UNIVERSITÉ LOUIS PASTEUR STRASBOURG

LSiiT

Mémoire de DEA d'informatique

ESODYP : un mécanisme dynamique de préchargement
de données entièrement logiciel basé
sur une modélisation de Markov des pas en mémoire

Jean Christophe Beyler
sous la direction du Professeur Philippe Clauss

ICPS/LSIIT Université Louis-Pasteur de Strasbourg

2003/2004

Table des matières

Table des figures

Liste des tableaux

Liste des Algorithmes

Remerciements

Le travail réalisé pendant ce stage a nécessité l'aide de plusieurs personnes. Je tiens donc à les remercier toutes pour leur soutien et leur contribution.

Je remercie tout d'abord ma famille pour le soutien qu'elle m'a apporté jour après jour, non seulement pendant ce stage mais aussi durant ces 6 dernières années.

A Técla, qui a subi mes humeurs, mes craintes et mes incertitudes (et il y en a eu !) et qui a réussi à me faire taire pendant les cours et travailler les soirs où *rien faire* semblait tellement mieux. C'est grâce à elle que je suis arrivé aussi loin. Je lui dis merci pour tout et encore plus.

A Philippe Clauss, le professeur qui a cru qu'une optimisation dynamique était réaliste malgré les premiers résultats ! Il m'a énormément aidé, autant pour la direction que devaient prendre les recherches que par les conseils donnés au cours de cette étude. Si j'ai réussi quoi que ce soit pendant ce stage, nous l'avons fait ensemble.

Finalement, j'aimerais remercier toute l'équipe de l'ICPS pour leur accueil et leur aide. Mais particulièrement Romaric David qui a installé mon ordinateur plusieurs fois et a pris la peine de rendre mon xterm bruyant ; Stéphane Genaud, Arnaud Giersch et Guillaume Latu pour leur aide quotidienne et leur bonne humeur ; Vincent Loechner qui a su ramener le soleil lorsque la tempête faisait rage et Benoît Meister qui m'a gracieusement inspiré et nourri (pas de jaloux la liste est alphabétique !).

Merci encore à tous.

Introduction

Exécuter un programme dédié au contrôle du comportement d'un autre programme s'exécutant simultanément est une idée qui peut sembler irréaliste, particulièrement lorsqu'il s'agit d'améliorer le temps d'exécution. L'objectif premier de ce travail de DEA est de montrer qu'au contraire, des améliorations significatives peuvent être obtenues grâce à un tel mécanisme.

Il est bien connu que, malgré la présence de caches, les accès mémoire provoquent un phénomène de goulot d'étranglement. Les stratégies de chargements et de remplacements des données dans les caches ne sont malheureusement pas adaptées au comportement mémoire de tous les types de programmes. Un compilateur peut aider le processeur en générant des instructions de conseil à partir d'une analyse statique du code source [8] lorsque de tels conseils sont présents dans le jeu d'instructions. Mais, une telle approche n'est réalisable que pour des structures de contrôles et de données statiques, comme dans le cas du parcours d'un tableau multi-dimensionnel par une boucle utilisant des références aux éléments ayant la forme de fonctions affines. Lorsque nous considérons des cas plus généraux, les optimisations statiques, c'est à dire sur le code source, ne peuvent généralement pas s'appliquer puisque certains paramètres ne seront connus qu'au moment de l'exécution. C'est pour cette raison que seule une approche dynamique peut être utilisée.

L'approche dynamique consiste à développer un processus matériel et/ou logiciel qui, parallèlement à l'application que l'on désire optimiser, s'exécute et intervient avantageusement sur son comportement. Le principal défi d'une telle approche est de compenser le plus largement possible le coût de l'exécution de cet optimiseur par le gain qu'il engendre.

Parmi les recherches en analyses et optimisations dynamiques, certains travaux utilisent des spécificités de l'architecture [6, 23], d'autres utilisent une première exécution ou l'aide du compilateur pour configurer l'optimiseur. Pour Kim *et al.* [20] et Luk [24], la détection des défauts de caches est faite au préalable afin de créer des threads qui préchargeront les données sur un processeur hyperthreadé [15].

Lorsque nous voulons construire un système entièrement dynamique et portable [7, 13], il faut faire attention aux différents coûts et l'optimiseur doit rapidement repérer les zones qu'il peut optimiser afin de se rendre rentable. Ces systèmes ont l'avantage d'être transparents pour l'utilisateur et n'ont pas besoin d'une première exécution.

Des solutions entièrement matérielles [5, 6, 11, 19, 22, 26] existent et ont généralement un coût moindre en temps par rapport aux solutions logicielles. Mais ils n'ont pas la possibilité de faire des optimisations évoluées et ne sont pas portables. Des systèmes hybrides fournissent un compromis mais sont encore liés à une architecture précise [20, 21, 23, 24]. Enfin, certains utilisent des outils non portables mais obtiennent des bons résultats sur la plateforme ciblée. Par exemple, l'optimiseur présenté dans [10] utilise l'outil *VULCAN* [27] qui n'est disponible que sur les systèmes *Windows*.

La plupart des solutions matérielles [19, 22] ne s'intéressent qu'aux adresses qui provoquent des défauts de cache. Pour y parvenir, elles dépendent d'un cache dédié qui est ajouté au processeur et utilisé pour les calculs des préchargements. Une solution logicielle permet de ne pas utiliser un tel cache. La plupart des techniques logicielles [7, 9, 10, 13, 23] utilisent un grand bloc de mémoire partagé entre l'optimiseur et le programme à optimiser. Dans un tel système, il n'est pas possible de savoir facilement si un accès mémoire donné a provoqué un défaut de cache. Dans le système *ADORE* [23], on se base sur les compteurs du processeur *Itanium* pour y parvenir. Il s'agit alors d'un système hybride matériel/logiciel.

Dans ce rapport, nous proposons un système entièrement logiciel et dynamique basé sur une étude Markovienne des sauts entre les accès mémoire consécutifs. Généralement, un prédicteur Markovien utilise un historique des accès mémoire pour prédire le prochain effectué par le programme [19, 22]. Utilisant des données statistiques sur le comportement antérieur, il essaie de placer les données dans le cache avant que le processeur n'en ait besoin. Si cela est fait correctement, on peut obtenir une accélération considérable.

Notre modèle ne se préoccupe pas des adresses des accès mémoires mais des sauts réalisés entre deux accès. A la différence d'autres techniques, une grande partie de la mémoire concernée peut alors être considérée lorsqu'on modélise un comportement relativement constant. Donc, dans beaucoup de cas, peu de mémoire est nécessaire pour créer le modèle capable d'effectuer les préchargements. De plus, il considère un historique de plusieurs accès précédents pour faire sa prédiction, à l'instar d'autres prédicteurs qui se contentent simplement du dernier [19, 22]. Nos expériences montrent qu'en utilisant notre optimiseur, nous obtenons des accélérations significatives sur plusieurs programmes issus de bancs d'essai connus.

Dans le premier chapitre, nous présenterons un état de l'art concernant les optimiseurs dynamiques et les prédicteurs markoviens. Dans le chapitre 2, nous verrons les aspects théoriques de la mise en place d'un prédicteur de Markov et son intégration dans un modèle dynamique. Le chapitre 3 est consacré à expliquer les structures mises en jeu et les grandes étapes de l'implémentation. Ensuite, le chapitre 4 illustre la mise en place du modèle avec deux exemples concrets. Finalement, les expérimentations sur différents programmes sont présentées au chapitre 5.

Chapitre 1

Les travaux antérieurs

1.1 Les prédicteurs utilisant Markov

Un modèle de Markov étudie une suite de données et en tire des informations statistiques. Cela peut être la probabilité qu'une instruction A s'exécute après une instruction B, ou la probabilité que la donnée A soit accédée avant les données B,C ou D...

De façon plus formelle, et si nous nous restreignons à une étude d'accès mémoire, nous pouvons définir le modèle de Markov comme étant la récupération des triplets (S, y, n) où S est une séquence d'accès mémoire dits *antérieurs* et y un élément qui l'a suivie n fois. L'élément y n'a pas forcément suivi la séquence de façon directe (il a pu y avoir d'autres éléments entre la séquence S et l'élément y).

Si nous fixons S et construisons tous les triplets (S, y, n), nous obtenons alors tous les suivants de cette séquence et pouvons en déduire des probabilités afin d'extraire le successeur le plus probable.

Si nous parlons de séquences, c'est parce que nous pouvons nous intéresser à plus d'un élément avant de créer le triplet (S, y, n). Nous pouvons, par exemple, en avoir un de la forme $((A, B), C, n)$ qui signifie que l'élément A a été suivi par B puis par C (et ceci n fois).

Nous évoquons dans cette section deux implémentations d'un système d'optimisation dynamique basé sur un prédicteur de Markov.

Doug Joseph et Dirk Grunwald ont publié, dans la revue *IEEE Transactions on computers*, un article intitulé "Prefetching Using Markov Predictors" [19]. Dans ce dernier, ils présentent l'idée d'un cache spécial servant à étudier la suite des défauts de cache. Cette étude permet ensuite de prédire le prochain défaut en supposant qu'une séquence passée se reproduira. Par exemple, si nous remarquons qu'après un défaut de cache A, il y a eu un défaut de cache B, la prochaine fois que A provoque un défaut, on suppose qu'ensuite B le fera aussi.

Leur système se base sur les adresses parcourues, et non pas sur les sauts entre chaque accès, et uniquement sur la connaissance du dernier accès qui a provoqué le défaut de cache.

Le système présenté dans [22] utilise plusieurs types de prédicteurs de Markov qui exécutent le programme une première fois afin de récolter les informations nécessaires au bon fonctionnement du système.

La première méthode ressemble beaucoup au système précédent. Par contre, la deuxième essaie de mémoriser plusieurs accès qui suivent chaque défaut de cache. Par exemple, supposons que la suite des défauts de cache est A, B, C, D, alors B, C, D seraient tous répertoriés comme étant des suivants de A. De cette façon, on est capable de précharger les trois adresses (si nécessaire et si possible) afin d'augmenter les chances d'avoir au moins un préchargement correct par étape.

Les résultats, bien que n'étant que des simulations de ce qui pourrait être obtenu, sont tout de même intéressants. En effet, l'utilisation de la deuxième méthode semble plus rentable dans certains programmes.

1.2 Les optimiseurs non Markoviens

1.2.1 Dynamo

En 2000, une équipe de Hewlett-Packard présente un optimiseur qu'elle nomme *Dynamo* (Dynamic Optimisation) [7]. Sa principale caractéristique est d'être transparent à l'utilisateur. On entend par transparent le fait que le programmeur ne soit pas obligé de modifier son code pour l'utiliser. Il suffit d'employer une option spécifique du compilateur. Pour résumer, nous dirons que le système interprète le code en cours d'exécution et, lorsqu'il remarque qu'une zone du programme est souvent utilisée, il l'optimise avec des optimisations légères. Ce nouveau code est mis dans un bloc de mémoire partagée et lorsque le programme veut exécuter l'ancienne version, il est redirigé vers la version optimisée.

Une technique mise en place dans ce système est l'utilisation de "vidanges" qui permettent d'utiliser une taille mémoire raisonnable. Les auteurs se basent sur l'idée qu'un programme se décompose en un nombre fini de phases. Une phase est caractérisée par un certain comportement du programme. Si *Dynamo* est en train de créer beaucoup de nouveaux blocs, une nouvelle phase s'est déclarée et donc les anciens blocs ne serviront probablement plus. *Dynamo* vide alors sa mémoire et recommence la création de codes optimisées. Par conséquent, un des inconvénients de ce système est la possibilité d'optimiser plusieurs fois le même bloc.

1.2.2 *Deli*

En 2002, une autre équipe de Hewlett-Packard présente un projet dénommé *DELI* (Dynamic Execution Layer Interface) [13]. Ce système possède deux formes d'exécutions :
- Une version transparente qui reprend les caractéristiques principales du système *Dynamo*.
- Une version sous forme d'API (une quinzaine de fonctions) qui permet à l'utilisateur de paramètrer le système *DELI*.

