0 ;
update $(keys, handler)$: ✘ ;
close $(handler, outcome)$: ✘ ;
purge $()$: ✘ ;
trigger $(\texttt{str})$: *TryLocal* if `str` = "do operate" else ✘ ;

avec :

procedure *TryLocal(req, h)* **:**
  if *req.rtype != "operate"* **:**
    return FailureResponse("Request type not recognized.")
  $c \leftarrow req.op.candidates(\sigma_{me}, req.\beta_{in})$
  if *c !=* [] **:**
    resp = SuccessResponse(c[0])
    if $P^{me}_{I_T}.space()$ *!=*0 **:**
      resp = FailureResponse("Not enough space.")
    $int \leftarrow I_T.integration(me, me)$
    $effect \leftarrow req.op.effect(req.\beta_{in}, resp.\beta_{out})$
    $\sigma_{me} \leftarrow int(\sigma_{me}, effect)$
    $e_{me} \leftarrow e_{me} + effect$
    $jobs_{me}$.done(h)
  else :
    $jobs_{me}$.done(h)
    resp = FailureResponse("Could not find a binding out.")
  return resp

Acteur A

Pour produire de l'activité, on introduit des nœuds dédiés, appelés *acteurs*. Leur seul rôle est d'envoyer des requêtes et d'en recevoir les réponses. Par exemple, un processeur est un acteur dans une hiérarchie mémoire.

Il n'est pas possible de définir un modèle d'acteur générique ; chacun correspond à un profil d'activité particulier qui agira sur le système d'une manière spécifique. C'est la raison pour laquelle il existe sur le marché des traces d'accès correspondant à divers cas d'utilisation.

Ici, nous ne nous interessons pas à ces profils mais il est important d'en considérer l'existence. Le choix d'un modèle adéquat est crucial pour une analyse statistique correcte. En effet la plupart des politiques de cache se fondent sur des hypothèse fortes quant aux profils d'accès des systèmes qui les utilisent. Pour les exemples suivant, nous allons définir un générateur de scénario.

Choix des opérations Le scénario est généré requête par requête. Chacune de ces requête sera un `read` avec une probabilité P_r ou un `write` avec une probabilité $1 - P_r$.

Choix des resources cibles Pour chaque requête du scénario, une fois l'opération choisie, il faut définir ses paramètres. En particulier la clé sur laquelle elle s'appliquera.

Afin d'observer des effets de cache clairs, on génerera la liste de ces clés selon une distribution adaptée aux politiques de cache tests définies plus bas. Celles-ci ayant pour base LRU, cette distribution devra choisir une clé avec une probabilité d'autant plus forte que cette clé aura été désignée récemment.

À cette fin on maintiendra une liste LRU L de clés $k \in K$ de taille $T_{LRU} < card(K)$. On générera pour chaque nouvelle requête un indice i suivant une distribution exponentielle[1] de paramètre λ. On choisira ce λ de sorte que la probabilité de tirer un indice entre 0 et T_{LRU} soit P_{SDD}, donné en paramètre. Si l'indice est plus grand que T_{LRU}, on choisira la clé dans $K \setminus L$ et on l'ajoutera en tête de L ; on en supprimera le dernier élément si L devenait plus grand que T_{LRU}.

1 Cette distribution étant continue, on prendra à chaque fois l'entier inférieur le plus proche.

Pour résumer, les paramètres de notre générateur de clés sont :

— K, l'ensemble des clés de notre exemple ;
— $T_{LRU} < card(K)$, la taille de la liste LRU de clés dans laquelle on espère tirer une clé ;
— P_{SDD}, la probabilité avec laquelle on souhaite tirer une clé dans notre liste LRU.

Distribution exponentielle On choisit cette distribution car sa densité de probabilité correspond à notre attente : grande proche de 0, elle s'amenuise en allant vers $+\infty$. Un indice généré selon cette loi sera donc souvent plus proche du MRU que du LRU.

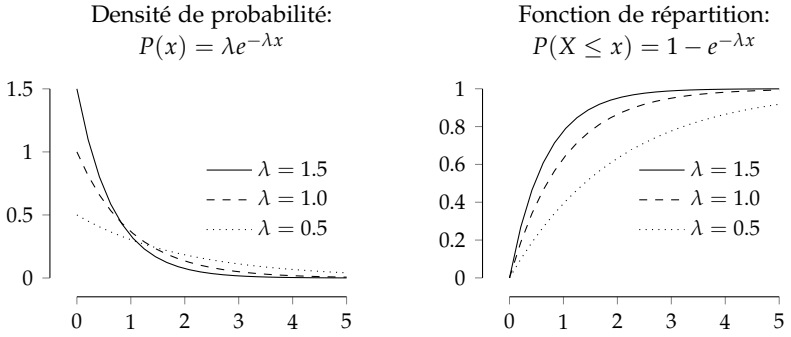


FIGURE 4.1: Densité de probabilité et fonction de répartition de la loi exponentielle.

Pour générer des nombres aléatoires en respectant la loi exponentielle, on applique la méthode d'inversion. Il suffit de génerer des nombres aléatoires suivant la loi uniforme sur $[0, 1]$ et d'appliquer à chaque nombre trouvé la fonction inverse de la fonction de répartition : $gen(\lambda) = \frac{-log(1-uniform(0,1))}{\lambda}$. On obtiendra notre indice en en prenant l'entier inférieur le plus proche : $indice(\lambda) = floor(gen(\lambda))$.

Calculer le paramètre λ On veut que la probabilité de générer un indice $x \leq T_{LRU}$ soit égale à P_{SDD}.

La fonction quantile renvoie, pour une probabilité p donnée dans la répartition d'une variable aléatoire, la valeur à partir de laquelle la probabilité de la variable aléatoire X sera inférieure ou égale à p. Autrement dit, avec F une fonction de répartition d'une variable aléatoire X : $Q(p) = x$ tel que $P(X \leq x) = p$. La fonction quantile de de la loi exponentielle est $Q(p, \lambda) = \frac{-log(1-p)}{\lambda}$ et on cherche λ tel que $Q(P_{SDD}, \lambda) = T_{LRU}$, donc :

$$\lambda = \frac{-log(1 - P_{SDD})}{T_{LRU}}$$

Générer les clés et le scénario On a maintenant tous les éléments pour générer le scénario. Dans ces exemples il sera lu séquentiellement mais on peut envisager d'intégrer de la concurrence à ce niveau sans difficulté.

Processus de l'acteur Le scénario est composé de requêtes de type `operate`. Elles sont ajoutées au fur et à mesure dans le job-manager comme si elles avaient été reçues par le listener. Le processus au niveau de l'acteur devient :

$$\mathsf{listener!} \parallel \mathsf{worker!} \parallel \mathsf{scenarioReader}(\mathsf{scenario})$$

Le processus scenarioReader lit une liste de requêtes et les ajoute à son gestionnaire de requêtes afin qu'elles soient traitées.

processus *scenarioReader(scenario)* **:**
  for $req \in scenario$ **:**
    $h \leftarrow jobs_{me}.add(req)$
    $P^{me}_{I_T}.update(keys(req.\beta), h, req)$
    $\texttt{returns}_{me}.receive(resp, h)$

État initial L'acteur, comme d'ailleurs les caches, débutera avec un état initial vide.

Politique La politique de l'acteur P_A est la suivante :

space $(keys, \sigma_{in})$: 0 ;
update $(keys, handler)$: ✘ ;
close $(handler, outcome)$: ✘ ;
purge $()$: ✘ ;
trigger $(\texttt{str})$: *TryTransfert* if $\texttt{str} =$ "do operate"
  else ✘ ;

procedure *TryTransfert(req, h)* **:**
  $l \leftarrow jobs_{me}.getdeps(h)$
  $last, i \leftarrow synchronizeOperate(l, h)$
  $jobs_{me}.done(h)$
  return *last*

Grossièrement, l'acteur A ne modifie jamais son état et transfert systématiquement les requêtes issues du scénario à son stockage référant.

4.3 *Définition des cas ACS, ACCS et ADDS*

Les éléments communs à tous nos cas étant définis, on peut maintenant s'intéresser à ce qui les différencie.

Comme dit précédemment, ces cas ont été définis de manière à illustrer la modularité du modèle et à monter progressivement en complexité. Les caches C, C1 et C2 des cas ACS et ACCS partagent la même politique : LRU. Les cas ACCS et ADDS partagent la même topologie et la même interprétation, qui sont toutes deux légèrement plus complexes que pour le cas ACS. Enfin la politique DEMOTE des caches de ADDS est sensiblement plus complexe que LRU.

Premier cas : `ACS`

Topologie Notre premier exemple utilise une topologie simple comprenant un acteur A, un cache C et un stockage S. Ces nœuds sont arrangés sur la topologie suivante :

$$T_1 \stackrel{\text{df}}{=} (\{A, C, S\}, \{\{A, C\}, \{C, S\}\})$$

Interprétation Les politiques mises en jeu ici se basent sur une interprétation hiérarchique de la topologie. Plus précisément, elles se basent sur une interprétation *en pile* de la topologie, ce qui est un cas particulier de hiérarchie. Elle se définit comme suit :

— il existe une relation d'ordre total entre les nœuds du système ; on peut toujours identifier un nœud de stockage par rapport à tous les nœuds sauf un qui est celui le plus bas ;

— l'état global est obtenu en projetant les niveaux supérieurs sur les niveaux inférieurs ;

— l'intégration projette un effet observé sur l'état local.

Plus formellement, I_1 se définit comme suit :

globalview	$\{(A, \sigma_A, e_A), (C, \sigma_C, e_C), (S, \sigma_S, e_S)\} \mapsto (e_A \gg e_C) \gg \sigma_S$
integrate	$me, pos \mapsto \begin{cases} \sigma_{me}, e_{pos} \mapsto \sigma_{me} & \text{if } me = A, \\ \sigma_{me}, e_{pos} \mapsto e_{pos} \gg \sigma_{me} & \text{otherwise.} \end{cases}$
relativestorage	$me \mapsto \begin{cases} C & \text{if } me = A, \\ S & \text{if } me = C, \\ \text{✘} & \text{otherwise.} \end{cases}$

Éléments de politique On définit deux procédures qui seront réutilisées au sein des différentes politiques.

La procédure *synchronizeOperate* transmet dans l'ordre toutes les requêtes (de type `operate`) d'une liste l au nœud qui correspond au stockage du nœud *me*.

La procédure *makeRoomOperate* supprime de l'état courant n ressources, en prenant soin de synchroniser toutes les opérations en attente qui concernent les-dites ressources.

procedure *synchronizeOperate(l, h)* **:**
  $i \leftarrow I_T.relativestorage(me)$
  for $req, h_2 \in l$ **:**
    atomic :
      `buses`$.receive(b)$
      $b[i, me]$
      $sendRequest(b, i, req, h_2)$
      $last \leftarrow receiveResponse(h_2)$
  return $last, i$

procedure *makeRoomOperate(n)* **:**
  while *n>o* **:**
    atomic :
      $least \leftarrow P^{me}_{I_T}.purge()$
      $h_2 \leftarrow$
      $jobs_{me}.key2lastRequestHandler(least)$
      $l \leftarrow jobs_{me}.getdeps(h2)$
    $last, i \leftarrow synchronizeOperate(l, h_2)$
    atomic :
      $\sigma_{me} \leftarrow \sigma_{me}/least$
      $e_{me} \leftarrow e_{me}/least$
      $jobs_{me}.done(h2)$
      $n \leftarrow n - 1$

Politique La politique du cache P_C utilise une liste de type LRU pour choisir la ou les ressources à évincer.