C'est la version API qui transforme l'optimiseur en un système plus puissant. En effet, il est capable de s'exécuter sur un PocketPc en optimisant directement le système d'exploitation, mais aussi de modifier dynamiquement le code. Supposons que le processeur puisse exécuter une (ou plusieurs) instruction assembleur de façon plus rapide avec une autre instruction spécifique mais que le compilateur ne peut pas générer [1]. On peut changer directement le code assembleur à la main ou utiliser *DELI* pour le faire dynamiquement. Pour chaque instruction qui doit être modifiée, *DELI* la transforme avant de l'envoyer au processeur (et ceci en un temps relativement faible). Ce n'est qu'un exemple du potentiel de ce système.

1.2.3 *Adore*

Bien que les systèmes *Dynamo* et *DELI* soient présentés comme étant des systèmes performants, ce sont surtout des systèmes d'optimisations d'instructions et non d'accès mémoire. *ADORE* (ADaptive Object code REoptimization) [23] est un système qui insère des instructions de préchargement de données dans le code source afin d'accélérer le programme. C'est un système dynamique qui utilise les mêmes idées que *Dynamo* puisqu'il interpréte le code exécuté et, lorsqu'il remarque que l'exécution reste dans la même portion de code, il la met de côté, l'optimise et la range dans un bloc mémoire partagé (pour le moment, ce système ne s'intéresse qu'au cas des boucles). Les différences majeures sont :
- *Dynamo* utilise un système de compteurs pour savoir si une zone est souvent utilisée. *ADORE* utilise les compteurs matériels du processeur *Itanium* pour

[1] comme, par exemple, les instructions multimedia SIMD de type MMX.

connaître la position du compteur programme, le nombre de défauts de cache dans cette zone, etc.
- *Dynamo* optimise les instructions, tandis que le système *ADORE* se place au niveau de la mémoire. Il insère des instructions de préchargement *lfetch* dans les boucles, ce qui permet d'éviter des défauts de cache coûteux.
- *Dynamo* va tenter d'optimiser le plus de portions de code possible afin de rentabiliser sa présence. *ADORE* est relativement peu coûteux, car il peut détecter les défauts de cache d'une certaine zone, donc améliorer uniquement les portions de code vraiment optimisables.
- *Dynamo* peut s'exécuter, en théorie, sur n'importe quelle architecture. *ADORE*, lui, ne peut s'exécuter que sur un *Itanium* (ceci est dû aux compteurs matériels qu'il utilise).

Bien que leur système ne s'intéresse qu'au cas des boucles, il est tout de même automatique. Dans les différents programmes qui ont été testés, deux ont été accélérés à plus de 20%, alors qu'ils étaient compilés au niveau d'optimisation 3 (option de compilation -O3). Malheureusement, ce système est entièrement dépendant de l'architecture et n'est donc pas portable.

1.2.4 Un autre système dynamique sous *Windows*

En 2002, Chilimbi et Hirzel [10] proposent une solution de préchargement dynamique qui repose sur deux programmes déjà existants : *Vulcan* et *Sequitur* [25, 27]. C'est à l'aide de ces derniers qu'ils implémentent un système capable de lancer des préchargements efficaces. *Vulcan*, est un outil capable de modifier du code binaire, même dans le cas de threads multiples. *Sequitur* permet d'étudier une suite d'accès et d'en tirer des motifs en construisant une grammaire. C'est en mettant ces deux outils en pratique que les suites d'accès intéressantes sont identifiées afin de mettre en place des préchargements.

Mais, de nouveau, leur système ne s'intéresse qu'aux adresses parcourues et, de plus, utilise deux sous-programmes. Ce n'est donc pas un système entièrement autonome et portable, puisque *Vulcan* ne fonctionne pour l'instant que sous *Windows*.

Chapitre 2

La théorie

2.1 Présentation

Comme nous l'avons mentionné dans l'introduction, notre optimiseur dynamique va tenter de rendre un programme cible plus rapide. Un des enjeux principaux est de rentabiliser son utilisation. Son exécution et celle du programme à optimiser peuvent avoir lieu :

- sur deux processeurs différents ;
- sous la forme de deux threads ;
- de façon séquentielle et entrelacée, en passant du programme à l'optimiseur (et vice versa) de temps en temps.

En principe, la technique utilisant deux processeurs sera plus avantageuse car l'optimiseur aura les ressources matérielles pour faire des calculs plus lourds sans pour autant ralentir le programme de base. Mais cette solution est bien évidemment plus coûteuse en matériel !

Une solution intermédiaire est d'utiliser des threads. En effet, si nous avons à notre disposition plusieurs processeurs, nous pouvons aisément nous retrouver dans le premier cas. Par contre, si on n'en possède qu'un seul, les deux threads vont être concurrents, mais l'avantage par rapport à une solution séquentielle est que, dans ce cas, si le programme de départ fait des entrées-sorties, le deuxième thread peut en profiter pour avancer dans ses calculs. Notons que les processeurs hyperthreadés permettront de rendre cette solution encore plus attrayante comme dans [20, 24].

Puisque les deux premières versions nécessiteraient des mécanismes de synchronisations, le temps d'exécution de notre optimiseur risquerait de devenir trop important. C'est pour cela que nous avons choisi, dans un premier temps, un modèle séquentiel.

Dans la section suivante, nous présentons les idées de base pour la mise en place du modèle Markovien et son insertion dans un système dynamique.

2.2 Markov, une technique utilisant des statistiques

Notre objectif est la réduction du nombre de défauts de cache qui entraînent de grandes pertes de temps. Si le programme effectue de grands sauts en mémoire entre ses accès, il y a une forte probabilité qu'il provoquera un nombre important de défauts de cache et passera donc plus de temps en attente de données qu'en exécution de code. Notre optimiseur tentera de prédire les prochains sauts en mémoire afin d'utiliser le mécanisme de préchargement de données du processeur, et ainsi éviter les défauts de cache.

C'est à l'aide d'informations statistiques que nous modélisons le comportement mémoire des programmes cibles. Dans un cas idyllique, le programme fait toujours la même séquence d'accès, il suffit alors de la connaître et de charger en avance les données dont il a besoin.

Mais remarquons que, pour effectuer des préchargements avec cette technique, il faudrait connaître toute la suite des accès qui peut être arbitrairement longue. Cette solution peut servir à montrer l'accélération d'un programme avec des préchargements mais ne servira pas en général pour une véritable implémentation (sauf pour des exemples bien particuliers).

Une autre solution répertorie les différents accès provoquant un défaut de cache sous forme de couples (X, Y) où X et Y sont deux défauts de cache successifs. La prochaine fois que X fera un défaut de cache, le préchargement de Y pourra être effectué. Cette solution peut être problématique si le nombre d'adresses accédées distinctes est trop grand. La plupart des optimiseurs [19, 22] l'implémentent, mais pour limiter la taille mémoire nécessaire, ils se privent d'informations et se servent d'approximations ou d'heuristiques pour combler ce manque.

Une troisième solution, qui est la notre, étudie les pas (ou sauts) du programme entre chaque accès. A partir d'une telle suite, nous pouvons essayer de tirer des motifs. Par exemple, après chaque saut de 4 octets, le programme en a effectué un de 8 octets. Ce genre d'information permettrait de précharger une donnée lorsque le dernier saut a été de 4 octets. Cette proposition peut nécessiter trop de mémoire si le nombre distinct de sauts est trop grand, mais nous supposons que cette solution est plausible pour un programme qui a un comportement mémoire "suffisamment constant".

Remarquons que nous pouvons généraliser la profondeur avec laquelle nous essayons d'établir une prédiction en nous intéressant aux n derniers sauts pour précharger le prochain. Ceci engendrera nécessairement des prédictions plus fines. Prenons un exemple pour démontrer ce point :

Soit la suite de sauts effectués par un programme donné par la Figure 2.1.

1 2 16 2 32 2 16 2 32

FIG. 2.1 – Une séquence de sauts

Si la prédiction est uniquement en fonction du dernier saut, nous dirions qu'après un saut de 2, il y a équiprobabilité qu'il y ait ensuite un saut de 16 ou de 32. Par contre, si nous nous intéressons aux deux derniers sauts, alors nous pouvons dire qu'après 16 2 il y a un 32, mais après 32 2 il y a un 16. C'est dans ce sens que nous disons que nous pouvons prédire de façon plus fine.

2.2.1 Version naïve

Une solution pour calculer les prédictions à effectuer est d'établir dans un tableau toutes les différentes séquences de profondeur n et les différents accès qui ont été effectués ensuite.

Dans l'exemple de la table 2.1, on remarque que si on ne considère que le dernier saut, on a une incertitude en ce qui concerne le saut qui va suivre un pas de 2. Une possibilité serait de précharger tous les suivants potentiels, nous aurions ainsi plus de chance de précharger la donnée que le programme va utiliser. Dans la théorie, ceci semble plausible, mais, en prenant en considération qu'un préchargement a un temps d'exécution non négligeable et que les processeurs ont généralement un nombre maximum de préchargements pouvant être en attente (c'est le cas par exemple pour le *Pentium III* et l'*Itanium-2* [18]), il est donc déconseillé d'en utiliser un trop grand nombre à chaque étape. On pourrait imaginer un système qui précharge, par exemple, les deux meilleures prédictions.

Notons tout de même plusieurs inconvénients à l'utilisation d'un tel tableau :

1. La taille mémoire utilisée : nous devons garder en mémoire toute la suite et si la profondeur n devient trop grande, cela risque de demander trop de ressources. Nous nous intéressons à réduire le nombre de défauts de cache, mais si nous

Sauts précédents	Prochain saut
1	2
2	32, 16
16	2
32	2
1 2	16
2 16	2
16 2	32
2 32	2
32 2	16

TAB. 2.1 – Un exemple de table de prédiction. En première partie, nous avons la prédiction avec une profondeur de 1 et, en dessous, une profondeur de 2.

utilisons trop de mémoire nous risquons d'en provoquer un grand nombre dans notre recherche de la prochaine prédiction à effectuer. Une solution serait de compresser la suite et les données mais le temps de compression et décompression risque de rendre le système inefficace.

2. Le nombre de prédictions possibles : si nous avons un système de tableau, nous devons déterminer le nombre de différentes prédictions nécessaires par séquence étudiée. Malheureusement, ce nombre ne pourra pas toujours être connu à l'avance puisque les programmes n'ont pas tous le même comportement. Nous avons donc besoin d'une stratégie d'allocation. Si nous implémentons une technique incrémentale (d'abord on alloue une case, puis lorsqu'une deuxième prédiction est déterminée, on alloue un tableau plus grand avec une recopie...), cela peut devenir désastreux. Si nous utilisons un système de bloc (on en alloue pour dix prédictions possibles et si nous débordons, on en allouera dix autres), nous risquons de gaspiller de la mémoire. Nous pouvons bien sûr choisir une solution hybride en utilisant des listes chaînées ou des tableaux chaînés, mais le temps de parcours de ces structures est souvent plus élevé et risque de produire plus de défauts de cache.
3. Le nombre de séquences : de la même façon, nous ne pouvons pas connaître le nombre de séquences différentes au moment de la compilation. Il faudrait donc trouver une solution pour ne pas gaspiller trop de mémoire et pour ne pas dépenser trop de temps pour des recopies.
4. Un autre problème est le temps nécessaire pour récupérer le saut que nous voulons prédire (il faut retrouver la séquence que le programme a effectué dans la colonne de gauche du tableau). En effet, même avec un système de hachage (ou un arbre de recherche), cela risque de devenir trop long. On pourrait imaginer une solution qui lie les lignes du tableau entre-elles.

Sauts précédents	(Prochain saut, Indice pour la prochaine prédiction)
1	(2,1)
2	(32,3), (16,2)
16	(2,1)
32	(2,1)

TAB. 2.2 – Le tableau optimisé

La table 2.2 montre ce que pourrait être une représentation sous forme de tableau des prédictions. Nous avons supposé que le tableau est indexé de 0 à 3. Par exemple, si nous savons que notre séquence précédente est uniquement un saut de 1, nous comparons cette séquence avec celles de la colonne de gauche. Une fois trouvée, nous pouvons récupérer le couple (2,1) du tableau (couple de la colonne de droite). Le 2 représente notre prédiction, le 1 l'indice de la ligne où nous devons nous rendre si 2 est effectivement le prochain saut. Nous n'avons donc plus besoin de faire une recherche de ligne tant que l'optimiseur a raison. On pourrait même avoir directement le pointeur vers la case mémoire, cela permettrait d'économiser une indirection à chaque étape.

2.2.2 Version plus adaptée

Nous cherchons une solution conservant toute l'information de la technique précédente, nécessitant moins de mémoire et donnant presque immédiatement le prochain saut.

La solution choisie est d'utiliser une structure de graphe. Revenons à notre trace de la figure 2.1 et supposons que nous avons un graphe qui la représente. Si nous nous trouvons sur un de ses noeuds, alors cela veut dire que l'historique (les sauts précédents) nous y a conduit. Les fils de ce noeud (s'il y en a) représentent les prochains sauts possibles (à la connaissance de l'optimiseur) et nous pouvons donc prédire le meilleur de ces sauts (celui qui a été suivi le plus de fois) comme étant le prochain que fera le programme.

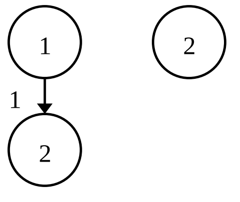

Fig. 2.2 – Un graphe Markovien (avec une profondeur de 2) après avoir donné deux sauts au modèle

Pour mieux exprimer ceci, regardons la figure 2.2. Elle représente le graphe en construction que nous obtiendrons après la connaissance des 2 premiers sauts de notre trace, et si nous ne nous intéressions qu'à une profondeur de 2, c'est-à-dire un historique d'au maximum deux sauts.

La signification de l'unique arc de la figure est, qu'après un saut de 1, il y aura, à notre connaissance, un saut de 2. Les noeuds n'ayant pas de fils signifient que nous ne savons pas encore ce qui se passera après. Par exemple, le groupe connexe (1,2) n'est pas attaché à un autre noeud puisque nous ne savons pas encore ce qui va suivre.

La racine 2 représente le fait que :
- nous ne savons pas ce qui s'est produit auparavant puisque ce noeud est une racine ;
- nous sommes dans l'incapacité de prédire correctement le prochain saut puisqu'il n'a pas de fils.

Remarquons encore deux choses sur les arcs du graphes :

1. Ils sont orientés et représentent la succession des sauts.
2. Ils sont pondérés, afin de compter le nombre de fois que chaque arc a été suivi.

Nous gardons un pointeur sur le noeud que nous nommons le *noeud courant*. Il exprime l'idée que les accès précédents nous ont menés à celui-ci. Par exemple, si les sauts effectués sont 1 puis 2, nous allons nous retrouver sur le noeud 2 du groupe connexe qui sera notre noeud courant.

Le schéma de fonctionnement du programme de prédiction est le suivant :

1. Le programme de départ fait un saut qui intéresse l'optimiseur ;
2. Il l'en informe ;
3. L'optimiseur cherche parmi les fils du noeud courant celui ayant le même saut ;
4. Le noeud courant devient ce fils.

Bien sûr, il arrive que le dernier saut fait par le programme ne se trouve pas parmi les fils du noeud courant. Nous avons donc besoin d'une solution pour permettre à notre optimiseur de reprendre son travail.

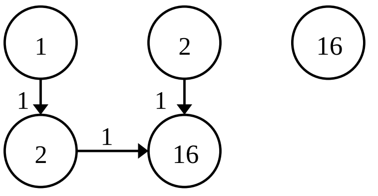

Fig. 2.3 – Un graphe Markovien (avec une profondeur de 2) après avoir donné trois sauts au modèle

La solution varie en fonction de la taille maximale du graphe souhaitée :

- Si nous voulons encore l'agrandir, nous allons créer un nouveau fils et d'autres noeuds. Par exemple, regardons ce qui se passe lorsque nous avons le graphe du schéma précédent et que nous apprenons que le prochain saut est 16 (Voir Figure 2.3).
  Remarquons qu'il y a une racine 16, et que rien n'y est encore attaché car nous n'avons aucune idée sur ce qui va se produire par la suite. Par contre, les deux 2 ont un arc vers un même noeud 16. Si nous étudions la signification de ces arcs, nous remarquons qu'après une suite (1,2), il y a eu un 16 donc on a créé cet arc. Mais n'oublions pas que l'on peut dire qu'après un simple 2, il y a eu un 16. D'où le deuxième arc sur le même noeud. Puisque nous nous sommes fixés un historique de deux sauts, le fait que les pas précédents étaient (1,2,16) ou (-453,2,16) ne nous intéresse pas. Nous voulons seulement connaître les deux derniers sauts, soient 2 et 16.
- Si la taille maximale a déjà été obtenue, l'idée est de chercher dans la forêt une racine qui a le même pas que le dernier saut fait par le programme. Si c'est le cas, ce noeud deviendra le noeud courant. Ses fils représentent tous les prochains sauts connus par l'optimiseur avec, comme unique information, le dernier pas effectué par le programme. Il suffit de prendre les meilleurs fils comme prédiction et continuer l'optimisation. En prenant une racine, nous perdons l'historique. Mais le fait que l'ancien noeud n'avait pas de fils correct signifie que nous n'avons jamais vu un tel comportement. Il vaut mieux oublier le passé et repartir à zéro.

Un autre cas particulier peut se produire lorsqu'il n'existe pas de racine ayant la bonne valeur. Si tel est le cas, la réaction de l'optimiseur peut varier :

- S'il considère qu'il n'a pas assez de données sur le programme, il peut créer le noeud pour étudier cette nouvelle suite ;
- Sinon, il y a encore deux possibilités :
  - Soit il ignore ce saut et ne prédit rien à cette étape, puis attend le prochain saut qui pourra peut-être avoir une racine correspondante ;
  - Soit il détruit son graphe et repart de rien, afin de le reconstruire. Comme pour d'autres optimiseurs [7, 23], nous considérons qu'un programme se déroule en un certain nombre de phases. Chacune étant caractérisée par un comportement mémoire différent, il est donc justifié de recommencer une nouvelle construction. Effectivement, si nous conservons l'ancienne, nous aurons des arcs déjà pondérés (avec des valeurs arbitrairement grandes) qui vont fausser le début de la nouvelle phase. Ceci peut mener à un nombre important de mauvaises prédictions.

Cette recherche de racine devant être la plus rapide possible, nous avons mis en place un arbre de recherche sur les différentes racines de la forêt. Cet arbre limitera, par rapport à un chaînage de base entre les différentes racines, le temps moyen de tests à faire avant de trouver le noeud voulu ou de savoir qu'il n'existe pas.

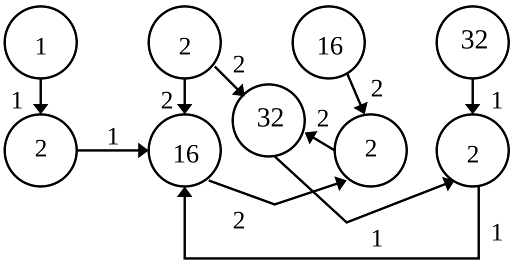

FIG. 2.4 – Un graphe Markovien (avec une profondeur de 2) après avoir donné tous les sauts au modèle.

La figure 2.4 montre le graphe une fois que toute la séquence a été traitée (en supposant que l'on n'arrêtait pas la construction). Remarquons que certains arcs sont maintenant étiquetés de 2 puisque nous les avons parcourus deux fois.

Notre graphe ressemble beaucoup aux arbres de suffixes ou aux automates de suffixes qui servent pour la recherche de motifs dans des chaînes de caractères [12]. Il est cependant un peu différent et peut donc être vu comme un mélange de ces deux représentations.

2.2.3 Distance

Nous appelons *distance* le nombre de sauts que nous prédisons à l'avance. Supposons qu'après une suite S, les sauts prédits soient A, B et C. Alors, il est possible de précharger les adresses $b + \sum S + A$, $b + \sum S + A + B$ ou $b + \sum S + A + B + C$, où b est l'adresse de base et $\sum S$ représente la somme de tous les sauts effectués dans la séquence S. Puisqu'un préchargement peut prendre un certain temps, si on précharge $b + \sum S + A$, il est possible que le programme arrive à l'instruction nécessitant cette donnée avant que le préchargement soit terminé. Le programme va donc devoir attendre. Dans ce genre de cas, il aurait été plus intéressant de précharger $b+\sum S+A+B$ ou $b+\sum S+A+B+C$ comme cela est fait également dans [22, 23].

2.2.4 Déroulement du modèle

Notre optimiseur fonctionne en 2 grandes phases :

1. Première phase : la construction. Cette phase est une phase de récupération de données sur le déroulement du programme. La version que nous utilisons est celle expliquée dans la Section 2.2.2.
2. Deuxième phase : une fois que le graphe est créé, le programme va pointer sur le noeud courant. Avec les informations qu'il possède, il précharge une donnée (lorsque c'est possible), cherche le prochain noeud courant et, si ce noeud est un des fils, incrémente la valeur de l'arc adjacent.

Ceci nous mène au problème difficile de détection d'un changement de phase. S'il y en a trop, nous allons passer notre temps à faire grossir ou à détruire notre graphe. Inversement, si nous ne détectons pas les changements, nous risquons de nous retrouver dans le cas où on ne peut plus rien prédire correctement (si nous arrêtons la construction mais ne changeons jamais le graphe) ou avec un graphe trop grand (si nous n'arrêtons pas la construction). Au final, nous ralentissons le programme. Il faut donc jongler entre l'utilisation mémoire et l'efficacité de l'optimiseur.

2.3 L'approche dynamique

La première version de notre optimiseur était un système assez général pour qu'il puisse s'adapter aux différents programmes. On a pensé qu'il serait possible de mettre en place plusieurs optimisations dynamiques en même temps (en utilisant diverses techniques), ou peut-être même de façon alternative.

L'optimiseur se découpait en une interface entre le programme de départ et les différents modules qui allaient être exécutés. Afin de permettre une souplesse à son utilisation, nous avions un système d'état (comme pour OpenGL par exemple).

Par exemple, nous voulions pouvoir écrire *Enable(ModuleMarkov)* ou *SetParam(4,PROF)*, pour respectivement activer le Module de Markov ou spécifier la profondeur que nous voulions. Vu que nous avions un optimiseur qui fonctionnait en deux phases, un indicateur signalait dans laquelle des deux nous nous trouvions. L'inconvénient de cette méthode était le nombre important de tests qui devaient être effectués à chaque appel de l'optimiseur. Il fallait vérifier si on utilisait le module Markov, puis si nous étions dans la phase de construction.

Assez rapidement dans la phase d'implémentation, il a été remarqué que cette technique, bien que pratique et permettant des modifications et ajouts ultérieurs, était trop coûteuse et ralentissait le programme. A la place, nous utilisons un pointeur de fonction qui change lorsque la phase actuelle du programme remarque qu'il est temps de s'arrêter. Pour le moment c'est un compteur qui est décrémenté à chaque appel et, lorsqu'il atteint zéro, nous changeons de phase.

Nous n'avons donc plus d'interface entre le programme et les différents modules puisque nous nous contentons, pour l'instant, d'étudier l'impact de l'utilisation du modèle markovien.

2.3.1 La rapidité est essentielle

Lors des premiers tests, l'optimiseur prédisait tous les fils du noeud courant et le *n-ème* saut dans le futur.

Pour l'instant, ceci a été mis de côté et l'optimiseur ne précharge que la meilleure prédiction de chaque noeud. Nous avons tout de même conservé l'idée de précharger avec un certain nombre de sauts en avance.

Dans le meilleur des cas, le programme ne passera pas beaucoup de temps dans la phase de construction. Cette phase peut alors être supposée négligeable en terme de temps d'*overhead*. On est bien entendu pas toujours dans le meilleur des cas : si le programme change trop souvent de phase ou si le graphe devient trop grand, le temps de construction sera loin d'être négligeable. Mais pour l'instant, et dans les tests que nous avons effectués, cette hypothèse est conservée.