Le point important à retenir ici est que $P_C.trigger(\text{"do operate"})$ répond $TryLocalElseTransfert$ et ✘ pour toute autre entrée.

procedure *TryLocalElseTransfert(req, h)* **:**
  $resp, i \leftarrow TryLocal(req, h)$
  if $resp.type \neq Success$ **:**
    $resp, i \leftarrow TryTransfert(req, h)$
    if $resp.type = Success$ **:**
      $int \leftarrow I_T.integration(me, i)$
      $effect \leftarrow req.op.effect(req.\beta_{in}, resp.\beta_{out})$
      $\sigma_{me} \leftarrow int(\sigma_{me}, effect)$
      $e_{me} \leftarrow e_{me} + effect$
  return $resp, i$

Grossièrement, $TryLocalElseTransfert$ tente d'abord de réaliser la requête localement (comme le stockage S le ferait). En cas d'échec, elle tente de transférer la requête à son stockage référent (ici S), puis en cas de succès elle intègre le résultat et retourne la réponse.

Deuxième cas : `ACCS`

Topologie Notre deuxième exemple utilise une topologie similaire à celle d'`ACS`, comprenant un acteur A, un cache $C1$, un cache $C2$ et un stockage S. Ces nœuds sont arrangés sur la topologie suivante :

$$T_2 \stackrel{\text{df}}{=} (\{A, C1, C2, S\}, \{\{A, C1\}, \{C1, C2\}, \{C2, S\}\})$$

Interprétation Les politiques mises en jeu ici se basent comme précédemment sur une interprétation en pile de la topologie. I_2 se définit comme suit :

globalview	$\{(A,\sigma_A,e_A),(C1,\sigma_{C1},e_{C1}),(C2,\sigma_{C2},e_{C2}),(S,\sigma_S,e_S)\} \mapsto (e_A \gg e_C) \gg \sigma_S$
integrate	$me,pos \mapsto \begin{cases} \sigma_{me},e_{pos} \mapsto \sigma_{me} & \text{if } me = A, \\ \sigma_{me},e_{pos} \mapsto e_{pos} \gg \sigma_{me} & \text{otherwise.} \end{cases}$
relativestorage	$me \mapsto \begin{cases} C1 & \text{if } me = A, \\ C2 & \text{if } me = C1, \\ S & \text{if } me = C2, \\ \text{✘} & \text{otherwise.} \end{cases}$

Politiques Les politiques de C1 et C2 sont identiques à celle de C dans `ACS` : $P_{C1} = P_{C2} = P_C$.

Troisième cas : `ADDS`

Dans ce cas la topologie et l'interprétation seront les mêmes que pour le cas `ACCS`. Seules les politiques changeront.

Éléments de politique On intègre des procédures similaires à celles développées précédemment.

synchronizeDemote est l'équivalent de *synchronizeOperate*, mais au lieu d'envoyer une séquence de requêtes `operate`, elle envoie une seule requête `demote`. Cette requête est composée d'un effet et d'un graphe de requêtes correspondant aux dépendances de la ressource dont on se débarrasse.

La procédure *makeRoomDemote* est similaire à *makeRoomOperate*, mais elle utilise *synchronizeDemote* et non *synchronizeOperate* pour faire de la place.

La procédure *tryPatch* applique une requête `demote` et fait de la place si nécessaire. Appliquer une requête `demote` consiste à projeter son effet sur l'état local et intégrer son graphe de requêtes au gestionnaire de requêtes local.

Politique Les politiques de C1 et C2 diffère l'une de l'autre. En effet C1 devra lancer *makeRoomDemote* lors de l'évenement "on make room" et C2 continuera à lancer *makeRoomOperate* pour le même évenement. C1 comme C2 devront lancer $TryLocalElseTransfert$ à "do operate" et C2 devra en plus lancer *TryPatch* pour "do demote". C1 devra pousser les ressources des requpetes `operate` en tête de LRU, C2 devra supprimer les ressources ciblées par les `operate` de sa LRU et ajouter les ressources ciblées par les `demote` en tête de LRU.

4.4 *Résultats*

On peut déjà constater que le modèle défini au chapitre précédent nous a permis de représenter complètement les cas qui nous intéressaient. Nous avons utilisé notre implémentation du modèle afin d'exécuter ces cas, ce qui nous permet de dire que le modèle est opérationnel et pas exclusivement théorique.

Afin de valider notre modèle, nous avons réalisé une implémentation directe des cas présentés plus haut. Nous les avons instrumentés de manière équivalente à notre moteur d'exécution de manière à en retirer les mesures présentées en début de chapitre.

Maintenant que tous les cas sont définis, on cherche à vérifier en premier lieu que l'implémentation de notre modèle à un comportement équivalent à celui de l'implémentation témoin. Ensuite on présentera et expliquera les résultats obtenus, certains d'entre eux pouvant paraître étonnants.

Paramètres Les courbes présentées ci-après ont été obtenues en faisant dérouler une cinquantaine de simulations pour chaque taille de cache de 1 à 80 pour `ACS` et de 1 à 40 pour chaque cache de `ACCS` et `ADDS`, soit une taille globale de cache de 2 à 80. Le simulateur était paramétré comme suit :

scenar_len	200
numkeys	2000
T_{LRU}	30
P_{SDD}	0.5
P_r	0.7

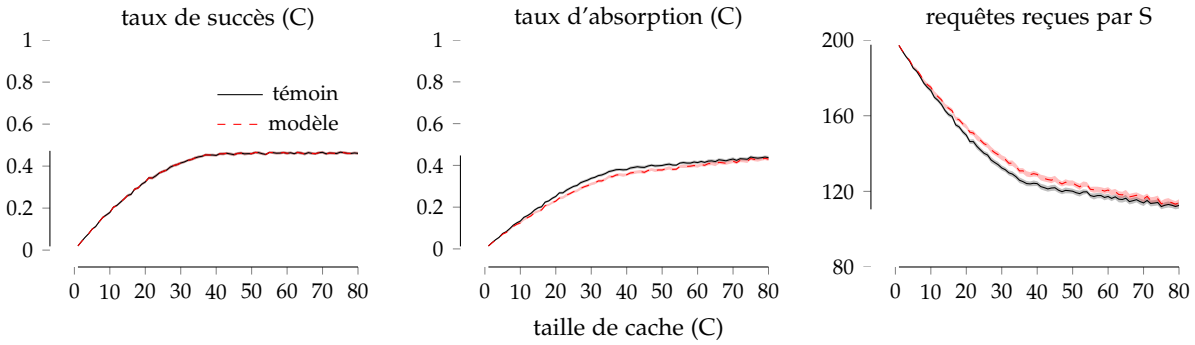
Comparaisons

Les graphes ci-dessous montrent diverses mesures réalisées d'une part sur les exécutions de notre modèle (en rouge et pointillés) et d'autre part sur les exécutions d'un code témoin implémentant directement les algorithmes modélisés (en noir plein). Ce qui nous intéresse ici ce ne sont pas tant les résultats que la proximité, voire l'identité, des résultats obtenus grâce à ces deux codes.

Les courbes ci-dessous correspondent à la moyenne des résultats obtenus pour chaque cas. On a aussi voulu représenter la précision de ces résultats en accompagant ces courbes de barres d'erreurs. Pour des raisons de lisibilité on a remplacé les barres par des surfaces au sein desquelles on trouve quatre-vingt-quinze pourcent des résultats, autour de la moyenne. Comme vous le verrez la précision est très bonne.

`ACS` Cette première série concerne le cas `ACS`.
Les taux de succès sont identiques entre le témoin et le modèle. Les taux d'absorptions et le nombre de requêtes reçues par S sont très légèrement différentes. Cette différence s'explique par une particularité du modèle par rapport au témoin.

Dans le témoin, le caractère synchrone ou asynchrone d'une ressource par rapport au stockage est repéré, mais n'est utilisé pour choisir ou non de synchroniser que lors de l'éviction de la-dite ressource. Dans le modèle, cette information est représentée comme une opération en attente, susceptible d'être absorbée mais belle et bien en attente. Le moteur d'exécution peut alors choisir de lancer le pro-

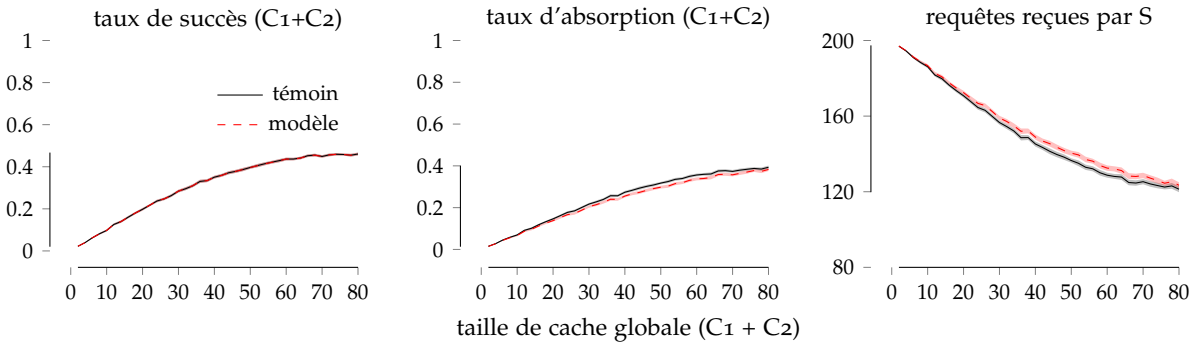


cessus de syncronisation sans y être contraint, ce qui a tendance à arriver.

Ainsi le cache synchronise plus d'opérations que ce qui est strictement nécessaire et donc le taux d'absorption est moins bon sur le modèle, et le nombre de requêtes transmises à S est plus grand. Cela n'a en revanche aucune incidence sur le taux de succès de C. Les ressources sont simplement plus synchronisées, mais pas plus évincées.

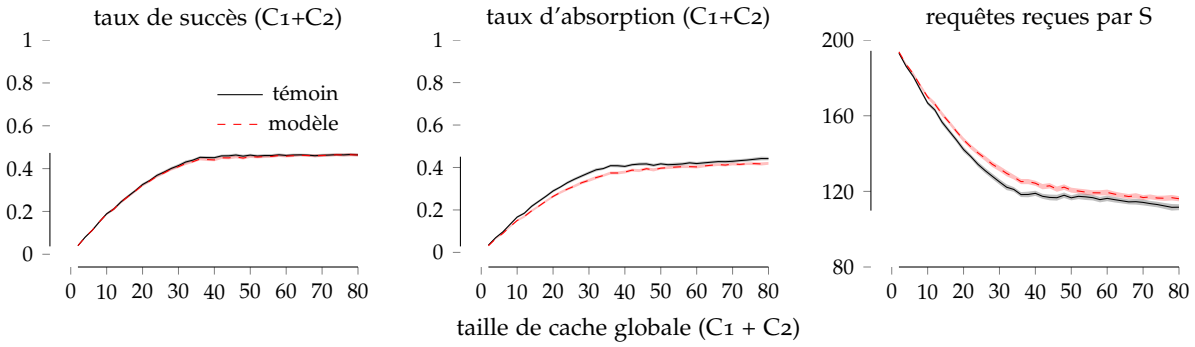
Ces résultats valident que notre modèle est correct pour ce cas. Voyons maintenant si on peut renouveler cela avec des cas plus complexes.

ACCS Cette deuxième série concerne le cas ACCS. Les taux de succès et d'absorption sont mesurés globalement sur les deux caches C1 et C2.



Comme pour l'exemple précédent, le modèle et le témoin se comportent de manière identique pour le taux de succès et très proche pour les taux d'absorption et le nombre de requêtes transmises à S. Ces différences s'expliquent exactement comme pour le cas ACS.

ADDS Cette troisième série concerne le cas ADDS. Là encore les taux de succès et d'absorption sont là encore mesurés globalement.
Mêmes remarques que précédemment : les résultats sont identiques pour le taux de succès et les différences pour le taux d'absorption et le nombre de requêtes transmises à S s'expliquent très bien.



Conclusion Deux choses apparraissent clairement des courbes présentées ci-dessus. La première c'est que le modèle défini permet effectivement de décrire des cas et de les exécuter. La seconde c'est que les éxécutions de ces cas sont correctes au regard des exécutions correspondantes de l'implémentation témoin.