2.3.2 Une première approche

Afin de permettre une souplesse optimale, la première solution à l'intégration de l'optimiseur dans un modèle dynamique était de faire une suite de tests pour savoir dans quelle phase nous étions. La technique basique découpait le déroulement de l'optimiseur en 4 phases et non 2. Nous expliquons brièvement leur utilité puis présentons comment ces étapes ont été ensuite fusionnées :

1. Construction : cette phase se contente de construire le graphe mais ne s'occupe pas des pondérations des arcs.
2. Comptage : c'est dans cette phase que nous mettons à jour les valeurs des arcs.
3. Sélection : dans cette troisième phase, nous choisissons la meilleure prédiction de chaque noeud, la mise à jour des arcs ne se fait déjà plus.
4. Prédiction : une fois que l'optimiseur se trouve ici, le graphe se fige. Nous nous contentons de précharger la meilleure prédiction et de comptabiliser les mauvaises.

2.3.3 Une solution plus adaptée

La rigidité de ce modèle n'est pas négligeable. Supposons par exemple que nous avons un noeud qui a deux fils A et B et, qu'au départ, c'est l'arc menant à A qui est presque toujours suivi. Lors de la phase de sélection, nous allons donc choisir A et les prédictions supposerons toujours que c'est A le meilleur fils. Mais si, après la phase de sélection, le programme commence à aller vers B, l'optimiseur va se tromper jusqu'à ce que le graphe soit reconstruit.

Pour résoudre ce problème, nous devons apporter une certaine souplesse et un dynamisme au modèle :

- Premièrement, pour ne pas avoir de pertes d'informations, la phase de comptage se fera tout au long de la vie de l'optimiseur. Ce n'est qu'une incrémentation de la valeur de l'étiquette de l'arc que nous suivons. La précédente phase de comptage peut donc être enlevée ;
- Deuxièmement, puisque nous voulons pouvoir changer *dynamiquement* la meilleure prédiction, la phase consacrée à la sélection du meilleur fils est inutile.

Ces deux changements de concept réduisent le nombre de phases à deux seulement.

2.3.4 La détection de phase

Un autre point capital pour un optimiseur dynamique est la possibilité de mettre en place un système d'arrêt s'il remarque qu'il est inefficace. L'étude des sauts réellement effectués par le programme cible permet de surveiller le nombre de mauvaises et de bonnes prédictions. Si le nombre de mauvaises prédictions est trop conséquent, nous détruisons le graphe et recommençons sa construction. Nous pouvons facilement stopper l'optimiseur si le nombre de remises à zéro devient trop important.

Chapitre 3

Implémentation

Dans ce chapitre, nous expliquons plus en détail comment l'optimiseur a été implémenté et intégré dans un système dynamique.

3.1 La structure Markovienne

Dans un programme d'une certaine ampleur, il est courant d'avoir plusieurs phases. Pendant chaque phase, il est possible qu'une seule zone mémoire soit traitée ou parcourue de façon intensive. Ceci peut mener à un grand nombre de défauts de cache. Le fait d'avoir plusieurs phases rend le travail de notre optimiseur plus délicat.

Si nous supposons qu'il n'y a qu'une seule phase, une unique structure globale, contenant toute l'information de l'optimiseur, est suffisante. Les fonctions de l'optimiseur l'utilisent et n'ont donc pas besoin de prendre une telle structure en paramètre. Ceci peut simplifier leur code et réduire *l'overhead* de l'optimiseur. Malheureusement, cette hypothèse n'est pas plausible puisque plusieurs zones mémoires distinctes peuvent être parcourues de façon différente.

En effet, les différentes séquences étudiées n'auront pas forcément les mêmes paramètres et ne basculeront pas forcément au même moment dans la phase de prédiction. Nous proposons donc que le programme déclare autant de structures que de séquences à surveiller.

3.1.1 Présentation

La structure qui contient toute l'information sur la séquence étudiée est donnée par la figure 3.1. Voici une brève explication :

- *prof* : profondeur de la construction ;
- *nbr_err* : nombre d'erreurs consécutives dans les prédictions ;
- *nbr_err_max* : nombre maximal d'erreurs consécutives avant d'effacer le graphe et reprendre la construction ;
- *depart* : racine de l'arbre binaire. Rappelons que le graphe possède un certain nombre de noeuds racines. Lorsqu'un saut se produit et que le noeud courant n'a pas de fils associé, l'optimiseur parcourt les racines du graphe afin de trouver un noeud ayant la même valeur. Cette recherche est facilitée par l'arbre de recherche *depart*. En fait nous n'avons pas de pointeurs vers le graphe, la seule façon d'y accéder est en parcourant cet arbre ;
- *cur* : tableau de pointeurs sur le graphe de prédiction, ces pointeurs servent à rendre la construction la plus rapide possible. Comme cela est expliquée dans la section 3.2.1 ;
- *iter* : indice qui facilite la construction et, plus particulièrement, l'usage du tableau *cur* ;
- *fct* : pointeur de fonction qui permet d'appeler soit la fonction de construction, soit la fonction de prédiction. L'utilisation d'un pointeur de fonction permet de ne plus avoir de tests pour vérifier quelle procédure doit être appelée. Nous accédons directement à la fonction concernée ;

```
typedef struct sMarkov
{
int prof;          //Depth of the Markovian model
int nbr_err;       //nbr of consecutive miss-predictions
int nbr_err_max;   //Max. of consecutive miss predictions
                   //before a flush
void* adresse;     //Current adress or base adress
Noeudbin depart;   //Root of the binary tree
Noeud *cur;        //Pointers on the graph
int iter;          //Index on the cur table

void (*fct)(struct sMarkov*, void *);  //Pointer function
int fct_ttlmax; //Maximum Time to live
                //for the construction function
int fct_ttl;    //Current time to live for
                //the construction function

int prof_prech; //Distance of the predictions

#ifdef __OPT_SPACE_LOC__
Noeud new_node_cur; //Pointer to the next free block
Noeud min;          //Last pointer on this block
int nbr_noeud;      //Number of nodes per block
int taille_bloc;    //Size of a block
#endif
}SMarkov,*Markov;
```

Fig. 3.1 – La structure markov

- *fct_ttl* : *time to live* de la phase de construction. A chaque appel de la fonction de construction, nous décrémentons ce nombre. Lorsque cette variable est nulle, nous passons à la phase de prédiction ;
- *fct_ttlmax* : valeur maximum que prend *fct_ttl*. A chaque reconstruction, *fct_ttl* est remis à cette valeur ;
- *prof_prech* : nombre de sauts que nous prédisons en avance. C'est donc le nombre d'adresses accédées en avance que nous prédisons. Si ce nombre est trop petit, le préchargement ne sera pas fini en temps utile ; par contre, s'il est trop grand, la donnée risque d'être écrasée avant de servir.

3.1.2 Une optimisation de la localité spatiale

Les quatres dernières variables servent à une optimisation de la localité spatiale des données qui a été mise en place pour améliorer les performances de notre optimiseur. Une optimisation spatiale rapproche en mémoire les données accédées consécutivement afin d'éviter des sauts trop grands. Nous avons déjà dit que si le graphe devient trop grand, il y a un risque que le nombre de défauts de cache, dû à l'optimiseur, devienne trop important. Nous avons donc mis en place une alternative qui permet de rapprocher les noeuds les plus externes afin de réduire ce problème.

L'idée est d'allouer un grand bloc dans lequel nous pouvons placer plusieurs noeuds du graphe. Nous y mettons en priorité les noeuds de profondeur maximale puisque, si le graphe fonctionne dans le meilleur des cas, on ne parcourera que ces noeuds là. Ensuite, ce n'est qu'une question de gestion du bloc, en gardant un pointeur sur la prochaine case libre pour allouer un autre bloc lorsque cela est nécessaire.

Voici les variables qui aident à gérer ce procédé :

- *new_node_cur* : prochaine case vide ;
- *min* : dernière case mémoire du bloc. Lorsque *new_node_cur* est égal à *min*, nous savons que c'est la dernière case du bloc. Nous l'utilisons puis nous réallouons un nouveau bloc, mettant à jour *min* et *new_node_cur* ;
- *nbr_noeud* : nombre de noeuds que nous voulons par bloc ;
- *taille_bloc* : taille d'un bloc. Cette information permet ne pas être obligé de recalculer la taille d'un bloc à chaque allocation.

3.1.3 Pointeurs *prochain*, *dernier* et *dernier2*

```
//Prediction node
typedef struct snoeud
{
int pas;      //Stride of this node
int premier;  //First node of this block?

//Now we have the info for the children of this node
struct snoeud *prochain; //most probable child

//We then have a LRU system between
//the two last children that followed this node
struct snoeud *dernier, *dernier2;

//List of the other children
struct slistnoeud *next;
//We have the number of times we used
//the edges prochain, dernier and dernier2
int vis_prochain, vis_dernier, vis_dernier2;
}SNoeud, *Noeud;
```

Fig. 3.2 – La structure d'un noeud

La structure d'un noeud du graphe est donnée à la figure 3.2. De nouveau, nous ne donnerons qu'une brève explication de son contenu :
- *pas* : étiquette du noeud. C'est le saut qui a provoqué la construction du noeud ;
- *next* : liste chaînée des fils du noeud ;
- *prochain* : fils le plus probable, c'est-à-dire celui qui a suivi le plus souvent ce noeud dans le parcours. Ce pointeur est mis à jour si un autre fils devient plus probable ;
- *dernier* : pointeur qui représente le dernier fils accédé, autre que le noeud *prochain*. Nous essayons de limiter la recherche du fils dans la liste *next*. En gardant un pointeur sur le dernier accès fait, nous avons une chance d'avoir tout de suite sous la main le bon fils (si ce n'est pas *prochain*).
- *dernier2* : Ce pointeur a le même rôle que *dernier*, il représente l'avant-dernier noeud accédé. On utilise une stratégie LRU (*Least Recently Used*) pour la gestion de ces deux pointeurs (Voir Table 3.1).
- *vis_prochain*, *vis_dernier* et *vis_dernier2* : Ces variables permettent de garder le nombre de visite à chaque noeud qui a été parcouru après le noeud courant. Les fils de chaque noeud sont stockés dans une liste chaînée *next* et nous voulons éviter de parcourir cette liste à chaque étape. Une solution est de conserver le nombre de visites aux noeuds et lorsque nous remplaçons, par exemple, le noeud *dernier2* à cause d'une mise à jour, nous modifions l'étiquette de l'arc qui conduit vers *dernier2*.

Remarquons que les trois pointeurs *prochain*, *dernier* et *dernier2* ne sont pas toujours des fils du noeud courant. En fait, si on suppose qu'un noeud A a deux fils

B et C, et que l'accès suivant est D alors on va faire une recherche pour une racine D. Si on la trouve, on liera A à D avec le pointeur *dernier*. C'est une solution pour rendre le modèle un peu plus souple.

Pour la mise à jour des visites aux noeuds, cette ambiguité ne pose pas de problèmes. Nous ferons simplement un test sur l'existence d'un tel fils et si ce noeud n'en n'est pas un, aucun arc ne sera mis à jour.

Etat de départ	(Saut récupéré,nbr)	Etat final
(B,10) (C,9) (D,1)	(B,10)	(B,20) (C,9) (D,1)
	(C,2)	(C,11) (B,10) (D,1)
	(D,1)	(B,10) (C,9) (D,2)
(B,10) (C,9) (D,2)	(D,10)	(D,12) (B,10) (C,9)
(B,10) (C,9) (D,1)	(E,1)	(B,10) (E,1) (C,9)
	(F,450)	(B10) (C,9) (D,1)

TAB. 3.1 – Des exemples de gestion des pointeurs *prochain, dernier* et *dernier2*. On suppose que le noeud courant est A, qu'il a 3 fils B, C et D et qu'il n'existe que A,B,C,D et E comme racines du graphe (F n'appartient donc pas au graphe).