Celà doit nous pousser à utiliser ce modèle afin de réaliser des études inédites. Par exemple on pourrait définir un tout autre modèle de données sans rien toucher d'autre et étudier le comportement de nos politiques dans ce nouveau contexte. Dans la suite de ce chapitre, nous nous contenterons de détailler un peu plus nos résultats.

Étude des résultats

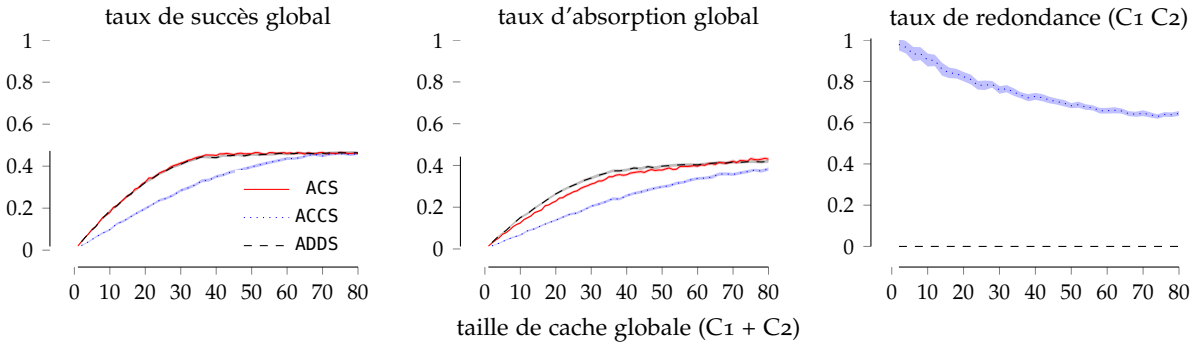
Certains de ces résultats, sans être inédits, méritent qu'on s'y arrête. Les résultats mettant en jeu le taux d'absorption sont, eux, nouveaux puisque cette mesure à été développée lors de cette thèse.

Voyons d'abord quelques résultats connus et/ou prévisibles. On n'utilise à partir de ce point que les résultats obtenus à partir de notre modèle, et plus ceux du témoin.

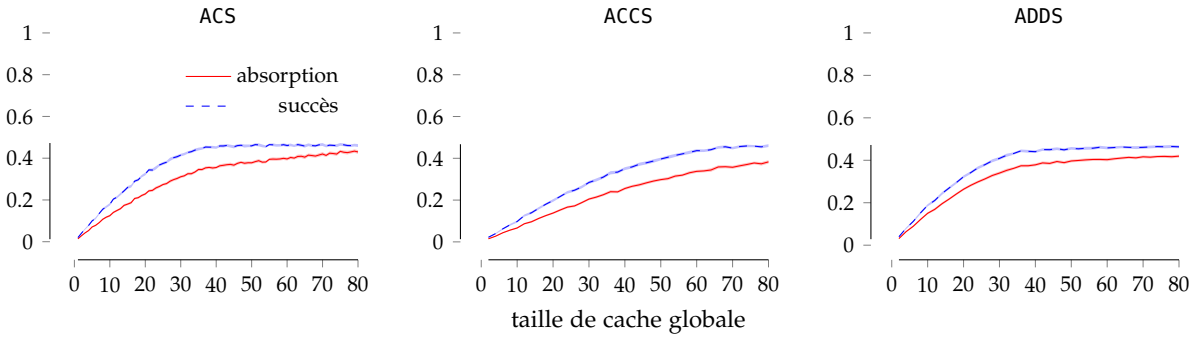
Taux de succès de `ACS`, `ACCS` et `ADDS` Un premier résultat rassurant est que le taux de succès global du cas `ADDS` est toujours supérieur à celui d'`ACCS`. C'était tout le propos derrière le développement de cette politique : supprimer la redondance entre les niveaux de cache pour améliorer le rendement. Le troisième graphique de cette série nous confirme que le taux de redondance entre les caches C1 et C2 varie entre 0.6 et 1 pour `ACCS` et reste à 0 pour le cas `ADDS`.

Cela mène à un second résultat tout aussi prévisible et rassurant : pour une taille de cache globale T, `ADDS` se comporte exactement comme `ACS` avec la même taille de cache T.
On peut noter au passage qu'on retrouve là de manière plus explicite un résultat de 1992 [28] traitant du fait que les hiérarchies de caches avaient tendances à devenir redondantes. Plutôt que d'améliorer les performances, la présence du cache superflu se traduit par un ralentissement du point de vue utilisateur.

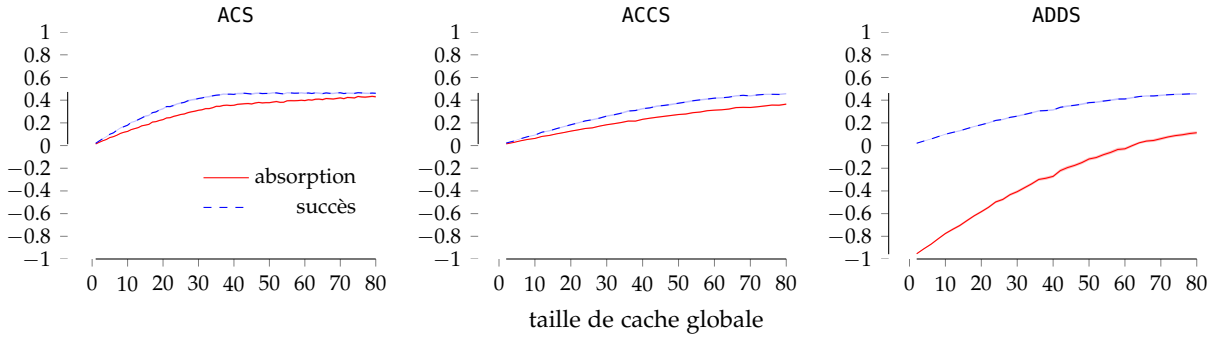
Taux d'absorption Un deuxième résultat, inédit celui-là, mérite qu'on s'y attarde. Jusqu'ici le taux de succès et le taux d'absorption bien



que différents, sont extrêmement proches. Il faut rappeler qu'on s'intéresse là aux taux globaux. Le taux d'absorption est mesuré entre ce qui est envoyé par l'acteur et reçu par le stockage. Le taux de succès est soit local à C pour `ACS` ou global entre C1 et C2 pour `ACCS` et `ADDS`.



Mais si on s'intéresse à des taux locaux, on peut voir que les taux d'absorption et de succès ne sont plus forcément proches. La série suivante s'intéresse à ces mêmes taux sur C pour `ACS` et C1 pour les deux autres cas. On observe pour le cas `ADDS` un taux d'absorption négatif, quand le taux de succès reste évidemment positif.



Cela s'explique par le fait que la politique DEMOTE ait le comportement suivant :

— les échecs sur le cache C1 produisent un transfert des requêtes ayant échouées sur C2, comme pour `ACCS` ;

— chaque éviction de C1 (à chaque échec quand le cache est plein) génère en plus une nouvelle requête demote entre C1 et C2.

Ainsi le traffic inter-cache peut devenir plus grand que le nombre de requêtes initialement envoyées par A, au lieu de devenir plus petit. C'est ce que remarquait l'auteur de la technique PROMOTE [12] dans son article, et qu'il corrigeait avec la-dite technique (non étudiée ici).

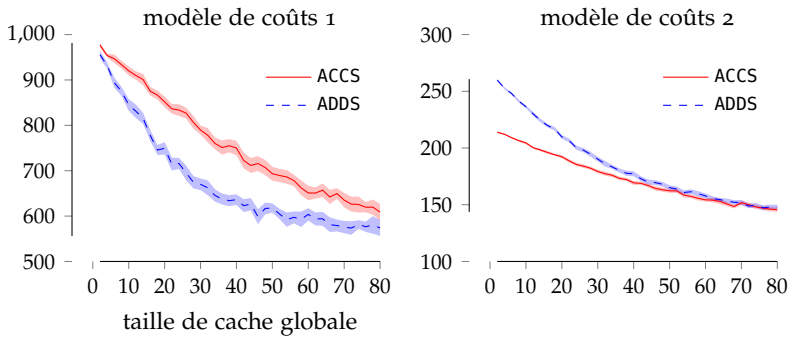
Ce qui est intéressant ici, c'est que le taux d'absorption nous permet de mesurer la qualité d'un cache de manière complémentaire au taux de succès.

Temps de réponses Intéressons-nous maintenant aux temps de réponses observés par le nœud A dans les cas `ACCS` et `ADDS`. On rappelle que ces coûts sont en unité arbitraire. Ils sont choisis ici de manière à observer des changements probants entre les différents cas. Pour un système réel, une étude devra être conduite afin de déterminer des coûts réalistes.

Observons maintenant l'effet du changement de politique entre `ACCS` et `ADDS` sur le temps de réponse perçu par A. On s'attend à ce que le cas `ADDS` produise des temps de réponses plus petit que le cas `ACCS`. Nous présentons ci-dessous deux courbes de temps de réponses.

Le premier modèle de coûts est paramétré comme suit : — chaque transition ayant lieu sur le nœud A vaut 0 ; — chaque transition sur les nœuds C1 et C2 valent 1 ; — les transitions sur le nœud S valent 10 ; — les transitions correspondant à des communications valent 0 pour A, 10 pour C1 et C2, et 100 pour S..

Le second modèle de coûts est paramétré comme suit : — chaque transition ayant lieu sur le nœud A vaut 0 ; — chaque transition sur les nœuds C1, C2 et S valent 1 ; — les transitions correspondant à des communications valent 0 pour A, 10 pour C1, C2 et S.



On remarque deux point intéressants. 1. Selon le modèle de coûts choisi, les temps de réponses peuvent être meilleurs sur `ACCS` que sur `ADDS` ou l'inverse. 2. Les valeurs ne sont pas comparables entre deux modèles de coûts.

Concernant le premier point, deux éléments entrent en jeu pour expliquer ce phénomène. Comme dit plus haut, la politique DEMOTE génère un traffic intercache important. Ce traffic ayant un coût, il faut que le gain assuré par une redondance nulle entre les caches soit plus important que ce surcoût. C'est le cas lorsque les coûts de traitement et de communication sont de plus en plus forts à mesure que l'on

se rapproche du nœud S (modèle 1), mais pas lorsque les coûts sont similaires entre les caches et S (modèle 2).

Ces deux points nous invitent à nous méfier des comparaisons de performances que l'on peut réaliser. Ces comparaisons ne peuvent être valides qu'étant donné un modèle de coût quand il s'agit de simulation ou des spécificités d'implémentation ou de matériel quand il s'agit d'un système réel.

Troisième partie
Ouvertures

5 Discussions 71

5.1 Justifications 71

5.2 Résultats 74

5.3 Ouvertures et perspectives 77

6 Conclusion 83

« *Nous ne pouvons voir qu'à courte distance devant, mais nous pouvons voir là bien des choses qui doivent être faites.*

. . . . . .

We can only see a short distance ahead, but we can see plenty there that needs to be done. »

A. Turing, *"Computing machinery and intelligence". in : Mind – A Quarterly Review of Psychology and Philosophy, vol. 59, #236, pp. 433-460.*, 1950

5
Discussions

Maintenant que nous avons développé notre modèle de stockages distribués et que nous en avons illustré l'usage au chapitre précédent, il est temps d'en discuter un certain nombre d'aspects.

Dans ce chapitre on tâchera de justifier certains choix qui peuvent être étonnants par rapport aux usages courants ; on fera ensuite le point sur les résultats importants de la thèse ; enfin, on discutera des ouvertures permises par ces travaux, des notions qu'il reste à aborder et des améliorations envisageables du modèle et de l'implémentation.