Dans la table 3.1, on montre l'effet qu'aurait un certain nombre d'accès consécutifs d'un même saut à partir d'un même noeud.

Voici les explications de chaque colonne :

- La première colonne représente l'état initial d'un noeud. Nous donnons les noeuds *prochain*, *dernier* et *dernier2* avec leur nombre de visites par les arcs adjacents associé. Par exemple, $(B, 10)(C, 9)(D, 1)$ représente que B est le noeud *prochain* avec 10 visites passant par l'arc adjacent, et ainsi de suite.
- La deuxième colonne représente la récupération du prochain saut du programme lorsque le programme se trouve sur le noeud courant de la première colonne. En plus, nous supposons que, lorsque l'optimiseur reviendra sur ce noeud, il refera le même saut. C'est-à-dire, pendant l'exécution de l'optimiseur, à chaque fois que nous arrivons sur ce noeud courant, nous suivrons le même fils (s'il existe). Par exemple, $(B, 10)$ signifie qu'on a reçu, à partir du noeud ayant les pointeurs dans l'état de la première colonne, le saut B dix fois de suite.
- Finalement, la dernière colonne donne l'état des pointeurs après ces accès successifs.

La deuxième ligne montre le noeud C devenant le noeud le plus probable, il doit être pointé par *prochain*, B devient l'avant dernier noeud accédé (ou plutôt le dernier noeud le plus probablement accédé).

La troisième ligne montre la conséquence qu'aurait un seul accès à D : on n'a pas de permutation entre les pointeurs *dernier* et *dernier2*. Effectivement, on laisse les deux pointeurs en place pour éviter un effet de ping-pong (la permutation étant tout de même un procédé lourd puisqu'il faut aussi changer les variables *vis_dernier* et *vis_dernier2*). Le saut D migrera vers le pointeur *prochain* uniquement lorsque son nombre de visite dépassera celui de B (comme dans le cas de la quatrième ligne).

Dans l'avant dernière ligne, on remarque que E, bien que n'étant pas un fils de A, se retrouve dans le dernier noeud accédé. Ceci permettra de le prédire malgré le manque d'information qu'il y a eu pendant la phase de construction. Autoriser ce lien permet de retarder le moment où nous devrions reconstruire le graphe. Nous pouvons considérer ce lien comme étant un lien faible du graphe. Le noeud E étant considéré comme un fils temporaire de A.

Finalement, dans la dernière ligne, nous remarquons que, même si le saut F a été accédé 450 fois, un saut n'existant pas dans le graphe ne changera rien à la disposition des pointeurs. Dans ce cas, il serait judicieux de, soit reprendre la construction (et enrichir le graphe), soit recommencer à zéro. Ceci permettrait d'intégrer le saut F dans le graphe. La différence entre les deux derniers cas se manifeste lors de la prochaine étape. Le noeud E qui est devenu le pointeur *dernier* existe et c'est une racine de la forêt. L'optimiseur pourra donc continuer ses prédictions lorsque le noeud E sera pointé par *prochain*. Aucune racine F existe dans le graphe, donc nous ne pouvons pas correctement mettre à jour le pointeur *dernier* (sans créer de nouveau noeud).

L'utilisation de ces pointeurs permet de conserver les fils les plus probables et les plus récemment utilisés sans avoir besoin de parcourir la liste chaînée *next*. L'optimisation est efficace lorsque les noeuds parcourent des groupes de fils par vague. De plus, elle permet de rendre le modèle plus souple en permettant de lier des noeuds aux racines lorsqu'il y a un faible changement de comportement. Si le changement est trop fréquent, le graphe sera tout de même reconstruit.

3.2 Les algorithmes

Dans cette section, nous présenterons en détail les différentes étapes de construction du graphe de prédiction. En deuxième partie, nous expliquons comment la prédiction est implémentée.

3.2.1 Première phase : Création de noeuds

La Figure 3.3 remontre la création du graphe avec comme entrée la même suite. La construction du graphe se décompose en deux phases. La première se caractérise par de simples ajouts de noeuds comme on a pu le constater avec la sous-figure (a) qui reprend la figure 2.2. A ce stade, chaque noeud a, au plus, un père et le graphe est en fait une forêt.

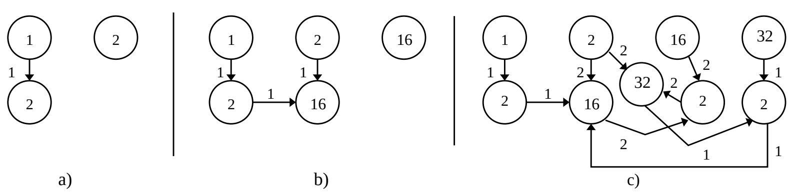

Fig. 3.3 – Reprise des figures précédentes

Afin de pouvoir créer ces arbres de façon simultanée, on garde un pointeur sur chaque noeud actif (c'est-à-dire, chaque noeud qui vient d'être construit ou qui vient d'être visité). Lorsqu'un saut est récupéré par l'optimiseur, chaque pointeur regarde si leur noeud a un fils du même saut, si c'est le cas il ne fait qu'avancer vers ce noeud. Sinon, il crée ce fils et avance.

Ensuite, avant de redonner la main au programme, nous regardons s'il y a une racine ayant la même valeur. On utilise un nouveau pointeur qui le référencera (si cette racine n'existe pas, on commence par l'ajouter à la forêt).

Dans notre exemple, à l'étape (a), les noeuds actifs sont les deux noeuds 2. Lorsque l'optimiseur reçoit le prochain saut 16, les noeuds actifs deviennent les noeuds 16. Remarquons que les noeuds actifs auront toujours le même étiquetage. Par contre, tous les étiquetages égaux à un noeud actif n'appartiennent pas forcément à des noeuds actifs.

Cette phase se déroule dans un nombre d'itérations égale à la profondeur voulue et c'est aussi le nombre de pointeurs nécessaires pour son bon fonctionnement, étant donné, qu'au départ, nous n'avons aucun pointeur et qu'on ajoute un pointeur par tour. Dans notre cas, on aura donc au maximum 2 pointeurs.

3.2.2 Deuxième phase : Les liens

Lorsque la première phase se termine, nous avons un arbre ayant une profondeur maximale. Dans la figure 3.3 (a), c'est l'arbre de gauche qui a cette profondeur.

Supposons que notre optimiseur soit dans cet état et étudions le comportement des pointeurs lors de l'envoi d'un nouveau saut à l'optimiseur :

- Le premier pointeur, qui se trouve sur la racine 2, va regarder si ce noeud a un fils étiqueté 16. Ce n'est pas le cas, donc il en créé un et y migre.
- Le deuxième pointeur, qui se trouve sur le 2 du groupe connexe (1,2), va se lier au même 16. Mais il ne migrera pas vers ce 16 puisqu'on a déjà un pointeur dessus. En effet, il va chercher une racine 16. Puisqu'il n'en trouve pas, il va en créer une et y migrer.

Lors de la prochaine étape, les rôles vont s'inverser. Le deuxième pointeur va ajouter un noeud à la racine 16 (puisque cette racine n'a pas encore de fils) et y migrera. Le premier pointeur liera son noeud courant au nouveau noeud avant rechercher une racine déjà existante.

Cette idée se généralise bien dans un cas de profondeur n. C'est la variable *iter* qui nous permettera facilement de savoir quel pointeur est à la profondeur maximale. En se basant sur cet indice, nous pouvons facilement poursuivre la création du graphe.

3.2.3 L'algorithme de construction

L'algorithme 1 permet de construire le graphe lors de la deuxième phase :

Nous noterons *cur[iter-1]* le pointeur qui se trouve à la profondeur maximale moins une. La variable *iter* pouvant être nulle, le noeud en question se trouverait en fait à l'indice *cur[prof-1]*. Cet abus est effectué pour une question de clareté de l'algorithme.

Nous distinguerons *le pointeur cur[n]* comme étant la valeur du pointeur et *le noeud cur[n]* comme étant vraiment le noeud pointé par *cur[n]*.

```
Soit prof la taille du tableau cur, c'est aussi la profondeur maximale du graphe de prédiction;
Soit stride le dernier saut effectué par le programme;
Soit cur le tableau de pointeurs courants du graphe, indexé de 0 à prof - 1;
Soit cur[iter] le pointeur du noeud qui se trouve à la profondeur maximale;
pour chaque noeud cur[n] du tableau cur à l'exception de cur[iter] faire
    si cur[n] n'a pas de fils ayant comme saut stride alors
        Le noeud cur[n] créé un fils d'étiquette stride
    fin
    Le pointeur cur[n] sera désormais égal à ce fils
fin
si cur[iter] n'est pas encore attaché à un fils ayant comme étiquette stride alors
    Nous créons un arc entre lui et le noeud cur[iter-1] (la valeur de ce pointeur a déjà été mise à
    jour)
fin
si aucune racine du graphe a comme étiquette stride alors
    On créé cette racine
fin
cur[iter] migre vers la racine ayant comme étiquette stride;
On incrémente iter (modulo prof);
```

Algorithme 1: Algorithme de construction (Deuxième phase)

Cet algorithme est assez simple, c'est une mise à jour des pointeurs avec un cas particulier pour le pointeur *cur[iter]*. Remarquons tout de même que lorsque nous parcourons un arc, il ne faut pas oublier d'incrémenter son étiquette.

3.2.4 L'algorithme de prédiction

Lorsque la construction prend fin, nous n'avons plus besoin de tous les pointeurs. L'optimiseur garde simplement le pointeur qui est à la profondeur maximale. Ensuite, il fait migrer ce pointeur à travers le graphe comme cela a été expliqué dans la section 2.2.2 page 11.

L'algorithme de prédiction est donné par l'algorithme 2. Son but est de prendre en compte l'accès mémoire que le programme vient d'effectuer. Pour l'instant, l'optimiseur a un pointeur vers le noeud courant et connaît la dernière adresse accédée. Il recherche

```
Soit noeud_cur le noeud courant;
Soit dern_saut le dernier saut effectué par le programme;
Soit pas_prediction le pas qu'il faut prédire à partir du noeud courant et de l'adresse de base;
Soit prochain le pointeur vers le prochain noeud courant le plus probable à partir du noeud
courant actuel;
Soient dernier et dernier2 les pointeurs dernièrement accédés à partir du noeud courant;
Soient sprochain, sdernier et sdernier2, les étiquettes de chaque noeud;
si noeud_cur est égal à NULL alors
|  Recherchons une racine ayant un saut égal à dern_saut;
|  Affectons ce résultat à noeud_cur (Si la racine n'existe pas, le résultat vaut NULL);
|  Incrémentation du nombre d'erreurs consécutives;
|  Pas de prédiction à ce tour;
sinon
|  si sprochain == dern_saut alors
|  |  noeud_cur = prochain;
|  |  On prédit le saut qui se trouve dans la variable pas_prediction du nouveau noeud
|  |  courant;
|  |  Mise à jour du compteur d'erreurs;
|  sinon
|  |  Mise à jour du compteur d'erreurs;
|  |  si sdernier == dern_saut alors
|  |  |  noeud_cur = dernier ;
|  |  |  On prédit le saut grâce à pas_prediction du nouveau noeud courant;
|  |  |  Mise à jour du compteur du noeud dernier;
|  |  sinon
|  |  |  si sdernier2 == dern_saut alors
|  |  |  |  noeud_cur = dernier ;
|  |  |  |  On prédit le saut grâce à pas_prediction du nouveau noeud courant;
|  |  |  |  Mise à jour du compteur du noeud dernier2;
|  |  |  sinon
|  |  |  |  Recherche du fils ayant le saut dern_saut;
|  |  |  |  si la recherche est fructueuse alors
|  |  |  |  |  Le noeud courant devient ce fils;
|  |  |  |  |  On prédit le saut grâce à pas_prediction du nouveau noeud courant;
|  |  |  |  |  Mise à jour du compteur du noeud de ce fils;
|  |  |  |  |  dernier2 devient ce fils;
|  |  |  |  sinon
|  |  |  |  |  Recherche d'une racine ayant le saut dern_saut;
|  |  |  |  |  Mise à jour de noeud_cur (NULL si pas de racine);
|  |  |  |  fin
|  |  |  fin
|  |  fin
|  fin
|  Mise à jour des pointeurs dernier, dernier2 et prochain;
fin
```

Algorithme 2: Algorithme de prédiction

un fils ayant le saut correspondant, c'est-à-dire la différence entre la dernière adresse accédée et l'avant dernière.

Pour parcourir les fils du noeud courant, il regarde d'abord le fils *prochain* puisqu'il s'agit du fils le plus suivi. Ensuite, il regarde les noeuds *dernier* et *dernier2*. Si aucun de ces noeuds ne correspondent, il parcourera la liste chaînée *next*.

Comme il a déjà été dit, l'optimiseur posséde un compteur surveillant le nombre de mauvaises prédictions. C'est dans cette fonction qu'il est mis à jour. Si le fils *prochain* est suivi, le nombre d'erreurs est remis à zéro. Dans tous les autres cas, le compteur est incrémenté et testé (s'il est supérieur à un seuil donné) pour voir s'il ne serait pas mieux de recommencer le graphe.

Puisque la valeur du saut prédit est en fonction du fils *prochain*, si le prochain saut du programme n'est pas celui-ci, on aura fait une mauvaise prédiction (bien que, si la distance de prédiction est assez grande, il est possible de se retromper et donc de prédire correctement). C'est pour cela que, même si le saut effectué correspond à un des fils *dernier* ou *dernier2*, on incrémente le compteur d'erreurs.

Lorsque nous suivons un arc vers un des fils, son étiquette est incrémentée. Si ce fils est un des noeuds *dernier* ou *dernier2*, alors on compare la valeur de ce compteur avec celui du noeud *prochain*. Si elle est supérieure, ce noeud devient le noeud *prochain* (et le prochain devient *dernier*, et *dernier* devient *dernier2*).

Si une telle mise à jour est effectuée, nous devons recalculer la variable *pas_prediction* puisqu'elle dépend du noeud *prochain*.

Enfin, si le nouveau noeud courant est un autre fils de la liste *next* ou une racine, ce noeud va être pointé par *dernier*. On met à jour le compteur de l'arc (s'il existe) menant à *dernier2* à l'aide de *vis_dernier2*. Le noeud *dernier* devient *dernier2*.

3.3 Markov dans le modèle dynamique

Dans cette section, nous présentons comment les deux algorithmes vus précédemment s'intégrent dans le modèle dynamique.

3.3.1 Les 5 fonctions de l'API

L'optimiseur que nous proposons n'est pas encore entièrement autonome. Pour le moment, il faut intégrer au code du programme à optimiser de petites fonctions pour paramètrer l'optimiseur et lui envoyer les données accédées que nous souhaitons précharger. Cette API se forme de quelques fonctions données par la table 3.2.

Nom de fonction	Description
initialize	Initialise la structure markov
set_param	Permet de paramétrer la profondeur de la construction, la distance de préchargement
set_adress	Permet de mettre à jour l'adresse de base
fct	Fonction qui sert de lien entre le programme de départ et notre modèle
clear	Désalloue la structure

TAB. 3.2 – Les fonctions de base du modèle

3.3.2 *fct* et *set_adress*

Le pointeur *fct*

Nous avons mis *fct* dans la table 3.2 parce qu'elle joue un grand rôle dans le déroulement de l'optimisation. C'est à travers cette fonction que la construction ou le préchargement peut se faire. Il y a en fait 4 fonctions différentes que *fct* est susceptible d'appeler :

– *first_ad* : cette petite fonction met simplement à jour l'adresse de base. Puisque nous utilisons les sauts effectués, il nous faut un repère avant de commencer la construction. Cette fonction nous permet de récupérer l'adresse de base puis change automatiquement *fct* pour qu'il pointe sur *const_begin* ;
– *const_begin* : cette fonction sert de départ à la construction. Elle gère la première phase (Voir section 3.2.1) et ensuite, lorsque nous arrivons à la deuxième phase, elle dirige *fct* vers *construct*.
– *construct* : *construct* permet de terminer la construction. Elle représente la deuxième phase (Voir Section 3.2.2) de la construction du graphe. C'est elle qui décrémente la variable *ttl_live* de la structure markovienne. Lorsque cette variable est nulle, *fct* pointera sur *predict*.
– *predict* : c'est cette fonction qui précharge les données. Elle effectue des tests pour vérifier que le modèle est efficace et si ce n'est pas le cas, décide de reconstruire le graphe (on repasse donc à la fonction *first_ad*).

Nous avons dit précedemment que le modèle ne posséde que deux phases. Ceci est toujours vrai car la première fonction n'est pas une phase à elle toute seule, elle n'est simplement qu'une fonction de démarrage. Les deux parties de construction ont été séparées pour permettre l'économie d'un test. Ensemble, elles constituent une phase entière. Il y a donc bien deux phases.

L'utilité de *set_adress*

L'intéret de la fonction *set_adress* n'est pas forcément évident. Son utilisation ne va pas directement accélérer le programme et ne va pas augmenter le nombre de bonnes prédictions.

0	48	60	36	24	72	**80**	48	60	36	24	72	**64**	48	60	36	24	72	**-80**	48	60	36	24	72

TAB. 3.3 – La suite sans l'utilisation de *set_adress*

Prenons un exemple en supposant que nous avons un programme qui fait 6 accès à un tableau mais à des indices différents (Voir table 3.3). Si nous ignorions ses sauts d'indices et concentrions nos efforts sur les 5 accès nous aurions un graphe plus simple. Malheureusement, entre le 5^{e} accès d'un parcours et le 1er du parcours suivant, le saut ne pourra pas être ignoré. Nous aurons donc des noeuds supplémentaires qui vont représenter tous les différents sauts entre les parcours. Ce nombre peut devenir arbitrairement grand.

0	48	60	36	24	72	**0**	48	60	36	24	72	**0**	48	60	36	24	72	**0**	48	60	36	24	72

TAB. 3.4 – La suite avec l'utilisation de *set_adress*

Par contre, si avant le premier accès nous utilisons la fonction *set_adress* sur cet accès, nous aurons un saut de 0 entre le 5^{e} accès et le 1er du prochain parcours (Voir table 3.4). En effet, avant d'obtenir le 1er saut, nous avons changé l'adresse de base. Pour l'optimiseur, lorsqu'il reçoit ce premier saut, c'est comme si le programme original faisait un deuxième accès sur la même donnée. Nous avons donc confondu tous les sauts différents en un saut fictif de 0. Comme nous l'avons mentionné, ceci ne rendra pas le modèle plus précis mais il permet de réduire considérablement la taille du graphe et donc la taille mémoire utilisé. Bien sûr, cette solution ne présente un intéret que si le nombre de sauts intermédiaires distincts est important. Cela peut être le cas lors de nombreux appels à une fonction parcourant toujours de la même manière des segments mémoire différents. Pour un exemple concret de l'utilisation de *set_adress*, voir le paragraphe 4.2.3 page 27.

Chapitre 4

Exemples

Dans ce chapitre, nous présentons en détail deux exemples d'utilisation de notre modèle Markovien. Tout d'abord, nous présentons le programme *treeadd* provenant du banc d'essai *Olden* [1] et ensuite *181.mcf* (qu'on nommera par la suite *mcf*) provenant du banc d'essai *Spec2000* [3].

Nous expliquons les procédés d'initialisation de la structure markovienne, de mise en place d'une surveillance de séquences d'accès mémoire puis de désallocation de la structure.

4.1 treeadd

Le programme *treeadd* crée puis parcourt un arbre binaire par la gauche. Nous avons modifié le code de sorte à parcourir plusieurs fois le même arbre. Avec ce changement, plus de 90% du temps est passé dans la fonction de parcours *Treeadd*. Notre modèle s'intéresse donc aux accès mémoire effectués dans cette fonction.

4.1.1 Initialisation

```
...
#ifdef __MARKOV__
     m1 = initialize();

     set(m1,&j,M_ARRCONS);
     set(m1,&k,M_PROF);
     set(m1,&l,M_ELAG);
     set(m1,&m,M_ERRMAX);
#endif
...
```

Fig. 4.1 – L'initialisation de la structure

Observons la figure 4.1. Tout d'abord nous appelons la fonction *initialize* qui alloue la structure markovienne avec des valeurs par défaut. Il n'y a qu'un accès qui nous intéresse dans ce programme, donc nous n'avons qu'une seule structure. Nous verrons par la suite un exemple qui en nécessite deux.

Nous utilisons une encapsulation de macros pour permettre une compilation du code sans optimisation dynamique. Ces macros englobent les appels du modèle, mais aussi l'inclusion du fichier d'entête contenant la déclaration de la variable *m1*. Ceci permet, si le programmeur le souhaite, de revenir au code original de façon simple et rapide.

La fonction *set* permet de préciser les paramètres que le programmeur a choisi.

Voici les différentes options et leur signification :

- M_ARCONS : *time to live* de la phase de construction, c'est le nombre de fois que la fonction de construction sera appelée ;
- M_PROF : profondeur du graphe de prédiction qui représente le nombre d'accès antérieurs voulu pour déterminer le prochain saut ;
- M_ELAG : distance de préchargement, c'est-à-dire le nombre de sauts que l'optimiseur prédit en avance.
- M_ERRMAX : nombre maximum d'erreurs consécutives avant une reconstruction du graphe ;

4.1.2 Appel de la fonction *fct*

```
int TreeAdd (t)
     register tree_t    *t;
{
...
#ifdef __MARKOV__
        m1->fct(m1,t);
#endif
    tleft = t->left;
    leftval = TreeAdd(tleft);
    tright = t->right;
    rightval = TreeAdd(tright);
    value = t->val;
    return leftval + rightval + value;
}
```

FIG. 4.2 – Mise en place de *fct* dans l'exemple *Treeadd*

La figure 4.2 montre l'utilisation de la fonction *fct*. C'est grâce à cette fonction que nous passons en deuxième paramètre l'adresse accédée à l'optimiseur. Le premier paramètre consiste en la structure markovienne correspondant à la séquence étudiée. Remarquons que nous appelons *fct* qu'une seule fois. Nous pourrions aussi, par exemple, le faire entre les deux appels récursifs ou même après. Dans le cas de ce programme, cela n'est pas rentable puisque le graphe devient trop grand et ne semble pas pouvoir prédire correctement le comportement mémoire.

En revanche, dans la section 4.2.3, nous verrons un exemple où plusieurs appels sont utiles.

Dans cet exemple, le code à optimiser se trouve dans une fonction récursive. Les données qui provoquent les défauts de cache sont les différents accès aux noeuds de l'arbre binaire. Remarquons la simplicité de mise en place de l'optimiseur dans cet exemple, puisqu'il suffit d'une seule ligne pour permettre au modèle d'enregistrer le comportement mémoire, changer de phase, décider s'il faut reconstruire le graphe et lancer les préchargements.

4.1.3 Clear

Finalement, lorsque le programme se termine, ou lorsque nous savons qu'il n'a plus besoin du modèle markovien, nous appelons la fonction *clear*. En regardant la figure 4.3, nous constatons qu'il n'y a qu'un seul appel qui permet de faire tout le travail de désallocation de la structure markovienne et du graphe de prédiction.

4.1.4 Discussion

L'exemple de *treeadd* permet d'illustrer l'utilisation de l'optimiseur proposé. Mais lors de la mise en place du modèle, il y a quelques points importants à déterminer au préalable :

```
...
#ifdef __MARKOV__
    clear(m1);
#endif
...
```

Fig. 4.3 – Désallocation de la structure markovienne

1. Quel est l'accès (ou les accès) qui provoque les défauts de cache : il existe plusieurs outils permettant d'identifier le, ou les accès coûteux [16, 14] ;
2. Les paramètres du modèle : à l'heure actuelle, ceux-ci sont déterminés manuellement. Mais dans l'optique de rendre le système plus transparent, nous projetons de le rendre plus indépendant et autonome, voire entièrement automatique ;
3. Le moment où la structure peut être désallouée : le plus tôt est bien sûr le mieux puisque cela libère de la mémoire.