5.1 Justifications

Commençons par justifier certains choix étonnants du modèle. Dans le chapitre 2 nous avons développé les points de réflexion ayant le plus contribué à la forme du modèle présenté. Il constitue déjà une première partie aux justifications qui vont suivre et aurait pu être intégré à cette section.

Lectures et opérations

Le premier choix qui porte à discussions est celui qui fait de tout accès à un système de données une opération, y compris les lectures dans des systèmes de type mémoire. Cet étonnement vient du fait que ce type d'accès en particulier n'a pas d'effet de bord ; les données ne sont jamais modifiées par ce type de lectures. Comment alors les considérer comme des opérations ? Il y a deux raisons principales à celà.

Tout d'abord, mettons en perspective ce cas par rapport à des systèmes de données plus complexes, tels que des systèmes de fichiers.

Dans un système de fichier, il existe une douzaine d'opérations. La plupart opère exclusivement sur la structure arborescente du système (les méta-données et l'organisation des fichiers). Ce sont la création de fichier, le renommage, la destruction, le changement de droits, etc. D'autres agissent sur le contenu des fichiers : ce sont les opérations de lecture et écriture, qui sont les mêmes que dans les systèmes de mémoire.

On peut se poser la question suivante : pourquoi, dans cet ensemble d'opérations cohérent devrait-on considérer la lecture à part ? Autrement dit : le fait qu'elle n'aie pas d'effet de bord suffit-il à ce que ce ne soit pas une opération ?

À notre connaissance, les concepteurs de systèmes de fichiers ne se posent pas cette question. Ils intègrent la lecture avec les autres de manière égale. Les lecteurs venant de cette communauté seront donc peut-être déroutés par ce débat, mais il existe d'autres spécialités qui ne considèrent pas la lecture comme une opération. En tout état de cause, le choix qui a été fait pour le modèle est de considérer de manière égale tout accès au système de données en tant qu'opération.

Il y a une seconde raison à ce choix : c'est le rapport qu'entretiennent les opérations avec la notion d'absorption, qui est formalisée pour la première fois dans ce document.

En effet, comme il a été dit au chapitre 2, le cache est avant tout un mécanisme d'absorption. Un cache simple de lecture, lorsqu'un accès est fait sur une ressource qu'il ne gère pas, va transmettre la requête à son stockage puis maintenir la valeur de la ressource dans l'espoir que d'autres accès seront réalisés pour cette ressource. Dans ce cas il n'aura plus besoin de transmettre la requête. En y répondant directement, il aura absorbé une lecture. Un cache qui gère en plus les écritures à un comportement similaire, à ceci prêt qu'il aura éventuellement besoin de se synchroniser avec son stockage s'il réalise des écritures à sa place. Dans ce cas, il aura quand même absorbé des opérations.

En lecture, il synchronise une fois pour absorber après. En écriture, il absorbe avant de synchroniser une fois. Autrement dit, on observe un mécanisme d'absorption, qu'il s'agissent de lectures ou bien d'écritures. C'est un argument supplémentaire qui devrait clore ce débat : les lectures, bien que sans effet de bord, sont des opérations puisqu'elles font exhiber aux caches le même comportement que les autres opérations.

Valuations

Concernant la définition des opérations dans notre modèle, deux questions (liées les unes aux autres) ont tendance à revenir souvent. Elles concernent toutes les trois les paramètres d'entrée et de sortie. 1. Pourquoi le système devrait-il évaluer une partie des paramètres des opérations ? N'était-il pas possible de concevoir un système dans les lequel tous les paramètres des opérations soient fournis en entrée par l'utilisateur ? 2. Pourquoi était-il nécessaire de permettre au système de calculer plusieurs valuations de sorties possible ? Permettre de rendre des opérations non déterministes semble dangereux.

Concernant la première question, notons que les paramètres d'entrée fournis par l'utilisateur sont sensés être suffisant pour calculer les paramètres de sortie à partir de l'état initial. De ce fait, on peut dire que tous les paramètres *nécessaires* sont bien fournis par l'utilisateur,

mais qu'ils ne sont simplement pas *suffisants*.

En fait, ils ne sont *quasiment jamais* suffisant, exceptés dans des cas qui semblent inutiles. Cela vient du système *d'effets*. Afin de rendre le système le plus générique possible, on a définit l'application d'une opération sur un état initial comme l'application d'un effet sur cet état. Pour rappel, les effets sont définis comme des différentiels entre un état initial et un état souhaité, avec une composante négative et une composante positive. Ainsi, chaque élément mis en jeu dans l'effet doit être défini concrètement lors de l'application. Or, l'utilisateur n'est pas sensé connaître complètement l'état du système lorsqu'il envoie ses requêtes. Il est donc nécessaire de compléter les paramètres en entrée par les paramètres que l'utilisateur ne peut pas deviner.

Concernant la seconde question, il est effectivement possible de définir des opérations non déterministes au sein de notre système de données. Cela peut paraître dangereux ou aberrant. On le justifie de deux manières.

La première c'est qu'on peut vouloir définir des systèmes non déterministes. C'est quelque chose qu'on évite de faire généralement en informatique système mais qui peut se rencontrer. Par exemple on peut imaginer qu'une spécification d'un système ne donne pas suffisamment de détails sur le calcul de certains résultats et laisse le soin à l'implémentation de faire certains choix. Si c'est cette spécification qu'on veut étudier, il est nécessaire de rendre compte de ce non déterminisme.

La seconde c'est qu'on peut au contraire vouloir vérifier si un système de données est déterministe ou non. Dans ce cas il est nécessaire d'être en mesure de définir des systèmes qui le soient comme des systèmes qui ne le soient pas.

Caches et proxys

Dans la thèse, il y a un parti pris dont nous n'avons pas parlé plus tôt mais qui est important. Les caches sont des éléments de la topologie, au même titre que les acteurs ou les stockages, et sont en fait des *proxys* des stockages. On les rencontre parfois dans certaines architectures sous le terme de *proxy-cache*. Ce terme est utilisé pour les différencier des systèmes plus classiques où les caches sont intégrés dans des composants de l'architecture, par exemple un stockage, ou un routeur, ou un processeur, etc.

Ici nous en faisons toujours un élément de premier plan avec lequel on communique explicitement (bien qu'on puisse ignorer qu'il s'agisse d'un cache). C'est un choix qui rend plus simple le modèle sans véritablement altérer le comportement du système. Le seul effet de cette modification est d'intégrer des communications explicites, là où il n'y avait que des accès mémoire, et d'ignorer les composants dont il est question et qui n'interviennent pas dans le stockage. C'est donc équivalent du point de vue du résultat. Du point de vue des coûts, il suffira de donner une valeur nulle à ces communications

superflues pour retrouver un coût comparable aux systèmes réels.

Réseaux de Petri

Notre moteur d'exécution est implémenté avec des réseaux de Petri. Nous avons fait ce choix pour deux raisons principales.

La première c'est que le modèle est d'abord destiné à l'étude. S'il doit être possible de l'implémenter, il est plus important encore qu'il permette de réaliser des mesures et de vérifier des résultats. Lorsque nous avons commencé l'implémentation nous voulions un outil permettant de réaliser du *model-checking*, ce qui est la raison d'être des réseaux de Petri. Nous n'avons pas réalisé ce type d'étude dans la thèse. Cela reste cependant une perspective ouverte.

La seconde raison est que nous ne voulions pas avoir à développer un modèle de concurrence, ce qui aurait ajouté une énième couche à notre modèle déjà assez dense et nous aurait probablement restreint en terme d'ordonnancement. Avec les réseaux de Petri, tous les ordonnancements possibles sont accessibles. Il sera toujours temps de les restreindre si on veut étudier l'impact de tel ou tel ordonnancement sur l'exécution. En attendant nous avons pu avancer dans les autres aspects du modèle, qui nous intéressaient plus.

5.2 *Résultats*

Dans cette section nous allons faire le point sur les résultats importants de cette thèse.

Stockages et caches

L'un des points de départ de notre réflexion sur les caches était de considérer ces derniers comme un cas particulier de stockage. En effet on les avait défini de manière informelle comme suit :

> (…) tout objet capable d'adresser des données, de les organiser, de répondre à des requêtes. C'est à dire tout objet fournissant un moyen d'appliquer des traîtements à des données. Sur cette base le cache sera lui-même une forme de stockage ; agira comme un proxy, toujours subordonné à un autre stockage ; et formera avec ce dernier un stockage de fait distribué.

Le fait d'être capable, même de manière informelle comme ci-dessus, de traîter des caches et des stockages de manière équivalente suggérait une unité de concept et justifiait de développer un modèle unifié. C'est ce que nous avons fait au chapitre 3.

Avec le travail présenté dans ce mémoire, nous sommes en mesure d'aller un peu plus loin. On peut maintenant dire qu'il existe au moins un modèle dans lequel caches et stockages sont traîtés de manière équivalente. En fait, ces deux types d'objets sont des nœuds d'une topologie qui se différencient uniquement sur leur politique et leurs attentes quant à l'interprétation de cette topologie.

Un cache nécessite une interprétation hiérarchique, c'est-à-dire que la présence d'un stockage auquel il puisse se référer est nécessaire à son fonctionnement. Ce stockage référent peut par ailleurs être lui-même un cache. Un stockage, à contrario, n'a pas cette attente et se suffit à lui-même. Il s'accomode très bien d'une interprétation hiérarchique mais n'a pas d'attente à ce sujet.

Le pendant du point précédant est que la politique d'un cache pourra se référer au stockage qui lui est attribué pour répondre à un acteur (qui peut d'ailleurs être un cache). Cette relation permettra au cache d'intercepter des requêtes et d'y répondre à la place du stockage. C'est que nous avons apellé l'absorption. Au passage cette absorption pourra générer de l'asynchronisme entre son état et celui du stockage.

Notre modèle nous permet donc de dire trois choses. 1. Un cache est un stockage. 2. Tous les stockages ne sont pas des caches. 3. Un cache et son stockage forment ensemble un stockage distribué.

Généricité

L'une de nos ambitions était que le modèle soit générique, c'est-à-dire qu'il permette de représenter des systèmes de données ainsi que des topologies arbitrairement complexes. Nous n'avons hélas pas eu le temps de valider cette généricité avec des exemples plus complexes. Cependant le modèle intègre deux éléments qui nous confortent sur ce point.

Concernant les données, nous avons noté dans le chapitre 2 que le problème était d'abord de rendre compte des dépendances entre les opérations agissant sur les données. Il était alors nécessaire d'expliciter ces dépendances ; c'était le rôle du gestionnaire d'opérations. Dans le chapitre 3 nous nous sommes contentés d'écrire que pour représenter fidèlement les ressources, il fallait représenter les relations (éventuellement typées) qui existent entre elles. Pour ne pas surcharger le lecteur, nous avons omis de préciser à ce moment deux éléments.

Le premier élément que nous avons omis est que dans les systèmes que nous avons rencontré (systèmes de fichiers, annuaires) les dépendances entre opérations étaient liées à la structure de ces sytèmes. Dans ce cas ce sont des systèmes arborescents, et il existe des relations de parentées entre les ressources. Les opérations agissant sur plusieurs ressources à la fois (et donc générant des dépendances) agissent toujours sur des ressources liées par de telles relations de parentés. Par exemple, dans les systèmes de fichier, l'opération de renommage de fichiers peut modifier les méta-données dudit fichier, de son répertoire parent source et de son répertoire parent destination (qui peut ou non être le même que le répertoire source). Cette opération créé donc inévitablement des dépendances avec les opérations suivantes concernant ces deux (voire trois) ressources. C'est la première raison (la moins forte) qui nous a poussé à expliciter les

relations entre ressources.

Le second élément que nous avons omis est la nécessité pour des politiques préemptives (qui ferait par exemple du *prefetch*) d'avoir connaissance des relations entre ressources afin de fonctionner [1]. Par exemple, dans un cache de mémoire, la politique pourrait détecter un schéma de *streaming* (lecture d'une vidéo par exemple) et décider de lire en avance un certain nombre de blocs afin de mieux répondre à la demande. Cette lecture en avance se fait en suivant l'ordre des adresses dans la mémoire. Implicitement, il existe de ce système une relation d'un bloc au bloc suivant qui permet à la politique de fonctionner. Par extension on peut dire que s'il existe une relation `r` entre deux ressources *a* et *b*, et qu'on détecte un schéma d'accès dans lequel l'existance de cette relation est liée à un accès successif de *a* puis de *b*, on peut étendre la technique de *prefetch* à ce système, en suivant cette fois la relation `r`.

1 Nous n'avons pas développé ce point dans la thèse, cela fait partie des perspectives (voir plus bas).

Ce second élément est une raison très forte pour laquelle il était intéressant d'expliciter les relations entre les ressources du système de données. Cela permet de développer des politiques qui s'intéressent à des classes entières de modèle de données et plus seulement à des cas particuliers. Il suffit pour cela de définir des relations standards que reconnaitront les politiques pour agir de manière informée. Nous gagnons donc ici un premier degré de généricité pour nos politiques grâce à notre modèle.

Nous gagnons un second degré de généricité grâce à la distinction entre topologies et interprétations, discutée au chapitre 2. En effet il devient maintenant possible de décorréler les politiques des formes particulières que peuvent prendre les réseaux.

Dans les cas que nous avons développés au chapitre 4 nous avons choisi une interprétation hiérarchique et l'avons réutilisée dans les deux topologies auxquelles nous avons eu affaire. Dans les cas `ACS` et `ACCS`, la même politique a été réutilisée dans les caches C, C1 et C2. Ainsi la politique n'a pas souffert du changement de topologie, grâce à l'interprétation qui elle s'applique à n'importe quelle topologie.

De même que pour les données, on peut maintenant développer des politiques qui s'adressent à des classes d'interprétation. Par exemple les politiques LRU (cas `ACS` et `ACCS`) et DEMOTE (cas `ADDS`) s'adressent toutes deux à une interprétation hiérarchique. Mais elles s'adressent de mêmes à toute interprétation qui inclus l'interprétation hiérarchique et l'étend ou la fait varier. Par exemple on peut définir une interprétation qui change le comportement de l'interprétation hiérarchique pour gérer plus intelligemment les informations venant de nœuds de même niveau, et donc faire changer le comportement du système dans des cas arborescents et non plus linéaires. Cette nouvelle interprétation resterait tout de même une interprétation hiérarchique, puisqu'elle gérerait les méthodes `integrate` et `relativestorage`. Les politiques présentées plus tôt pourraient donc tout à fait utiliser cette nouvelle version sans subir de modifications.

La généricité de notre modèle se manifeste donc à plusieurs niveaux. Nous sommes en mesure de représenter des systèmes de données arbitrairement complexes. Il n'y a en effet pas de contrainte sur les valeurs que peuvent prendre les ressources, et les relations sont des hypergraphes. Nous sommes de mêmes capables de représenter des topologies arbitrairement complexes. Une topologie est aussi un hypergraphe. Enfin les politiques peuvent s'appuyer sur des relations standards pour les données et sur des intérprétations indépendantes des topologies. Les politiques elles-mêmes gagnent donc en généricité.

Absorption

Dans la thèse nous avons développé la notion d'absorption et l'avons confrontée à celle de succès. Dans le chapitre 4 nous avons clairement montré que le taux d'absorption capturait un effet (en l'occurence une augmentation ou réduction de requêtes transmises par rapport aux requêtes reçues), là où la notion de succès ne capturait qu'une capacité à répondre.

Originellement, ce taux ne se mesurait que par rapport aux requêtes `operate` initiées par l'acteur A. Il mesurait donc strictement une absorption de ces requêtes, dans le sens où toutes n'étaient pas forcément transmises. C'est de là que vient son nom. Ce n'est que plus tard que nous avons décidé d'une définition plus générale qui capture toutes les requêtes.

Ce choix est discutable. D'un coté il permet de mesurer fidèlement l'effet d'un cache dans une chaîne, mais de l'autre nous perdons la notion d'absorption plus spécifique que nous avions précédemment. Nous nous sommes tenus à ce choix car nous voulions capturer le-dit effet.

5.3 *Ouvertures et perspectives*

Au chapitre 1 nous avons listé un certain nombre de problématiques à traîter quand on s'intéresse aux systèmes de caches. Nous n'en avons traîté qu'une partie. Dans cette section nous allons donner quelques pistes pour aller plus loin.

Cohérence

La notion de cohérence est la grande absente du modèle présenté au chapitre 3. Nous ne l'avons pas développée pour deux raisons. 1. Les contraintes de temps de la thèse ne nous auraient pas permis de développer ce point de manière générale. 2. Les cas que nous avons choisis ne posent pas vraiment de soucis sur ce point.

Le modèle présenté au chapitre 3 nous donne tout de même des éléments pour discuter du problème de la cohérence.

Ce problème apparaît lorsque un système comprend plus de deux caches, susceptibles de faire référence aux mêmes données et donc

d'en posséder des versions différentes, auquel cas ils seraient dits incohérents. Selon les cas il peut être résolu de différentes manières (cf. le chapitre 1 pour quelques exemples).

Nous avons déjà établi qu'au regard de notre modèle, les caches et les stockages étaient de même nature. La question de la cohérence doit donc maintenant être posée en général et plus seulement sur les caches. Dans ce cas on observe un phénomène intéressant. Dès que le cache devient asynchrone (il nous permet de réaliser des écritures sur les données sans se référer à son stockage par exemple), cette définition nous indique qu'il existe une incohérence potentielle entre le cache et le stockage. C'est pourtant un effet voulu qui ne semble pas poser de problème. Le terme d'incohérence dans ce cas semble donc trop fort, et il convient d'en établir une nouvelle définition, plus formelle et plus adaptée à notre modèle.

Cette définition (informelle) de la cohérence ressemble à l'une des propriétés que notre modèle permet d'établir sur un état : le caractère bien formé ou non d'un état σ. Il se définit comme le fait qu'il n'existe qu'une seule valeur v pour chaque ressource k de la composante $\sigma.h$ d'une part, et d'autre part que $\mathsf{dom}(\sigma.r) \subseteq \mathsf{dom}(\sigma.h)$ [2].

2 cf. la définition 3 page 36

La notion d'interprétation nous apporte un moyen de faire le lien entre les notions de cohérence et d'état bien formé. En effet elle intègre la méthode globalview qui à partir d'une topologie T et d'un *mapping* $T \to \Sigma_{K,V,L} \times \Sigma^2_{K,V,L}$ renvoie un état $\sigma \in \Sigma_{K,V,L}$. Cet état correspond à une vue globale de l'état distribué. On peut redéfinir la notion de cohérence comme suit : *un état distribué* $T \to \Sigma_{K,V,L} \times \Sigma^2_{K,V,L}$ *est cohérent au regard d'une interprétation* I *si et seulement si* $I.\mathsf{globalview}\,(T \to \Sigma_{K,V,L})$ *est bien formé*.

Cette définition a le mérite d'être complètement générale mais souffre d'un défaut important. Il est en effet nécessaire que la définition de I.globalview soit en cohérence avec la définition de I.integrate qui est appelée par le moteur d'exécution pour intégrer un effet venant d'un nœud i sur un nœud j. Il est peut-être possible de définir globalview comme une fonction de integrate, auquel cas ce problème serait résolu.

Notre nouvelle définition nous permet de résoudre le problème de la cohérence dans les cas traîtés au chapitre 4. Dans l'interprétation utilisée, la méthode globalview projette les effets des couches hautes de la topologie vers les états des couches basses.

La première condition pour que l'état reste cohérent est que l'état initial du système soit déjà cohérent. C'est le cas dans nos exemples, puisque seul l'état du stockage est non vide. La deuxième condition pour que l'état reste cohérent, c'est que les effets des opérations du système ne produisent pas d'état mal formé. C'est aussi le cas pour nos exemples, puisque les lectures n'ont pas d'effet de bord et que les écritures s'assurent de supprimer les références aux ressources qu'ils modifient avant d'ajouter la nouvelle version à l'état.

De cette manière, même s'il existe ponctuellement des ressources

dont les valeurs ne sont pas les mêmes entre les nœuds de la topologie, si les deux conditions précédentes sont respectées il n'existe pas d'incohérence. Ces pour cette raison que nous avons pu laisser de coté cet aspect jusqu'ici.

Si maintenant la topologie devient arborescente au lieu d'être linéaire, il faudra modifier l'interprétation et ajouter une condition. Il faut que l'interprétation intègre les effets d'un nœud i sur un nœud j de même niveau dans la hiérarchie en utilisant la projection comme précédemment. Si le nœud j a subi des modifications qui entrent en conflit avec celles du nœud i, l'état sera incohérent.

La nouvelle condition pour que l'état reste cohérent avec ce système est que les ressources doivent être partitionnées au sein d'un même niveau. Cela nous amene à la limite de ce que permet le modèle. Comment réaliser cette partition ? Est-ce nécessaire si on se dote de moyen de détecter les incohérences au sein d'un niveau et de les gérer à temps ?

Ni le modèle ni le moteur d'exécution présenté ne répondent à ces questions. Il existe cependant des protocoles gérant ces cas. Des travaux supplémentaires devront être réalisés afin d'intégrer ces solutions à nos outils. En attendant, notre modèle nous permet de définir une notion de cohérence plus générale que la définition classique. Cela devient une propriété du système que l'on peut prouver ou infirmer avec notre modèle.

Hiérarchies logiques

Au chapitre 1 nous avons fait référence à trois *problématiques latentes*. Nous avons traîté dans la thèse celle des modèles de données et des topologies complexes ; nous avons laissé de coté celle dite des *hiérarchies sémantiques* (ou *logiques*). Pour rappel cela partait de l'existence dans les systèmes informatiques de couches logiques reposant les unes sur les autres pour proposer des abstractions de plus en plus complexes. Par exemple un système de fichiers reposant sur une mémoire de type blocs (le disque).

Le modèle développé au chapitre 3 nous donne là encore quelques éléments pour traîter cette problématique. Grâce à la section sur les modèles de données[3] nous sommes en mesure d'une part de modéliser formellement les différentes couches du système, et d'autre part de définir une fonction ou un ensemble de fonctions réalisant la traduction de l'une dans l'autre.

3 cf. section 3.1

À partir de là, nous serons en mesure de définir un nouveau système qui serait la juxtaposition de ces couches. À la jonction on pourrait définir une nouvelle politique qui utiliserait ces éléments de traduction. De cette manière on pourrait étudier les effets des changements de l'un de ces sous-systèmes sur les autres. On peut aussi imaginer étudier des politiques distribuées à travers ces différentes parties. C'est une nouvelle piste d'étude qui devient accessible grâce à notre modèle.

Gestionnaire d'opérations

Des travaux ultérieurs devront être réalisés sur le *gestionnaire d'opérations*. Il a été conçu pour maintenir un graphe d'opérations en retenant leurs dépendances réspectives. De cette manière on se rend capables 1. de réaliser des synchronisations sans introduire d'erreur, et 2. d'exhiber de la concurrence au sein des caches. Cependant cette conception souffre de deux limites.

Comme il a été dit précédemment, le modèle nous impose de calculer les paramètres de sortie. Autrement dit, il est possible que le domaine de clés connu des opérations grandisse entre le moment où elle est prise en charge par le gestionnaire d'opérations et le moment où elle est réalisée. Le moteur d'exécution tiens à jour la politique avec ces nouvelles informations mais pas le gestionnaire d'opérations. Cela peut poser des soucis et doit être corrigé. Ce point est mineur, mais un deuxième point est plus gênant.

Imaginons que les paramètres de sortie fassent référence à des ressources déjà existantes. Peut-on s'assurer que nous n'allons pas créer de conflit avec des opérations en cours de traîtement mais non terminées ayant trait à ces ressources ? Nous n'avons pas exploré ce point mais nous pensons que cela peut poser des problèmes de cohérence interne. Ce point devrait être étudié.