Le programme *treeadd* permet de montrer que ce système peut être utilisé dans un cadre général. Il ne se préocuppe pas de savoir si l'accès se trouve dans une boucle ou dans une fonction récursive.

4.2 mcf

Dans l'exemple précédent, nous avons vu comment mettre en place le modèle pour suivre le comportement d'un accès. Dans le programme *mcf*, nous allons montrer comment gérer plusieurs séquences simultanément.

4.2.1 Les fonctions intéressantes

Ce programme est composé de deux fonctions dominantes (en terme de temps) : *price_out_impl* et *refresh_potential*. Dans notre exemple, nous ne nous intéressons qu'à la fonction *price_out_impl*.

4.2.2 Initialisation

La fonction *price_out_impl* posséde deux séquences d'accès mémoire qui sont intéressantes à modéliser. Le comportement des deux n'étant pas forcément le même, nous avons sûrement besoin de paramètres distincts. Il n'est pas opportun d'avoir une seule structure markovienne.

Une solution intermédiaire serait de supposer que la structure markovienne a un tableau de paramètres, chaque séquence ayant un indice particulier. Mais n'oublions pas que si les paramètres sont différents, le moment où le programme veut changer de phase pour une des séquences n'est pas forcément le bon moment pour l'autre. Remarquons aussi que, même si les paramètres sont identiques, le nombre d'erreurs consécutives, l'adresse de base et le graphe de chaque séquence peuvent tout de même être différents. En résumé, nous aurions besoin d'une structure contenant un tableau de structures markoviennes.

Puisque les différentes séquences n'ont pas d'intéractions entre elles, nous avons décidé de ne pas créer une telle structure. Dans l'exemple suivant, nous avons besoin de deux structures et nous avons choisi de les déclarer séparément.

Nous ne donnerons pas les codes sources d'initialisation et de désallocation puisqu'ils sont semblables aux figures 4.1 et 4.3, excepté le fait que nous le faisons pour deux structures.

```
...
#ifdef __MARKOV__
if(arcin)
        {
        set_adress(m1,(void*) arcin->tail);
        set_adress(m2,(void*) arcin->tail->mark);
        }
#endif

while( arcin )
        {
            tail = arcin->tail;
            #ifdef __MARKOV__
                (*m1->fct)(m1, arcin->tail);
            #endif
            if( tail->time + arcin->org_cost > latest )
            {
                #ifdef __MARKOV__
                        (*m2->fct)(m2, (void *)tail->mark);
                #endif
                arcin = (arc_t *)tail->mark;
                ...
                #ifdef __MARKOV__
                        (*m2->fct)(m2, (void *)tail->mark);
                #endif
                arcin = (arc_t *)tail->mark;
        }
...
```

FIG. 4.4 – Mise en place de *fct* dans l'exemple *mcf*

4.2.3 Appel de fct

La figure 4.4 montre la mise en place des deux structures markoviennes.

Remarquons les appels à *set_ adress*. Dans cet exemple, le changement de taille du graphe grâce à ces appels est considérable. En effet, sachant que nous travaillons avec une profondeur de 1, nous sommes passé d'un graphe global (somme du nombre de noeuds des deux graphes) de plus de 4000 noeuds à un graphe global de moins de 20 noeuds. Cette diminution de taille entraîne pour notre optimiseur un gain mémoire important mais aussi une légère accélération.

La deuxième remarque est la possibilité de surveiller différentes instructions d'accès mémoire avec la même structure, c'est le cas pour la deuxième séquence. Nous appelons deux fois la fonction *fct* puisque la donnée *tail->mark* est accédée plusieurs fois. Si nous omettions un des appels, l'accélération serait moindre.

Remarquons que rien ne nous oblige à surveiller les deux appels. Dans certains cas, il peut s'avérer intéressant de le faire (comme c'est le cas ici) et dans d'autres cas, il peut être est plus judicieux de ne considérer qu'un seul appel (c'est le cas pour *treeadd*).

Chapitre 5

Performance

Dans ce chapitre, nous présentons les différents résultats obtenus avec notre modèle. Nous avons testé les différents programmes sur trois types de processeurs : un *Pentium III 1.2 Ghz*, un *AMD Athlon XP 2600+* et un *Itanium-2*. Nous présentons brièvement les différents programmes étudiés, puis les temps obtenus sur les différentes architectures.

Afin d'implémenter une stratégie de préchargements, une technique doit être adoptée afin de charger les données à l'avance dans le cache. Une solution serait d'utiliser une instruction *load* lorsque nous avons trouvé l'adresse à précharger. Malheureusement, les instructions *load* nécessitent généralement que l'adresse soit dans l'espace mémoire du programme. Si ce n'est pas le cas, le programme provoque une erreur fatale. Une solution est d'ajouter un test qui vérifie que l'adresse que nous allons précharger est dans cet espace.

Mais vu que nous cherchons à minimiser les calculs de l'optimiseur, un tel test est impensable si nous voulons encore obtenir une accélération. Heureusement, il existe dans la plupart des jeux d'instructions des processeurs modernes une instruction dédiée au préchargement. Cette instruction est généralement dotée d'un test interne qui vérifie si l'adresse demandée est valide. Mais il faut faire attention à leur implémentation :

- Sur le *Pentium III* [17] et sur les processeurs *AMD* [4], l'instruction nommée *prefetch* est juste un conseil pour le processeur et il n'a aucune obligation d'effectuer le préchargement.
- Sur l'*Itanium-2*, l'instruction *lfetch* [18] est une instruction à part entière.

C'est pour cela que nous avons, dans un premier temps effectué nos expériences sur le processeur *Itanium-2*. Nous avons par la suite étendu nos tests aux deux autres architectures.

5.1 Les différents programmes

Nous avons étudié quatre programmes venant de différents bancs d'essais. Le tableau 5.1 montre les différentes provenances, les fichiers d'entrées, le fichier et la fonction que nous avons tentés d'optimiser.

Tous les programmes ont été compilés avec les compilateurs *GNU gcc* et *Intel icc* (sauf sur l'architecture AMD) et en utilisant le niveau d'optimisation 3 (option de compilation -O3). Les temps obtenus sont des moyennes de cinq exécutions.

5.2 Le processeur *Itanium*

Les tests sur le processeur *Itanium-2* ont été réalisés sur la machine parallèle du CECPV (ULP).

Programme (Benchmark)	Fichier d'entrée	Fichier optimisé	Fonction optimisée
Treeadd (Olden [1])	20 noeuds et 1 processeur	node.c	Treeadd
Equake (Spec2000 [3])	ref	quake.c	smvp
ks (Pointer Intensive [2])	KL-5.in	KS-2.c	FindMaxGp-AndSwap
mcf (Spec2000)	ref	implicit.c	price_out_impl

TAB. 5.1 – Les programmes étudiés

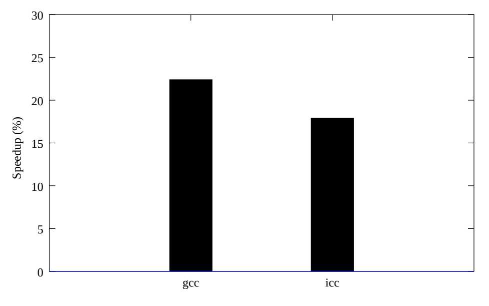

FIG. 5.1 – Accélération du programme *mcf* dépendant du compilateur en utilisant une profondeur de 1 et une distance de prédiction de 4.

5.2.1 Différences entre *gcc* et *icc*

La figure 5.1 montre la différence entre les gains de performance obtenus sur le programme *mcf* en fonction du compilateur utilisé. Le compilateur *gcc* obtient une meilleure accélération. Les optimisations mises en oeuvre sont celles de la section 4.2, la seule fonction modifiée est donc *price_out_impl*.

Mais le compilateur *gcc* n'optimise pas forcément le code de la même façon que le compilateur *icc*. En effet, si le compilateur *icc* optimise certaines portions du code différemment, notre optimiseur peut perdre son intêret dans ces régions du programme. Bien sûr, il est imaginable que pour certains programmes, ce soit l'inverse qui se produise.

Notons que l'accélération obtenue est relative au temps d'exécution global du programme et non au temps de la fonction d'origine. Ceci a une conséquence non négligeable : le programme *mcf* a été optimisé de plus de 20% avec le compilateur *gcc*, mais vu que *price_out_impl* prend la moitié du temps global, la fonction a été accélérée à plus de 40%. Dans la suite, nous utiliserons le compilateur du constructeur pour les processeurs *Pentium III* et *Itanium-2* et le compilateur *gcc* pour le processeur *AMD Athlon*.

5.2.2 Les performances

Programme	Profondeur	Distance
treeadd	1/1/1	1/3/4
ks	1/2/4	3/3/3
equake	2/2/2	1/3/5
mcf	1/1/1	2/4/9

TAB. 5.2 – Les différents paramètres utilisés par la figure 5.2

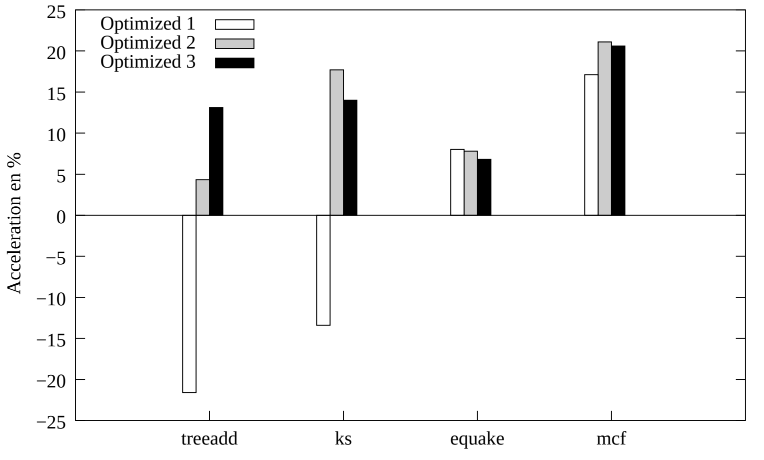

FIG. 5.2 – Accélérations obtenues avec les programmes *treeadd*, *equake*, *ks* et *mcf*. La signification des histogrammes est donnée par le tableau 5.2.

La figure 5.2 et la table 5.2 donnent les différents temps obtenus avec des paramètres différents. Les accélérations obtenues sont relatives au temps d'exécution global du programme et non pas au temps de la fonction d'origine.

La table 5.2 donne les différents paramètres qui ont été utilisés pour chaque programme. Par exemple, si nous regardons les histogrammes de *treeadd*, ils représentent les temps obtenus avec comme paramètres (1,1), (1,3) et (1,4) où la première valeur est la profondeur de construction et la deuxième est la distance de prédiction.

Les paramètres ont été choisis pour montrer certaines caractéristiques de l'optimiseur :

- *treeadd* n'a pas besoin d'une profondeur supérieure à 1 pour obtenir une accélération. Par contre, si la distance est trop faible, nous obtenons un ralentissement non négligeable.
- *ks* a besoin d'une profondeur supérieure à 1 pour être accéléré. Mais en profondeur 4 nous obtenons un moins bon temps. Il faut donc faire attention de ne pas utiliser une profondeur trop importante.
- *equake* obtient une petite accélération par rapport aux autres programmes mais nous montrons la baisse d'accélération en fonction de la distance.
- C'est avec *mcf* que nous obtenons les meilleurs temps. Nous avons tout de même une perte de performance lorsque notre distance de prédiction devient trop importante.

5.3 Le *Pentium III*

Sur le processeur *Pentium III*, nous obtenons des accélérations différentes sur les programmes testés. Ceci n'est pas étonnant, compte tenu du fait que l'architecture du processeur n'est pas la même, les instructions de préchargements diffèrent et la rapidité des calculs n'est pas identique.

Tout de même, nous obtenons avec les mêmes paramètres du tableau 5.2, les histogrammes de la figure 5.3.