Dans nos exemples, nous faisons référence à une version extrêmement simplifiée de système de fichiers. L'opération `create` ajoute une ressource au système. Dans ce cas il n'y a pas de risque de créer d'incohérence interne. Un autre cas, qui n'est pas dans nos exemples, serait une opération de type `readdir`, qui lit le contenu d'un répertoire et renvoie un ensemble de ressoures, sans les modifier. Dans ce cas, vu que rien de ce qui existe ne serait modifié, on peut affirmer qu'il n'y aurait pas d'incohérence interne.

Autrement dit, lors de la conception d'un modèle de données, il faudrait vérifier qu'aucune opération faisant référence à des ressources dans sa valuation de sortie ne modifie ces ressources. C'est un point du modèle qu'il faudrait développer ultérieurement.

La seconde limite dont souffre la conception du gestionnaire d'opérations est liée à la première. Cet objet a été conçu de manière à respecter l'algèbre d'effet. L'idée est que si deux opérations sont indépendantes, leur effet peut être appliqué en parallèle et donc elles ne sont pas liées dans le graphe d'opérations. Le problème vient du fait que l'évaluation du domaine de l'opération lors de son ajout dans le graphe n'est que partielle. On n'est donc pas sûrs que le graphe respecte effectivement l'algèbre d'effet.

De plus, la relation entre l'algèbre d'effet et le graphe d'opérations n'est qu'informelle. Des travaux ultérieurs devront formaliser cette relation pour s'assurer que le gestionnaire d'opérations puisse gérer toutes les dépendances de manière correcte par rapport à cette algèbre.

Ces deux limites sont clairement liées l'une à l'autre. Afin de gérer

des cas plus complexes il conviendra de les traîter toutes les deux.

6
Conclusion

L'objectif principal de la thèse était de fournir un premier modèle de cache générique permettant de séparer la politique des structures de données et des topologies. Ces contraintes nous ont poussé à développer le modèle présenté au chapitre 3.

Nous avons implémenté ce modèle sous la forme d'un simulateur nous permettant de faire évoluer les éléments du modèle indépendamment les uns des autres. Cela vient valider que le modèle répond bien aux contraintes que nous avions définies.

Cela ne représente que la première étape du travail. L'étape suivante consistera à étudier de plus prêt la manière dont on pourrait faire évoluer les politiques du modèle pour y intégrer les stratégies présentées en introduction. Nous pensons que le fait d'avoir séparé les données et les topologies des politiques aide à aller dans cette direction.

Annexes

A *Rappels sur les réseaux de Petri* 87

B *Implémentation* 93

B.1 *Moteur d'exécution* 93

A
Rappels sur les réseaux de Petri

Nous avons utilisé les réseaux de Petri afin d'implémenter notre modèle. Ainsi, bien que ce document ne porte pas sur les réseaux de Petri, nous allons maintenant en présenter une définition tirée directement de [33]. Nous l'étendrons ensuite légèrement pour y intégrer de la modularité.

[33] Pommereau: "Algebras of coloured Petri nets", 2009.

Définitions préliminaires

Nous utiliserons la *notation lambda* pour noter certaines fonctions. Par exemple $\lambda x : x > 0$ désigne la fonction qui prend un argument x et retourne la valeur booléenne $x > 0$. De manière similaire, la fonction $\lambda x, y : x > y$ calcule si x est plus grand que y. Si f et g sont deux fonctions aux domaines disjoints F et G, on définit $f \cup g$ comme la fonction de domaine $F \cup G$ telle que $(f \cup g)|_F = f$ et $(f \cup g)|_G = g$.

Un *multi-ensemble* m sur un domaine D est une fonction $m : D \to \mathbb{N}$ (entiers naturels), où, pour $d \in D$, $m(d)$ désigne le nombre d'occurences de d dans le multi-ensemble m. Le multi-ensemble vide est noté $\emptyset$ et est la fonction constante $\emptyset \stackrel{\text{df}}{=} (\lambda x : 0)$. Nous noterons les multi-ensembles comme des ensembles avec des repétitions, par exemple $m_1 \stackrel{\text{df}}{=} \{1, 1, 2, 3\}$ est un multi-ensemble ainsi que $\{d + 1 \mid d \in m_1\}$. Ce dernier, donné en intention, est équivalent à $\{2, 2, 3, 4\}$. Un multi-ensemble m sur D peut être naturellement étendu à n'importe quel domaine $D' \supset D$ en définissant $m(d) \stackrel{\text{df}}{=} 0$ pour tout $d \in D' \setminus D$. Si m_1 et m_2 sont deux multi-ensembles sur le même domaine D, on définit :

— $m_1 \leq m_2$ ssi $m_1(d) \leq m_2(d)$ pour tout $d \in D$;
— $m_1 + m_2$ est le multi-ensemble sur D définit par $(m_1 + m_2)(d) \stackrel{\text{df}}{=} m_1(d) + m_2(d)$ pour tout $d \in D$;
— $m_1 - m_2$ est le multi-ensemble sur D définit par $(m_1 - m_2)(d) \stackrel{\text{df}}{=} \max(0, m_1(d) - m_2(d))$ pour tout $d \in D$;
— pour $d \in D$, on désigne par $d \in m_1$ le fait que $m_1(d) > 0$.

Réseaux de Petri

Un réseau de Petri (coloré) met en jeu des valeurs, des variables et des expressions. Ces objets sont définis par un *domaine de couleur* qui fournit des valeurs pour les données, des variables, des opérateurs,

une syntaxe pour les expressions, éventuellement des règles de typage, etc. Par exemple, on peut utiliser l'arithmétique des entiers ou la logique booléenne comme domaine de couleur. Généralement des domaines de couleur plus élaborés sont utiles pour faciliter la modélisation. En particulier on peut considérer un langage de programmation fonctionnel ou la partie fonctionnelle (expressions) d'un langage impératif. On considère un domaine de couleur abstrait avec les éléments suivants :

— $\mathbb{D}$ est l'ensemble des valeurs des *données* ; il peut inclure en particulier le $\bullet$ des réseaux de Petri, des valeurs entières, les valeurs booléennes True et False, et une valeur "indéfinie" spéciale $\perp$;

— $\mathbb{V}$ est l'ensemble des *variables*, généralement désignées par les lettres x, y, ..., ou par des lettres indicées comme x_1, y_k, ... ;

— $\mathbb{E}$ est l'ensemble des *expressions*, impliquant des valeurs, des variables et des opérateurs appropriés. Soit $e \in \mathbb{E}$, on désigne par $\mathsf{vars}(e)$ l'ensemble des variables de $\mathbb{V}$ impliquées dans e. De plus, les variables et les valeurs peuvent être considérées comme des expressions simples, *i.e.*, on suppose que $\mathbb{D} \cup \mathbb{V} \subset \mathbb{E}$.

On ne fait aucune hypothèse sur le typage ou la correction syntaxique des valeurs et des expressions ; à la place, on suppose que toute expression peut être évaluée, éventuellement à $\perp$ (indéfinie). Plus précisément, une *valuation* est une fonction partielle $\beta : \mathbb{V} \to \mathbb{D}$. Soit $e \in \mathbb{E}$ et β une valuation, on désigne par $\beta(e)$ l'évaluation de e sous β ; si le domaine de β n'inclue pas $\mathsf{vars}(e)$ alors $\beta(e) \stackrel{\text{df}}{=} \perp$. L'application d'une valuation pour évaluer une expression est naturellement étendue aux ensembles et multi-ensembles d'expressions.

Par exemple, si $\beta \stackrel{\text{df}}{=} \{x \mapsto 1, y \mapsto 2\}$, on a $\beta(x + y) = 3$. Avec $\beta \stackrel{\text{df}}{=} \{x \mapsto 1, y \mapsto \texttt{"2"}\}$, selon le domaine de couleur, on peut avoir $\beta(x + y) = \perp$ (pas de coercition), ou $\beta(x + y) = \texttt{"12"}$ (coercition de l'entier 1 sur le caractère `"1"`), ou $\beta(x + y) = 3$ (coercition du caractère `"2"` sur l'entier 2), ou même d'autres valeurs définies par le domaine de couleur concrêt.

DÉFINITION 24. (Réseau de Petri) *Un* réseau de Petri *est un tuple* (S, T, ℓ) *tel que :*

— S *est un ensemble fini de* places ;

— T, *disjoint de* S, *est un ensemble fini de* transitions ;

— ℓ *est une fonction d'*étiquetage *telle que :*

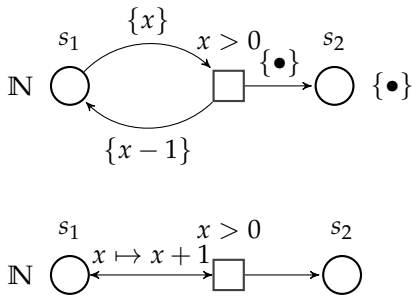
  — *pour tout* $s \in S$, $\ell(s) \subseteq \mathbb{D}$ *est le* type *de* s, i.e., *l'ensemble des valeurs que* s *peut prendre,*

  — *pour tout* $t \in T$, $\ell(t) \in \mathbb{E}$ *est la* garde *de* t, i.e., *une condition pour son exécution,*

  — *pour tout* $(x, y) \in (S \times T) \cup (T \times S)$, $\ell(x, y)$ *est un multi-ensemble sur* $\mathbb{E}$ *et définit l'*arc *de* x *vers* y.

Comme il est d'usage, les réseaux de Petri sont représentés comme des graphes dans lesquels les places sont des nœuds ronds, les transitions sont des nœuds carrés, et les arcs sont dirigés. La figure A.1

présente un réseau de Petri représenté en notations à la fois graphique et textuelle. Les arcs vides, *i.e.*, les arcs tels que $\ell(x, y) = \emptyset$, ne sont pas représentés. De plus, pour alléger les dessins, on omettra quelques annotations (voir figure A.1) : $\{\bullet\}$ pour les types de place ou annotations d'arcs, les accolades $\{\ \}$ autour des multi-ensembles d'expression sur les arcs, les gardes True, et les noms de nœud qui ne sont pas nécessaires aux explications. Enfin, deux arcs opposés peuvent être représentés par un arcs simple à deux flèches, ce qui est le cas entre s_1 et t dans la partie basse de la figure A.1.



$$
\begin{array}{rcl}
S & \stackrel{\text{df}}{=} & \{s_1, s_2\} \\
T & \stackrel{\text{df}}{=} & \{t\} \\
\ell & \stackrel{\text{df}}{=} & \{s_1 \mapsto \mathbb{N}, s_2 \mapsto \{\bullet\}, \\
 & & \ t \mapsto x > 0, \\
 & & \ (s_1, t) \mapsto \{x\}, \\
 & & \ (s_2, t) \mapsto \emptyset, \\
 & & \ (t, s_1) \mapsto \{x-1\}, \\
 & & \ (t, s_2) \mapsto \{\bullet\}\}
\end{array}
$$

FIGURE A.1: Un réseau de petri simple, avec annotations complètes (en haut à gauche) et simplifiées (en bas à gauche). À droite, la forme textuelle.

Deux réseaux de Petri (S_1, T_1, ℓ_1) et (S_2, T_2, ℓ_2) sont *disjoints* ssi $S_1 \cap S_2 = T_1 \cap T_2 = \emptyset$.

DÉFINITION 25. (Marquages et sémantique séquentielle) *Soit* $N \stackrel{\text{df}}{=} (S, T, \ell)$ *un réseau de Petri.*

Un marquage *M de N est une fonction sur S qui associe à chaque place s un multi-ensemble fini sur* $\ell(s)$ *représentant les jetons contenus dans s.*

Une transition $t \in T$ *est* activée *pour un marquage M et une valuation* β*, ce qu'on note* $M[t, \beta\rangle$*, ssi les conditions suivantes sont respectées :*

— *M a suffisamment de jetons,* i.e., *pour tout* $s \in S$, $\beta(\ell(s, t)) \leq M(s)$ *;*

— *la garde est satisfaite,* i.e., $\beta(\ell(t)) = \mathsf{True}$ *;*

— *les types des places sont respectés,* i.e., *pour tout* $s \in S$, $\beta(\ell(t, s))$ *est un multi-ensemble sur* $\ell(s)$*.*

Si $t \in T$ *est activée pour le marquage M et la valuation* β*, alors t peut être* tirée *et produire un marquage* M' *défini pour tout* $s \in S$ *comme* $M'(s) \stackrel{\text{df}}{=} M(s) - \beta(\ell(s, t)) + \beta(\ell(t, s))$*. Ceci est noté* $M[t, \beta\rangle M'$*.*

Le graphe de marquage *G d'un réseau de Petri marqué avec M est le plus petit graphe étiqueté tel que :*

— *M est un nœud de G ;*

— *si* M' *est un nœud de G et* $M'[t, \beta\rangle M''$ *alors* M'' *est aussi un nœud de G et il y a un arc dans G de* M' *vers* M'' *étiqueté par* (t, β)*.*

Notons que cette définition des graphes de marquages permet d'ajouter une infinité d'arcs entre deux marquages. En effet si $M[t, \beta\rangle$ il peut exister une infinité d'autres valuations activantes qui diffèrent de β seulement sur les variables non impliquées dans t. Ainsi, on ne considère que les tirages $M[t, \beta\rangle$ tels que le domaine de β soit $\mathsf{vars}(t) \stackrel{\text{df}}{=} \mathsf{vars}(\ell(t)) \cup \bigcup_{s \in S}(\mathsf{vars}(\ell(s, t)) \cup \mathsf{vars}(\ell(t, s)))$.

Par exemple, considérons encore le réseau de Petri de la figure A.1 et supposons qu'il soit marqué par $M_0 \stackrel{\text{df}}{=} \{s_0 \mapsto \{2\}, s_2 \mapsto \emptyset\}$, son graphe de marquage a trois nœuds comme représenté dans la figure A.2. Remarquons qu'à partir du marquage M_2, aucune valuation ne peut activer t car, soit $x \not\mapsto 0$ et alors M_2 n'a pas assez de jetons, ou $x \mapsto 0$ et alors à la fois la garde $x > 0$ n'est pas satisfaite et le type de s_1 n'est pas respecté ($x - 1$ s'évalue à -1).

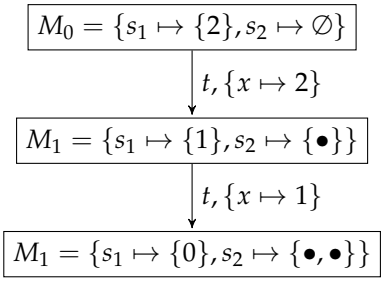


FIGURE A.2: Le graphe de marquage du réseau de Petri de la figure A.1.

Réseaux de Petri modulaires

Nous ajoutons maintenant une notion de modularité en permettant la fusion de réseaux de Petri contenant éventuellement un ensemble de places communes. Cela nous permettra de définir des petits réseaux ayant chacun une fonctionnalité bien définie et de les composer ensuite.

DÉFINITION 26. *Soit $\{(N_i, M_i)\}_{1 \leq i \leq n}$ une famille de réseaux de Petri marqués, avec $N_i = (S_i, T_i, l_i), \forall i \in [1, n]$ telle que :*

$$i \neq j \implies \begin{cases} T_i \cap T_j = \emptyset \\ s \in S_i \cap S_j \implies l_i(s) = l_j(s) \end{cases}$$

C'est à dire tel qu'aucun des réseaux n'a de transition commune et que si une place est commune à plusieurs réseaux, cette place à le même étiquetage dans tous ces réseaux.

On note $\langle \{(N_i, M_i)\}_{1 \leq i \leq n} \rangle$ la fusion des réseaux de cette famille :
$\langle \{(N_i, M_i)\}_{1 \leq i \leq n} \rangle \stackrel{\text{df}}{=} (S, T, l), M$ *tel que :*

— $S \stackrel{\text{df}}{=} \bigcup_{1 \leq i \leq n} S_i$
— $T \stackrel{\text{df}}{=} \bigcup_{1 \leq i \leq n} T_i$
— $t \in T_i \implies l(t) = l_i(t)$
— $s \in S_i \implies l(s) = l_i(s)$
— $M(S) = \bigcup_{1 \leq i \leq n \wedge s \in S_i} M_i(s)$

Exemple La figure A.3 représente la fusion de deux réseaux de Petri très simples. On ne représente ici que les réseaux et pas leurs marquages.

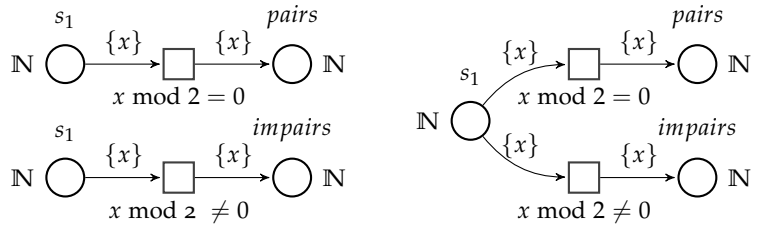


FIGURE A.3: À gauche, deux réseaux partageant une place s_1. Le premier place les nombres entiers de s_1 dans *pairs* s'ils sont pairs ; le second les place dans *impairs* s'ils sont impairs. À droite, la fusion de ces deux réseaux réalise un classement des entiers de s_1 en nombres pairs et impairs.

B
Implémentation

B.1 Moteur d'exécution

Notations

Notations générales Les places partagées sont représentées par des cercles gras et ont un nom. Les places non partagées sont dessinées avec des cercles fins et sont anonymes.

Pour des raisons de lisibilité, les annotations d'arcs ainsi que les gardes des transitions peuvent être écrites sous les réseaux et non à l'intérieur. De plus on se permet de factoriser certains arcs comme suit : les branchements ($\prec$) et les jointures ($\succ$) transportent les mêmes jetons sur chacune de leurs branches respectives. Cela évite de répeter trop de texte.

Pour les mêmes raisons que précédemment, on ne dessine pas toutes les places mises en jeu. À la place on introduit quelques notations. Par exemple, les politiques sont stockées dans une place partagée *Policies* qui contient des couples (me, P_{me}) pour chaque nœud me. Un arc test sur *Policies* pour utiliser P_{me} est remplacé par la notation suivante : $resultat \leftarrow calculAvec(P_{me})$. Si un nouveau P_{me} est calculé et supposé remplacer l'ancien, on le note comme suit : $P_{me}, resultats \leftarrow calculAvec(P_{me})$. On prend les mêmes conventions avec $jobs_{me}$, I_T et (σ_{me}, e_{me}).

Notations pour les communications Les communications au sein de la topologie sont simulées grâce à deux places partagées utilisées conjointement : *io* et *buses*. Chaque message envoyé est placé dans la place *io*, où il est supposé être pris par son nœud de destination. Lorsqu'un message est envoyé sur un bus b, le jeton b est pris par l'émetteur dans la place *buses* ; il est remis à sa place par le récepteur du message. De cette manière seul un message peut transiter à chaque instant sur un bus donné.

Pour des raisons de lisibilité, on introduit une notation particulière pour les communications. La figure B.1 montre les deux manières équivalentes de représenter ceci.

Notations pour les handlers Dans le modèle, les *handlers* pour les nouvelles requêtes sont crées par le gestionnaire de requêtes. Dans notre

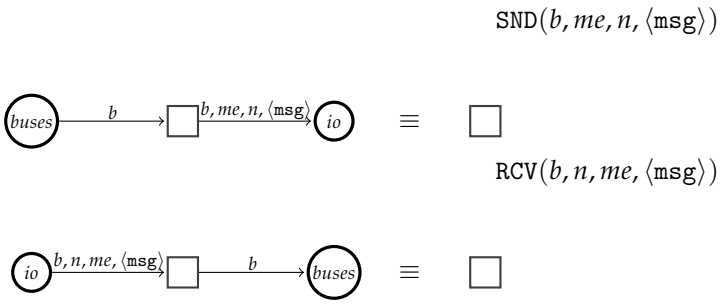


FIGURE B.1: Notations pour les communications

implémentation, nous utilisons une fonctionnalité de SNAKES : les *pids*. Cette fonctionnalité a initialement été créée pour gérer les processus dans les réseaux de Petri colorés [34]. Nous utilisons les notations décrites dans [35]: p étant une structure *Pid*, $\nu(p)$ créé une nouvelle structure *Pid* qui a une relation parent-enfant avec p. $\chi(p)$ détruit le *Pid* p.

[34] FRONC: "Effective Marking Equivalence Checking in Systems with Dynamic Process Creation", 2012.

[35] KLAUDEL et al.: "State space reduction for dynamic process creation", 2010.

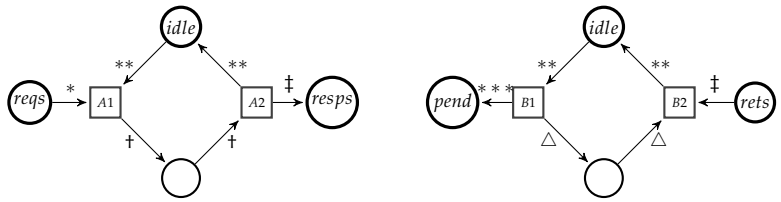
Implémentation des communications

Les procédures de communications (bloquantes, comme non bloquantes) sont implémentées de manières à être remplaçables sans changer la logique du moteur d'exécution. À cette fin on leur donne comme interface quatre places partagées : 1. *requests*, 2. *responses*, 3. *pendings*, et 4. *returns*.

requests est le point d'entrée des envois. Elle reçoit la requête à envoyer, le destinataire et le bus sur lequel transmettre. *responses* est le pendant de *requests*. Les réponses aux requêtes y sont placées.

pendings et *returns* sont les pendants de *requests* et *responses*. Lorsqu'une requête est reçue, elle est placée dans *pendings*. Lorsqu'une réponse est prête à être renvoyée, elle est placée dans *returns*.

Communications bloquantes La place *idle* est utilisée par le nœud *me* pour interdire toute communication avec le nœud n tant qu'une réponse n'a pas été reçue. Elle est initialisée avec : $\{(me, n) | me, n \in T \wedge T[me, n]\}$.

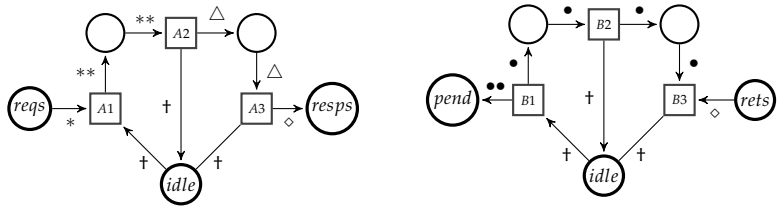


Légende :

$*$	me, req, h, n, b	†	me, h
$**$	me, n	‡	$me, resp, h$
$***$	me, req, h	△	me, n, h

$B1$	$\texttt{RCV}(b, n, me, (\texttt{block}, req))$	$B2$	$\texttt{SND}(b, me, n, (\texttt{return}, resp))$
	$h = \nu(me)$		$\chi(h)$
	$jobs_{me} \leftarrow jobs_{me}.add(req, h)$	$A1$	$\texttt{SND}(b, me, n, (\texttt{block}, req))$
	$P^{me}_{I_T} \leftarrow P^{me}_{I_T}.update(keys(req.\beta), h, req)$	$A2$	$\texttt{RCV}(b, n, me, (\texttt{return}, resp))$

Communications non bloquantes La place *idle* est partagée avec les réseaux des communications bloquantes.



Légende :

$*$	me, req, h, n, b	$\ddagger$	me, n, h
$**$	me, h	$\triangle$	me, n, h, h'
$\bullet\bullet$	me, req, h	$\diamond$	$me, resp, h$
†	me, n	$\bullet$	me, n, b, h

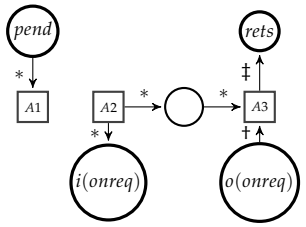
$A1$	$\texttt{SND}(b, me, n, (\texttt{nblock}, req))$	$B1$	$\texttt{RCV}(b, n, me, (\texttt{nblock}, req))$
$A2$	$\texttt{RCV}(b, n, me, (\texttt{wait}, h))$		$h = \nu(me)$
$A3$	$\texttt{RCV}(b, n, me, (\texttt{return}, h', resp))$		$jobs_{me} \leftarrow jobs_{me}.add(req, h)$
$B3$	$\texttt{SND}(b, me, n, (\texttt{return}, h, resp))$		$P^{me}_{I_T} \leftarrow P^{me}_{I_T}.update(keys(req.\beta), h, req)$
	$\chi(h)$	$B2$	$\texttt{SND}(b, me, n, (\texttt{wait}, h))$

Implémentation du moteur

Les processus *listener* et *dispatcher* ne sont pas implémentés comme tels. Ils émergent à partir de l'implémentation des communications et du réseau présenté ci-après.

Prise en charge des requêtes Comme décrit plus haut, les requêtes reçues sont placées dans la place *pendings*. Le gestionnaire de requêtes doit maintenant les prendre en charge.

Une fois ceci effectué, le moteur d'exécution peut demander un nouveau travail au gestionnaire de requête et faire appel à la politique afin de sélectionner le module adéquat. Enfin le moteur attend que le-dit module ait déterminé une réponse afin de la renvoyer dans *returns*.



Légende :

$*$	me, req, h	$\ddagger$	$me, resp, h$
†	$me, resp, h, n$		

$A1$	$jobs_{me} \leftarrow jobs_{me}.add(req, h)$	$A3$	$P^{me}_{I_T} \leftarrow P^{me}_{I_T}.close(h, resp[0])$
	$P^{me}_{I_T} \leftarrow P^{me}_{I_T}.update(keys(req.\beta), h, req)$		$jobs_{me} \leftarrow jobs_{me}.done(h)$
$A2$	$jobs_{me}, req, h \leftarrow jobs_{me}.next()$		
	$req \neq$ ✘		

Nous en avons terminé pour les processus *listener* et *dispatcher*.

Bibliography

[1] T. Kilburn et al. "One-Level Storage System". In: *Electronic Computers, IRE Transactions on* EC-11.2 (Apr. 1962), pp. 223–235. ISSN: 0367-9950. DOI: 10.1109/TEC.1962.5219356.

[2] John T. Robinson and Murthy V. Devarakonda. "Data cache management using frequency-based replacement". In: *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*. SIGMETRICS '90. Univ. of Colorado, Boulder, Colorado, United States: ACM, 1990, pp. 134–142. ISBN: 0-89791-359-0. DOI: http://doi.acm.org/10.1145/98457.98523. URL: http://doi.acm.org/10.1145/98457.98523.

[3] Binny S. Gill et al. "STOW : a spatially and temporally optimized write caching algorithm". In: *Proceedings of the 2009 conference on USENIX Annual technical conference*. USENIX'09. San Diego, California: USENIX Association, 2009, pp. 26–26. URL: http://dl.acm.org/citation.cfm?id=1855807.1855833.

[4] Jong Min Kim et al. "A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references". In: *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*. OSDI'00. San Diego, California: USENIX Association, 2000, pp. 9–9. URL: http://dl.acm.org/citation.cfm?id=1251229.1251238.

[5] Binny S. Gill and Luis Angel D. Bathen. "AMP : adaptive multi-stream prefetching in a shared cache". In: *Proceedings of the 5th USENIX conference on File and Storage Technologies*. San Jose, CA: USENIX Association, 2007, pp. 26–26. URL: http://dl.acm.org/citation.cfm?id=1267903.1267929.

[6] Binny S. Gill and Dharmendra S. Modha. "WOW : wise ordering for writes - combining spatial and temporal locality in non-volatile caches". In: *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*. San Francisco, CA: USENIX Association, 2005, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=1251028.1251038.

[7] Prabuddha Biswas, K. K. Ramakrishnan, and Don Towsley. "Trace driven analysis of write caching policies for disks". In: *Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*. SIGMETRICS '93. Santa Clara, California, United States: ACM, 1993, pp. 13–23. ISBN: 0-89791-580-1. DOI: http://doi.acm.org/10.1145/166955.166971. URL: http://doi.acm.org/10.1145/166955.166971.

[8] Ramakrishna Karedla, J. Spencer Love, and Bradley G. Wherry. "Caching strategies to improve disk system performance". In: *Computer* 27 (3 Mar. 1994), pp. 38–46. ISSN: 0018-9162. DOI: 10.1109/2.268884. URL: http://dl.acm.org/citation.cfm?id=176756.176763.

[9] Zhifeng Chen et al. "Empirical evaluation of multi-level buffer cache collaboration for storage systems". In: *SIGMETRICS'05*. ACM, 2005.

[10] Binny S. Gill and Dharmendra S. Modha. "SARC : sequential prefetching in adaptive replacement cache". In: *Proceedings of the annual conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, pp. 33–33. URL: http://dl.acm.org/citation.cfm?id=1247360.1247393.

[11] Theodore M. Wong and John Wilkes. "My Cache or Yours ? Making Storage More Exclusive". In: *FAST'02*. USENIX Association, 2002.

[12] Binny S. Gill. "On multi-level exclusive caching : offline optimality and why promotions are better than demotions". In: *FAST'08*. USENIX Association, 2008.

[13] Edward G. Coffman Jr. and Peter J. Denning. *Operating Systems Theory*. Prentice Hall Professional Technical Reference, 1973. ISBN: 0136378684.

[14] Alfred V. Aho, Peter J. Denning, and Jeffrey D. Ullman. "Principles of Optimal Page Replacement". In: *J. ACM* 18 (1 Jan. 1971), pp. 80–93. ISSN: 0004-5411. DOI: http://doi.acm.org/10.1145/321623.321632. URL: http://doi.acm.org/10.1145/321623.321632.

[15] Donghee Lee et al. "On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies". In: *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. SIGMETRICS '99. Atlanta, Georgia, United States: ACM, 1999, pp. 134–143. ISBN: 1-58113-083-X. DOI: http://doi.acm.org/10.1145/301453.301487. URL: http://doi.acm.org/10.1145/301453.301487.

[16] L. A. Belady. "A study of replacement algorithms for a virtual-storage computer". In: *IBM Syst. J.* 5 (2 1966).

[17] D. Lee et al. "LRFU : A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies". In: *IEEE Trans. Comput.* 50 (12 Dec. 2001), pp. 1352–1361. ISSN: 0018-9340. DOI: http://dx.doi.org/10.1109/TC.2001.970573. URL: http://dx.doi.org/10.1109/TC.2001.970573.

[18] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. "EELRU : simple and effective adaptive page replacement". In: *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. SIGMETRICS '99. Atlanta, Georgia, United States: ACM, 1999, pp. 122–133. ISBN: 1-58113-083-X. DOI: http://doi.acm.org/10.1145/301453.301486. URL: http://doi.acm.org/10.1145/301453.301486.

[19] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. "The LRU-K page replacement algorithm for database disk buffering". In: *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*. SIGMOD '93. Washington, D.C., United States: ACM, 1993, pp. 297–306. ISBN: 0-89791-592-5. DOI: http://doi.acm.org/10.1145/170035.170081. URL: http://doi.acm.org/10.1145/170035.170081.

[20] Song Jiang et al. "DULO : an effective buffer cache management scheme to exploit both temporal and spatial locality". In: *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*. San Francisco, CA: USENIX Association, 2005, pp. 8–8. URL: http://dl.acm.org/citation.cfm?id=1251028.1251036.

[21] Song Jiang and Xiaodong Zhang. "LIRS : an efficient low inter-reference recency set replacement policy to improve buffer cache performance". In: *SIGMETRICS Perform. Eval. Rev.* 30 (1 June 2002), pp. 31–42. ISSN: 0163-5999. DOI: http://doi.acm.org/10.1145/511399.511340. URL: http://doi.acm.org/10.1145/511399.511340.

[22] Nimrod Megiddo and Dharmendra S. Modha. "ARC : A Self-Tuning, Low Overhead Replacement Cache". In: *FAST'03*. USENIX Association, 2003.

[23] R. H. Patterson et al. "Informed prefetching and caching". In: *SIGOPS Oper. Syst. Rev.* 29 (5 Dec. 1995), pp. 79–95. ISSN: 0163-5980. DOI: http://doi.acm.org/10.1145/224057.224064. URL: http://doi.acm.org/10.1145/224057.224064.

[24] Xuhui Li et al. "Second-tier cache management using write hints". In: *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*. San Francisco, CA: USENIX Association, 2005, pp. 9–9. URL: http://dl.acm.org/citation.cfm?id=1251028.1251037.

[25] Xin Liu et al. "CLIC : client-informed caching for storage servers". In: *Proccedings of the 7th conference on File and storage technologies*. San Francisco, California: USENIX Association, 2009, pp. 297–310. URL: http://dl.acm.org/citation.cfm?id=1525908.1525930.

[26] Chentao Wu et al. "Hint-K : An Efficient Multi-level Cache Using K-Step Hints". In: *Proceedings of the 2010 39th International Conference on Parallel Processing*. ICPP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 624–633. ISBN: 978-0-7695-4156-3. DOI: http://dx.doi.org/10.1109/ICPP.2010.70. URL: http://dx.doi.org/10.1109/ICPP.2010.70.

[27] Alexandros Batsakis et al. "AWOL : an adaptive write optimizations layer". In: *Proceedings of the 6th USENIX Conference on File and Storage Technologies*. FAST'08. San Jose, California: USENIX Association, 2008, 5 :1–5 :14. URL: http://dl.acm.org/citation.cfm?id=1364813.1364818.

[28] D. Muntz and P. Honeyman. "Multi-level Caching in Distributed File Systems -or- Your cache ain't nuthin' but trash". In: *Winter 1992 USENIX*. 1992.

[29] Zhifeng Chen, Yuanyuan Zhou, and Kai Li. "Eviction based cache placement for storage caches". In: *in USENIX Annual Technical Conference. Usenix*. 2003.

[30] Gala Yadgar, Michael Factor, and Assaf Schuster. "Karma : know-it-all replacement for a multilevel cache". In: *Proceedings of the 5th USENIX conference on File and Storage Technologies*. San Jose, CA: USENIX Association, 2007, pp. 25–25. URL: http://dl.acm.org/citation.cfm?id=1267903.1267928.

[31] Zhenmin Li et al. "C-Miner : Mining Block Correlations in Storage Systems". In: *FAST'04*. USENIX Association, 2004.

[32] Franck Pommereau. "Quickly prototyping Petri nets tools with SNAKES". In: *Petri net newsletter* 10-2008 (Oct. 2008). The URL given in the abstract is outdated, now, http://www.ibisc.univ-evry.fr/~fpommereau/SNAKES SNAKES is available here., pp. 1–18.

[33] Franck Pommereau. "Algebras of coloured Petri nets. and their applications to modelling and verification". Habilitation thesis. University Paris-East, Créteil, Nov. 2009.

[34] Łukasz Fronc. "Effective Marking Equivalence Checking in Systems with Dynamic Process Creation". In: *INFINITY'12*. ENTCS. Elsevier, 2012.

[35] Hanna Klaudel et al. "State space reduction for dynamic process creation". In: *Scientific Annals of Computer Science* 20 (2010), pp. 131–157.