Voici quelques remarques sur cette figure :

- Le programme *mcf* conserve sa bonne accélération. On remarque tout de même que le meilleur temps est obtenu avec une distance de préchargement de 2 et non de 4, comme cela a été le cas pour le processeur *Itanium*.
- *treeadd* est ici toujours accéléré, nous obtenons même un meilleur temps lorsque la distance de préchargement est de 1. Remarquons que le temps moyen d'une exécution non optimisée sur le processeur *Itanium* était de 40 secondes, alors que sur le *Pentium III* elle dure 101 secondes. Le temps d'exécution étant plus grand, l'intervalle de temps entre deux accès qui nous intéressent s'élargit

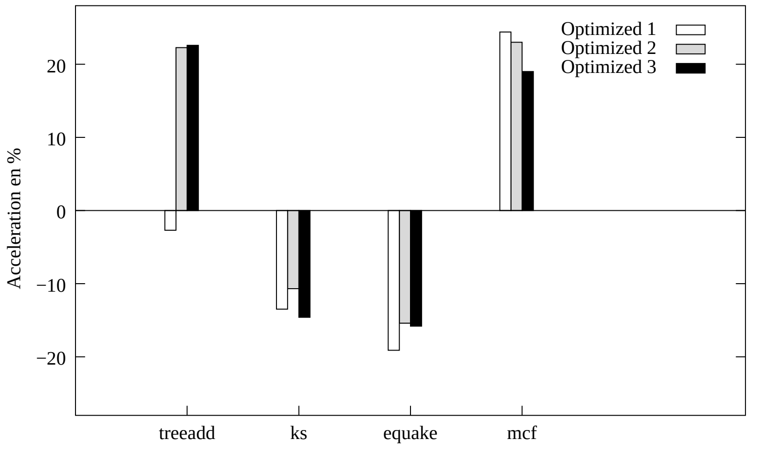

FIG. 5.3 – Accélérations obtenues avec les programmes *treeadd*, *equake*, *ks* et *mcf* sur un Pentium III . La signification des histogrammes est donnée par le tableau 5.2.

aussi. Ceci peut expliquer le fait que précharger trois sauts à l'avance suffit pour obtenir une accélération considérable.

– Malheureusement, *ks* et *equake* ne sont pas optimisés avec ce processeur. Une cause peut être la non prise en compte du "conseil" de préchargement par le processeur. Mais, vu que les tailles de cache sont différentes, nous ne pouvons pas certifier que le même accès soit encore coûteux sur un autre processeur.

En résumé, les résultats obtenus peuvent varier selon l'architecture que nous utilisons. Ceci n'a rien d'anormal, puisque les préchargements tentent de réduire les défauts de cache et, d'un processeur à un autre, les défauts de cache d'un même programme peuvent être différents. Un autre facteur important est le temps d'exécution d'un préchargement. En effet, ce temps n'est pas forcément le même et peut donc influencer le choix des paramètres.

5.4 L'AMD

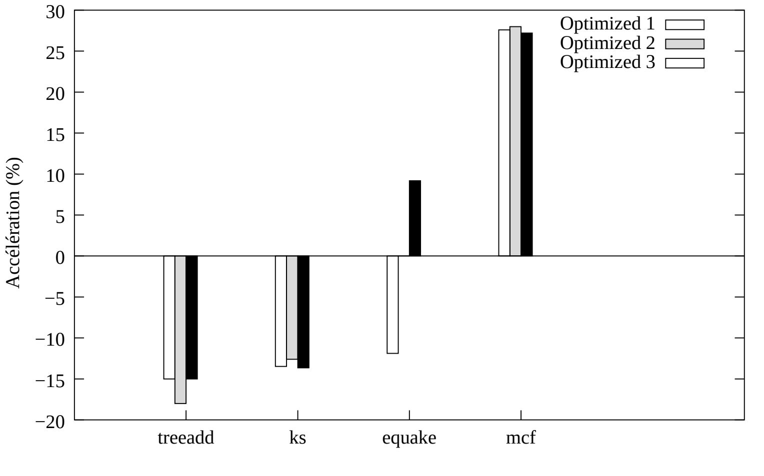

FIG. 5.4 – Accélérations obtenues avec les programmes *treeadd*, *equake*, *ks* et *mcf* sur un Athlon XP 2600+. La signification des histogrammes est donnée par le tableau 5.2.

Les accélérations sont encore différentes lorsqu'on nous exécutons les programmes sur un processeur Athlon. Le programme *mcf* est toujours bien optimisé et obtient même la plus grande accélération par rapport aux trois architectures. Par contre, *ks* et *treeadd* n'ont pas été optimisés avec ce processeur. Enfin, le programme *equake* est accéléré lorsque l'optimiseur précharge cinq sauts en avance.

Ces résultats confirment les différences que peuvent avoir les optimisations avec des préchargements. Mais l'accélération continuelle de *mcf* permet de dire que l'optimiseur est tout de même portable. Il n'optimisera pas tous les programmes sur toutes les architectures mais permettra, dans certains cas, d'accélérer considérablement un programme donné sur un ordinateur spécifique.

Conclusion

Nous avons montré qu'un système d'optimisation dynamique et entièrement logiciel est une technique réaliste pour améliorer les performances d'un programme. Utilisant des mécanismes matériels standards, il est possible de définir un processus dynamique générique dont le coût de maintien est généralement suffisamment bas par rapport à l'accélération globale du programme cible.

Le modèle Markovien basé sur les sauts des accès mémoires, présenté dans ce rapport, donne des améliorations significatives pour des programmes utilisant des données de façon intensive, comme cela a pu être montré sur nos exemples. De plus, ce modèle ne se restreint pas à optimiser des boucles mais peut facilement étudier des accès mémoires inter-procéduraux ou à l'intérieur de fonctions récursives.

Des améliorations pourraient encore être apportées à notre optimiseur. Par exemple, nous pourrions l'étendre en enlevant les parties du graphe qui ne sont pas souvent visitées, afin de réduire l'utilisation mémoire globale et permettre une meilleure accélération. Par contre, puisque la marge de manoeuvre est réduite, un équilibre doit être trouvé entre la stratégie d'optimisation utilisée et le coût de l'optimiseur.

Dans notre modèle, le programmeur doit encore faire sa part du travail pour aider l'optimiseur. Nous avons montré que les résultats dépendent de l'architecture utilisée. Les paramètres choisis pour l'une peuvent ne pas fonctionner pour une autre. C'est donc une des principales raisons pour laquelle un système autonome serait avantageux. Dans un premier temps, la gestion automatique des paramètres serait sûrement envisageable avec des heuristiques et des approximations. Ceci permettrait d'alléger considérablement le travail du programmeur et nous rapprocherait d'un système entièrement automatique et transparent.

Puisque nous voulons rendre le système plus autonome, une priorité est également d'accélérer l'optimiseur. Le temps que nous économiserons permettrait de faire des calculs plus évolués et rendre les prédictions plus précises.

Une des solutions consiste à enlever l'appel de la fonction *fct* et de directement manipuler le code binaire, en insérant le code de l'optimiseur pendant l'exécution. Ceci entraîne de grands problèmes de gestion des registres du code original. Une deuxième solution, moins draconienne, est d'établir une phase de précompilation qui insérerait le code de l'optimiseur au niveau du langage de haut-niveau, laissant au compilateur le soin d'optimiser statiquement le tout.

Si une des deux solutions est concluante, l'optimiseur pourrait tenter de déterminer quels accès provoquent les défauts de cache automatiquement. Si ceci est accompli, le modèle deviendrait un véritable système d'optimisation dynamique automatique.

Nos perspectives de développements futurs sont d'étudier diverses stratégies d'optimisation pour les préchargements, mais aussi des techniques plus globales, comme par exemple un optimiseur de la localité spatiale des données ou un générateur dynamique d'indications pour le cache. Le but final est de créer une interface dynamique et modulable à partir de laquelle plusieurs stratégies peuvent s'imbriquer dépendant des accès auquels on s'intéresse. Un système de détection de phase et paramétrage automatique guiderait efficacement les diverses stratégies d'optimisation.

Une optimisation dynamique ne pourra jamais être aussi efficace qu'une optimisation statique en ce qui concerne les structures de contrôles et de données statiques. C'est pour cela que nous ne nous intéressons qu'aux contrôles et aux accès mémoires dépendant de résultats en cours d'exécution pour lesquels l'optimisation dynamique est une réponse appropriée.

Bibliographie

[1] Olden benchmark. http ://www.cs.princeton.edu/ mcc/olden.html.

[2] Pointer intensive benchmark. http ://www.cs.wisc.edu/ãustin/ptr-dist.html.

[3] Spec2000 benchmark. http ://www.spec.org.

[4] AMD. Amd extensions to the 3dnow ! and mmx instruction sets manual. http ://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/22466.pdf, March 2000.

[5] M. Annavaram, J. M. Patel, and E. S. Davidson. Data prefetching by dependence graph precomputation. In *Proceedings of the 28th annual international symposium on Computer architecture*, pages 52–61. ACM Press, 2001.

[6] J.-L. Baer and T.-F. Chen. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput.*, 44(5) :609–623, 1995.

[7] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo : a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5) :1–12, 2000.

[8] K. Beyls and E. D'Hollander. Compile-time cache hint generation for EPIC architectures. In *Proceedings of the 2nd workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Techniques*, 2002.

[9] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *1st International Symposium on Code Generation and Optimization*, pages 265–276, 2003.

[10] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 199–209. ACM Press, 2002.

[11] J. Collins, S. Sair, B. Calder, and D. M. Tullsen. Pointer cache assisted prefetching. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 62–73. IEEE Computer Society Press, 2002.

[12] M. Crochemore and M.-F. Sagot. *Handbook of Computational Chemistry*, chapter Motifs in sequences : localization and extraction. Marcel Dekker Inc., 2004.

[13] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. Deli : a new run-time control point. In *35th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 257–268, December 2002.

[14] Hewlett-Packard. Perfmon kernel interface. http ://www.hpl.hp.com/research/linux/perfmon/perfmon.php4.

[15] Intel. Hyper-threading technology. http ://www.intel.com/technology/hyperthread/.

[16] Intel. Vtune performance analysers. http ://www.intel.com/software/products/vtune/.

[17] Intel. Intel architecture optimization (reference manual). http ://www.intel.com/design/pentiumii/manuals/245127.htm, 2000.

[18] Intel(R). *Itanium(R) 2 Processor Reference Manual for Software Development and Optimization*, chapter Optimal use of lfetch, pages 71–72.

[19] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *IEEE Transactions on Computers, Vol. 48, NO. 2*, pages 121– 133, Febuary 1999.

[20] D. Kim, S. Liao, P. Wang, J. del Cuvillo, X. Tian, X. Zou, D. Yeung, M. Girkar, and J. Shen. Physical experimentation with prefetching helper threads on intel's hyper-threaded processors. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization*, pages 27–38, 2004.

[21] H.-S. Kim and J. E. Smith. Dynamic software trace caching.

[22] J. Kim, K. V. Palem, and W.-F. Wong. A framework for data prefetching using off-line training of markovian predictors. In *20th International Conference on Computer Design (ICCD 2002)*, pages 340–347, 2002.

[23] J. Lu, H. Chen, R. Fu, W. Hsu, B. Othmer, P. Yew, and D. Chen. The performance of runtime data cache prefetching in a dynamic optimization system. In *36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 180–190, December 2003.

[24] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *28th annual international symposium on Computer architecture*, pages 40–51, 2001.

[25] C. G. Nevill-Manning. *Inferring Sequential Structure*. PhD thesis, University of Waikato, Nouvelle Zélande, 1996.

[26] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 115–126. ACM Press, 1998.

[27] A. Srivastava, A. Edwards, and H. Vo. Vulcan : Binary Transformation in a Distributed Environment. Technical Report MSR-TR-2001-50, 2001.

Annexe

Article *"ESODYP : An Entirely Software and Dynamic Data Prefetcher based on a Memory Strides Markov Model"*

soumis au

37th ANNUAL IEEE/ACM INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE