École Doctorale Mathématiques, Sciences de l'Information et de l'ingénieur

ULP-INSA-ENGEES

THÈSE

présentée pour obtenir le grade de

Docteur de l'Université Louis Pasteur-Strasbourg I
Spécialité : Informatique

par

Jean Christophe Beyler

OPTIMISATION DYNAMIQUE ET LOGICIELLE DES ACCES AUX DONNEES : Réduction de la latence mémoire par analyses et transformations binaires dynamiques à faible coût

Soutenue publiquement le 13 décembre 2007

Membres du jury

Directeur de thèse :	M. Philippe Clauss (Professeur, ULP, Strasbourg)
Rapporteur interne :	Mme. Dominique Bechmann (Professeure, ULP, Strasbourg)
Rapporteur externe :	Mme. Christine Eisenbeis (Directrice de Recherche, INRIA Futurs, Orsay)
Rapporteur externe :	M. François Bodin (Professeur, IRISA, Rennes)

Laboratoire des Sciences de l'Image, de l'Informatique et de la Télédétection (UMR ULP-CNRS 7005)

Contents

List of Figures

List of Tables

Listings

List of Algorithms

Koan Zen - *La voie est celle sous vos pieds*

Bash.org - *<geckosenator> my new approach to optimizing is comment random lines of code out and check speed and verify the results are still correct*

Thanks/Remerciements

Cela fait presque un mois que je suis docteur et je prends le temps, en ce 8 janvier 2008, d'écrire ces deux pages de remerciements. Il est important que chaque personne qui lit ces lignes comprenne grâce à qui tout cela a été possible.

Tout d'abord, un grand merci à Philippe Clauss pour avoir cru en moi pendant toutes ces années. Merci d'avoir lancé un tel sujet en l'air, cela n'a pas toujours été rose mais nous nous en sommes sortis à la fin ! Ce qui est sûr, c'est que sans son aide et sans ses conseils, je ne serais probablement pas arrivé au bout de cette thèse. Merci pour tout ! Par contre, j'attends toujours mon câble de téléphone mais je suis patient et cela me donnera une excuse pour t'embêter périodiquement.

Merci à l'ICPS pour la bonne humeur et de m'avoir supporté pendant toutes ces années. Merci à Alain et Romaric pour leur bonne humeur et leurs blagues. Merci à Vincent pour son aide précieuse, ses conseils et n'oublions pas les parties de backgammon ! Merci à Eric pour les discussions de tous les jours qui permettaient de changer les idées et de faire des pauses parfois bien méritées ! Merci à Guillaume, bien que loin pour le moment, d'avoir montré à Técla et à moi les secrets du Mont-d'Or. Merci à Stéphane Genaud pour sa présence lors des pauses estivales, sans toi nous manquions souvent d'un quatrième. Un grand merci à Catherine pour son aide lors des problèmes administratifs et pour ses conseils tout au long de mes études. Ensuite, merci à Rachid qui est déjà parti de l'ICPS mais le restera à jamais et à Benoît Meister qui se trouve à Enouaillecy pour leur soutien et surtout pour ne pas avoir été trop vindicatifs sur cette fin de thèse ! Merci à Stéphane Marchesin, le lapinou officiel de l'ICPS, qu'il saute vers un terrain qui lui correspond. Un grand merci à Choopan pour sa bonne humeur et sa présence sur le terrain de basket et à Olivier pour ses critiques totalement subjectives, pour avoir compris que _ ne doit pas se trouver n'importe où dans un code et pour avoir été là quand il le fallait. Merci à Sarah qui n'est pas un machin, merci à toi d'être simplement toi, ne change jamais !

Un grand merci à Ben sans qui les années d'études n'auraient pas été pareilles. Je pourrais faire une longue liste et sûrement cela serait mieux mais je préfère simplement dire : merci pour tout mon ami. Merci à JB pour sa bonne humeur, ses délires et son amour pour la bonne musique *I'm trying to rob a ****ing tune man !*. Un grand merci à Thor pour cette année de médecine, s'en souvenir permet toujours de passer de bons moments : *Bon, on va chercher du pain pour ces hot-dogs ?...*

Thanks to Michael Klemm to whom I dedicate this paragraph in english. It is on a winter day at CPC that we met and started to talk about Esodyp in Jackal and almost two years later we made it. Let's continue this road together, I'm sure there is more to do ! May you finish your thesis and get everything you deserve !

Bien sûr, j'aimerais prendre le temps de remercier ma mère et mon père pour leur soutien continuel à travers toutes les étapes de ma vie mais aussi d'avoir été là pour ma soutenance ! Merci pour votre aide, je ne sais pas comment j'aurais réussi sans vous ! Merci à Stéphanie pour être la soeur qui rattrape mes bourdes sans broncher ;-). Merci à Marie Anne qui, bien que loin physiquement, est toujours là lorsque j'en ai besoin.

Enfin, et bien qu'elle m'en voudra de l'avoir gardée pour la fin, j'aimerais remercier Técla, ma belle femme qui a toujours été là pour moi et a su me motiver quand il le fallait. Et lorsqu'il le fallait, elle a su changer le sujet de la discussion et me faire rire. Grâce à elle, des crocodiles imaginaires prenaient forme ou de simples balades dans la rue devenaient naturellement magiques. Merci à toi.

Résumé en français

1 Introduction

Les tâches relevant du processus de compilation tendent à se complexifier et à se multiplier. D'un simple rôle de traducteur de texte en langage de programmation évolué à un code binaire exécutable par une machine garantissant les exigences fonctionnelles, il est désormais demandé au compilateur de remplir plusieurs objectifs non-fonctionnels, soit exclusivement ou simultanément. Le code binaire produit devra par exemple :

- provoquer une durée d'exécution la plus courte possible ;
- respecter des contraintes temps-réel de temps d'exécution et d'ordonnancement ;
- minimiser sa consommation électrique ;
- minimiser sa consommation mémoire ;
- avoir un comportement prévisible à l'exécution.

De plus, tous ces objectifs sont souvent contradictoires ou possèdent de fortes interactions qui risquent d'annuler, voire de détériorer, leurs effets. Il est par exemple clair que l'objectif de rapidité d'exécution peut être en contradiction avec l'objectif de consommation électrique minimale, ou avec l'objectif de comportement prévisible. Les moyens employés pour les atteindre peuvent être variés et leur choix peut être déterminant quant à ces contradictions.

Tous ces aspects relèvent d'une maîtrise globale du comportement du couple matériel/logiciel. Un logiciel, même conçu, analysé, prouvé ou simulé par les moyens les plus avancés, aura presque toujours un comportement inattendu lorsqu'il sera confronté au matériel servant à l'exécuter. Ainsi, les approches dites *statiques* d'analyse et de transformation de programmes sont limitées par l'ignorance des effets de cette confrontation. Il en est de même si l'on se place du côté du matériel, parfois conçu sans réalisme par rapport aux logiciels susceptibles d'y être exécutés. Mais nous nous plaçons dans cette thèse du côté du logiciel, confronté à un matériel figé.

Une réponse à cette problématique de maîtrise du comportement consiste à opposer, ou à compléter, les approches statiques avec des approches dites *dynamiques*, c'est-à-dire où l'analyse et la transformation de programmes s'opèrent en cours d'exécution. L'analyseur ou l'optimiseur, que l'on peut appeler d'une manière plus générale *compilateur*, dispose alors des informations non fonctionnelles uniquement accessibles à l'exécution, liées à la confrontation logiciel/matériel, mais également à l'évolution fonctionnelle non prévisible du traitement.

Mais l'introduction d'un tel processus dynamique de compilation, en plus du logiciel lui-même, pose des problèmes difficiles de conception afin que ce processus ne soit pas lui-même en contradiction avec les objectifs recherchés. Il est par exemple évident que l'ajout de code de compilation s'exécutant en même temps que le programme cible pose un problème de temps d'exécution plus important, si l'objectif est la minimisation de ce temps d'exécution. Il en est de même pour tous les autres objectifs cités plus haut. Pourtant, les nombreux travaux du domaine de la compilation dynamique, dont ceux présentés dans ce manuscrit, montrent qu'il est tout à fait possible d'atteindre ces objectifs en compensant largement les effets négatifs par les effets positifs du processus dynamique. Par exemple, le surplus de consommation mémoire du à ce processus pourra être négligeable par rapport à la réduction de consommation obtenue grâce à lui.

Cette thèse se place dans ce cadre de développement d'approches dynamiques permettant une maîtrise du comportement. Plus particulièrement, les travaux présentés ici recouvrent l'objectif principal de minimisation des temps d'exécution sur une architecture mono ou multi-processeurs, par anticipation des accès mémoire des programmes via le préchargement des données utiles, et ce de manière entièrement transparente à l'utilisateur. Les processus développés comprennent une phase d'analyse dynamique du comportement, où les latences des accès mémoires sont mesurées, une phase de transformations dynamiques optimisantes si celles-ci ont été jugées utiles, où des instructions de préchargement de données sont insérées dans le code binaire, une phase d'analyse dynamique des effets de ces optimisations, et une phase d'annulation de transformations si celles-ci ont été observées comme inefficaces. De plus, toutes ces phases s'appliquent individuellement à chaque accès mémoire, et éventuellement de manière répétée lorsque ces accès présentent des comportements variables au cours de l'exécution du logiciel cible.

Nous montrons qu'il est possible de concevoir un tel système dynamique d'une relative complexité et entièrement logiciel, c'est-à-dire qui ne repose sur aucune fonctionnalité spécifique de la machine d'exécution, qui est efficace pour de nombreux programmes et très peu pénalisant pour les autres. A notre connaissance, notre travail constitue une première proposition d'un système dynamique d'optimisation entièrement logiciel qui ne se base pas sur une interprétation du code binaire.

2 Un optimiseur dynamique et automatique à faible coût

Le chapitre 2 commence par présenter un optimiseur dynamique qui instrumente et optimise un programme lors de son exécution. Il y parvient en modifiant directement le code binaire en fonction du comportement mémoire du programme. En calculant le coût moyen d'un chargement instrumenté, le système décide ou non de l'optimiser ou de le désinstrumenter. Dans ce dernier cas, le programme exécute alors le chargement sans instrumentation, ne ralentissant donc plus le programme.

L'optimisation que le système utilise est un prédicteur de pas qui émet des préchargements. Un tel prédicteur calcule le pas mémoire entre deux accès consécutifs et le multiplie par un facteur, appelé *distance de préchargement*, afin de prédire les prochains éléments. Cette distance de préchargement représente le nombre d'accès entre l'élément courant et celui prédit. Le préchargement de données est une solution pour limiter les défauts de cache. Ces derniers se produisent lorsque le programme a besoin d'une donnée qui n'est pas encore dans un des niveaux de cache. Le processeur arrête ainsi l'exécution du programme et charge la donnée depuis la mémoire centrale. Les préchargements permettent de faire des demandes en avance pour que, lorsque le programme requiert les données, elles soient déjà en cache.

Le travail présenté dans ce chapitre a été publié dans [12] et une version revue est en cours de soumission [11].

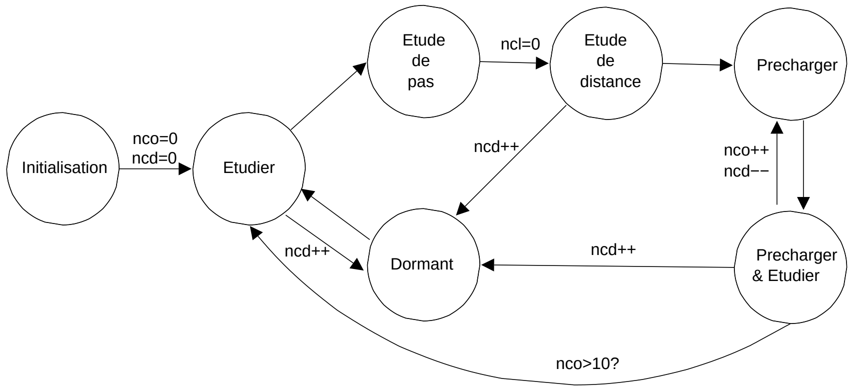


Figure 1: L'automate fini représentant les états et transitions des chargements

2.1 Sélection de chargement

Lorsqu'un système logiciel veut émettre des préchargements, il doit d'abord choisir les zones de code qui sont importantes pour un programme et ensuite comprendre le comportement mémoire afin de prédire correctement. Au lancement d'un programme, le système, parcourt le code et transforme chaque chargement en un saut inconditionnel vers une version instrumentée. Celle-ci met à jour un compteur d'exécutions du chargement et accumule le nombre de cycles processeur utilisés pour l'effectuer. Lorsque le système divise ces deux valeurs, un nombre moyen de cycles, ou *latence moyenne*, est obtenu.

Le système vérifie périodiquement l'état des chargements et, selon les différents compteurs associés, décide d'optimiser ou non les chargements. Afin de pouvoir gérer les différents chargements, un état est associé à chacun d'entre eux. La figure 1 présente un automate qui gére les différents états possibles.

Les chargements sont initialisés dans l'état *Initialisation*. Une fois qu'ils ont été instrumentés, ils se retrouvent dans l'état *Étudier*. Si le nombre d'exécutions du chargement dépasse un certain seuil, la latence moyenne du chargement est calculée. Si elle est en-dessous d'une valeur donnée, le chargement est considéré comme non coûteux. Il est alors désinstrumenté et mis dans l'état *Dormant*. Par contre, si elle est au-dessus du seuil, il devient candidat à l'optimisation.

L'optimisation que nous avons implémentée est un prédicteur de pas. Il faut donc calculer la différence entre deux éléments avant de pouvoir prédire correctement. Premièrement, le pas est calculé dans l'état *Étude de Pas*. Dans cet état, le code calcule la différence entre les adresses de deux accès consécutifs afin d'avoir le pas courant. Ensuite, le chargement est transféré vers l'état *Etude de distance* où la distance de préchargement est calculée. Une distance de préchargement représente le nombre d'accès entre l'accès courant et celui prédit. Si elle est trop faible, la donnée prédite ne sera pas encore dans le cache de données lorsque le programme en aura besoin. Par contre, si la valeur est trop importante, la donnée risque d'être écrasée par une autre qui nécessitait la même ligne de cache.

Nous pouvons remarquer que cette étude est donc empirique. Dans le cas où le comportement mémoire semble aléatoire, le système va remarquer que le pas choisi ne donne pas de bons résultats et finira par désinstrumenter le chargement et l'envoyer dans l'état *Dormant*. D'autre part, si le comportement mémoire posséde un pas constant, le système le trouvera et pourra optimiser le chargement.

Lorsqu'un pas et une distance sont choisis, le système envoie le chargement dans l'état

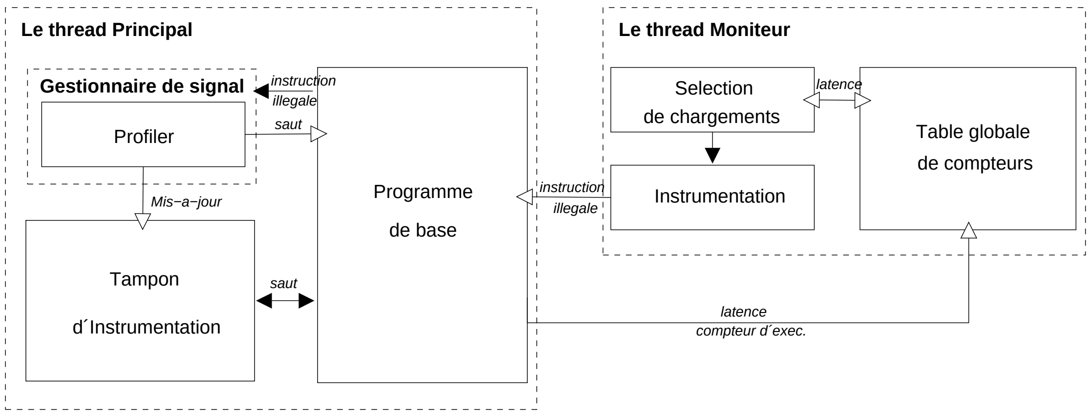


Figure 2: Architecture générale du système

Précharger. Dans cet état, aucune instrumentation autre que l'instruction de préchargement ne demeure afin d'optimiser le code au maximum. Par contre, périodiquement, le code est envoyé vers une version instrumentée, nommée *Précharger et Étudier*, pour vérifier si l'optimisation est toujours bénéfique. Ceci est testé en comparant le coût moyen actuel du chargement avec celui calculé lors de sa présence dans l'état *Étudier*. Si ce n'est plus le cas, le chargement est renvoyé dans l'état *Dormant*.

Afin de savoir quand ré-instrumenter un chargement ou tester l'optimisation mise en place, des niveaux de confiance ont été implémentés. Le niveau de confiance dormant *ncd* détermine le temps où le chargement demeure dans l'état *Dormant*. Plus il est élevé et plus le chargement restera sans instrumentation. Ceci permet de ne pas ré-instrumenter trop souvent un chargement qui n'est jamais coûteux ou pour qui l'optimiseur n'arrive pas à trouver une bonne optimisation. Partant de la même idée, un niveau de confiance d'optimisation *nco* permet de ne pas tester trop souvent un préchargement que le système considère comme vraiment bénéfique au programme.

Finalement, il peut arriver que la latence moyenne du chargement non-instrumenté ait changé. Dans ce cas, le système doit la ré-évaluer de temps en temps. Ceci est fait en testant la valeur de *nco*. Si elle est trop élevée, le chargement associé est renvoyé vers l'état *Étudier* et les compteurs sont réinitialisés.

2.2 Système général

Afin de pouvoir gérer et maintenir les états des différents chargements, le système utilise un deuxième *thread* (fil d'exécution). La figure 2 présente le système d'un point de vue général. Sur la gauche se trouve le programme instrumenté de base. En utilisant des sauts inconditionnels entre le programme et la zone d'instrumentation, le programme incrémente et met à jour des compteurs dans un tableau que le deuxième thread peut étudier.

Le deuxième thread, représenté par la partie de droite de la figure, se réveille périodiquement et vérifie le tableau de compteurs. Par exemple, lorsque le nombre d'exécutions d'un chargement dépasse un certain seuil, le système calcule la latence moyenne de ce chargement. Si elle dépasse une certaine valeur, le chargement est considéré comme coûteux et l'optimiseur tentera de prédire les prochains accès afin de minimiser son coût.

Pour effectuer les transitions d'état des chargements, il faut pouvoir modifier le code instrumenté associé. Or, pour le faire sans provoquer des modifications sémantiques à un programme,

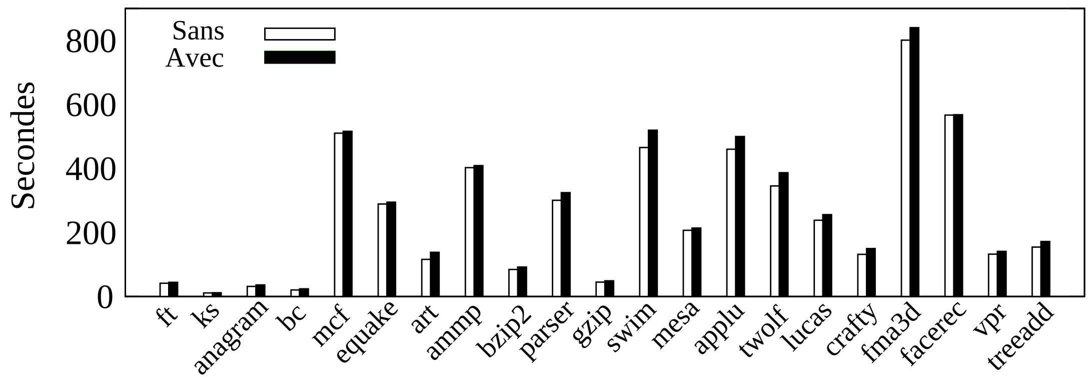


Figure 3: Le coût du système avec le compilateur `gcc`. Les histogrammes montrent les temps d'exécution des programmes avec et sans l'utilisation du système

le système doit prendre quelques précautions. Nous savons qu'un chargement est instrumenté en utilisant un saut inconditionnel vers la zone instrumentée. Le système commence donc par remplacer le saut par une instruction illégale. Ceci permet d'arrêter le programme lorsqu'il tente d'exécuter une des instructions illégales. Lorsqu'un programme exécute une instruction illégale, un signal est émis qui provoque normalement la fin de son exécution. Un gestionnaire de signal peut être mis en place pour tenter de réparer le problème. Notre système utilise ce mécanisme et, lors de cet appel, l'état du chargement et son code instrumenté sont mis à jour avant de remettre le saut inconditionnel en place et de reprendre l'exécution du programme.

2.3 Coût du système

Il est difficile de pouvoir évaluer précisément le coût d'un système dynamique puisque ceci demande de modifier le comportement interne. En perturbant le système, les résultats obtenus ne sont plus les mêmes par rapport à une exécution *normale*. Par contre, cela peut donner une idée qualitative du coût et c'est dans ce but que nous présentons la figure 3. En supprimant les instructions de préchargement, le système ne peut plus optimiser le programme mais calcule toujours la latence moyenne des chargements, trouve un pas et choisit la meilleure distance de préchargement. Comme le montre la figure, l'exécution de la version de base, sans instrumentation, est presque identique à celle avec l'instrumentation. Le coût moyen du système proposé est de 5%.

2.4 Performance

La figure 4 présente les résultats obtenus lorsque le système insére des instructions de préchargement sur différents programmes provenant de bancs d'essai connus. Cette figure compare les performances obtenues entre le compilateur *gcc* et le compilateur *icc* utilisant le niveau d'optimisation `O3`. Certains programmes montrent un réel gain de performance comme les programmes `ft`, `mcf`, `equake`, `art` et `treeadd`. Le pire cas est le programme `twolf`. Ce programme répond mal à la solution adoptée pour allouer des registres dynamiquement pour l'exécution des codes instrumentés. De plus, le système est incapable de trouver un chargement qui posséde un comportement mémoire répondant favorablement à l'optimisation proposée. Donc, pour ce programme, le système est incapable de l'accélérer et le coût initial est trop important.

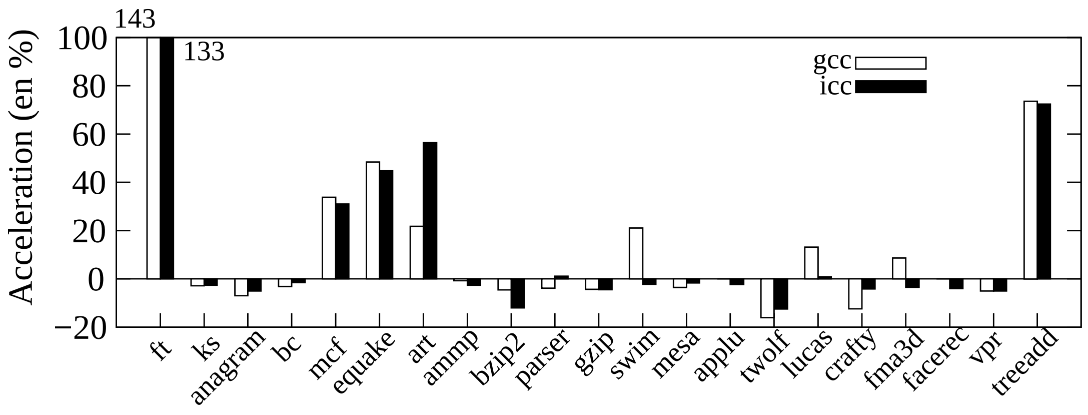


Figure 4: Les accélérations obtenues en pourcentage par notre système sur des codes générés par les compilateurs `gcc` et `icc`

2.5 Conclusion

Le système présenté permet d'instrumenter dynamiquement, avec seulement 5% de coût en moyenne, des programmes cibles. En modifiant le code binaire pour étudier et optimiser les accès mémoire, il est capable d'accélérer certains programmes testés de 2% à 143%.

Le reste du chapitre 2 présente les détails d'implémentation du système et propose une étude sur la viabilité d'une telle méthode sur une vingtaine de programmes issus de bancs d'essai. Par exemple, le système peut être étendu pour n'instrumenter que les chargements souvent utilisés et ce chapitre propose une solution dans la section 2.2.3 page 61.

3 Esodyp

Le système que nous venons de présenter montre qu'il est possible, sans connaissances préalables du programme cible, de modifier dynamiquement le code binaire d'un programme afin de l'accélérer. L'étude du comportement mémoire étant un facteur important pour la sélection des chargements, la suite de cette thèse présente la création d'un modèle Markovien permettant de mieux comprendre et prédire des accès mémoire.

Le modèle Markovien dont nous avons besoin doit pouvoir se construire rapidement et aussi pouvoir se mettre continuellement à jour. Puisque la construction d'un tel modèle est relativement coûteuse, il est devenu opportun de diviser le modèle en deux. D'abord, les premiers éléments de la séquence contribuent à la construction du modèle. Puis le modèle est figé et est utilisé tant que les prédictions qui lui sont dues sont correctes.

Le travail présenté dans ce chapitre a été publié dans [9].

3.1 La définition

Un modèle Markovien se base sur des accès précédents afin de calculer les futurs accès. La définition mathématique peut revenir à la formule suivante :

$$Pr(X_{n+1} = x | X_n = x_n, ..., X_1 = x_1) = Pr(X_{n+1} = x | X_n = x_n)$$

Cette formule montre la définition du calcul de probabilité Markovien. Si on considère les accès mémoire comme étant une suite numérique, la probabilité que le $(n + 1)^{ème}$ élément soit

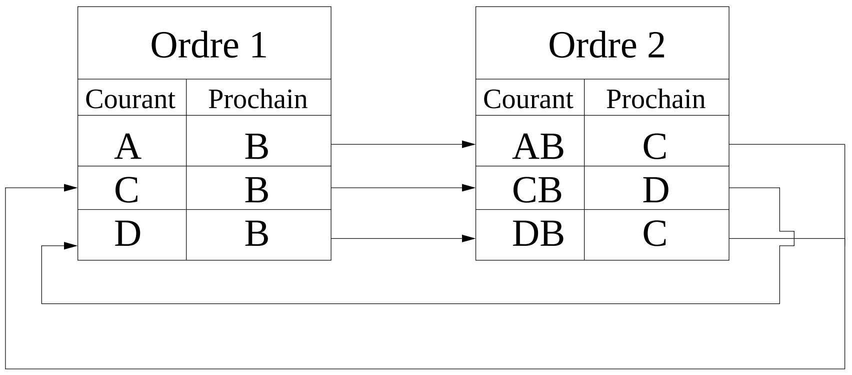


Figure 5: Représentation sous forme de table optimisée, chaque ligne posséde l'élément courant et le prochain élément aperçu dans la séquence

x ne dépend que du $n^{ème}$ terme. Cela se traduit par le fait qu'il suffit de connaître le dernier accès pour calculer la probabilité du suivant.

Par contre, il est possible que, pour avoir un modèle plus précis, il soit nécessaire de calculer les prédictions en prenant en compte plusieurs éléments précédents. La formule généralisée est donc :

$$Pr(X_{n+1} = x | X_n = x_n, ..., X_1 = x_1) = Pr(X_{n+1} = x | X_n = x_n, X_{n-1} = x_{n-1}, ..., X_{n-m+1} = x_{n-m+1})$$

En effet, la valeur de la probabilité du $(n+1)^{ème}$ terme dépend à présent des termes $x_n, x_{n-1}, ..., x_{n-m}$. On appelle m l'ordre du modèle et il représente le nombre d'éléments du passé utilisés pour calculer le prochain élément.

3.2 Structures de données

Il existe différentes solutions pour représenter un modèle Markovien d'un point de vue informatique. Nous présentons dans ce chapitre deux possibilités : un tableau ou un graphe. Le modèle que nous voulons implémenter doit pouvoir se construire suite à une série d'accès mémoire et ensuite produire des prédictions. A chaque appel, il faut donc que le modèle soit mis à jour. Il est évident qu'une simple représentation sous forme de tableau demanderait trop de temps lors de la recherche de la ligne correspondant au comportement mémoire actuel.

La figure 5 montre une technique afin d'accélérer le procédé de recherche. Les deux tableaux représentent la création d'un modèle d'ordre 2. Cela veut dire que les deux derniers éléments sont utilisés pour le calcul des prédictions. Chaque table contient deux colonnes : une pour les éléments précédents et une pour la prédiction. Dans ce cas, la séquence $ABCBDBCBD$ a été utilisée. Le modèle est divisé en deux parties, une partie où l'ordre 1 suffit et une table annexe qui permet de déterminer les cas où les deux derniers éléments sont nécessaires afin de pouvoir prédire correctement. Par exemple, on remarque que, dans cette séquence, après tout élément A se trouve un B. Aussi, nous savons qu'après chaque C et chaque D se trouve aussi un B. Donc dans la table de gauche, nous trouvons ces trois informations.

Par contre, après un élément B se trouve soit un C, soit un D. Afin de distinguer les deux cas, on doit connaître l'élément qui précédait le B. La table de droite permet de faire cela. Par exemple, après la séquence CB se trouve toujours un D. Finalement, avec deux tableaux distincts, on est capable de prédire correctement le comportement de cette séquence.

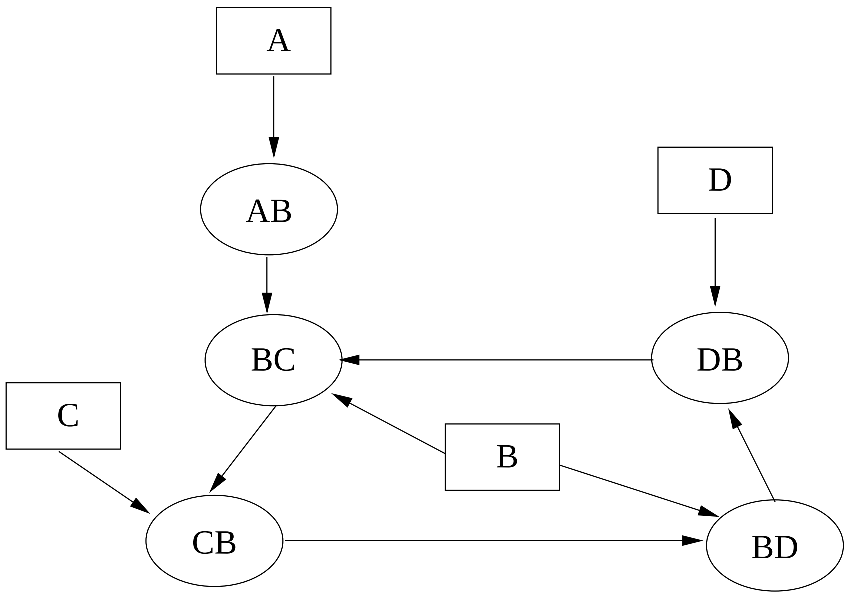


Figure 6: Le graphe d'ordre 2 incluant le premier ordre. Les noeuds du premier ordre sont représentés par des boîtes

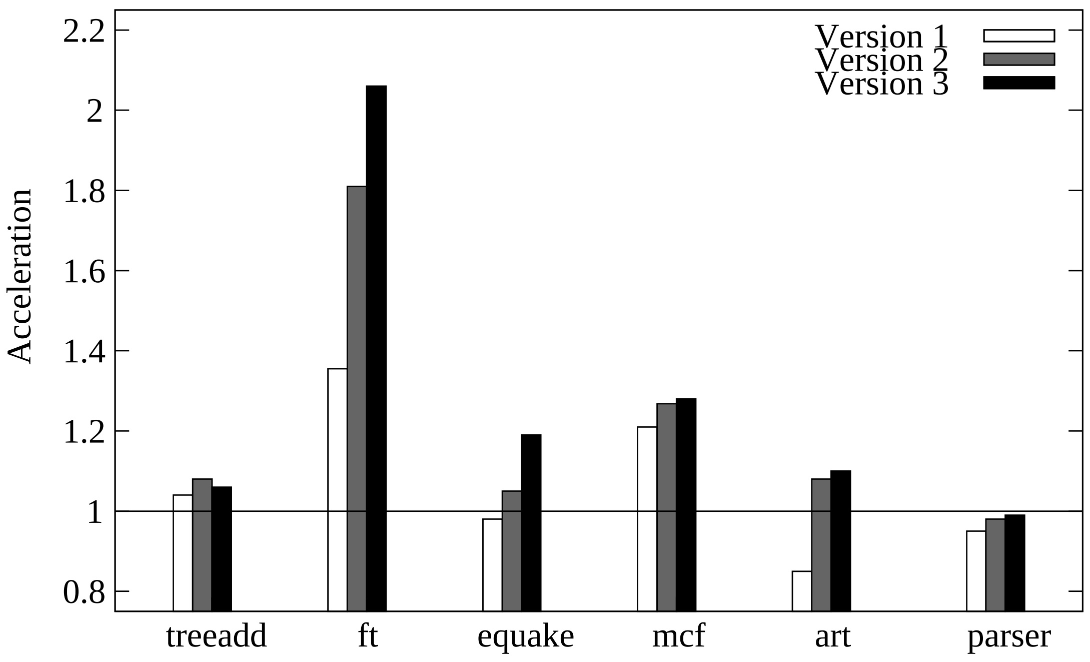
A présent, nous voulons pouvoir prédire le plus rapidement possible et une recherche à travers les tableaux à chaque appel au modèle est trop coûteuse. La solution que nous pourrions adopter est d'ajouter des liens entre les cellules des tableaux afin de trouver directement la prochaine cellule que le modèle utilisera **si** le système prédit correctement. Par exemple, si le modèle se trouve dans la ligne A du tableau de gauche, il prédira que B est le prochain élément. Si c'est le cas, nous savons que les deux derniers éléments sont AB et nous avons une ligne correspondante dans le tableau de droite. Le lien entre ces deux lignes permet donc de se déplacer directement vers la ligne correspondante du tableau de droite afin de ne plus avoir besoin de faire de recherches lors du prochain appel si le modèle a prédit correctement. Par contre, lorsque le modèle se trompe, une recherche coûteuse est toujours nécessaire.

Si nous regardons cette structure de données, nous remarquons qu'elle ressemble presque à un graphe. Il suffit d'ailleurs de séparer les lignes de chaque tableau pour l'obtenir. Figure 6 montre le graphe obtenu avec la même séquence. Nous remarquons que les liens des tableaux précédents sont devenus les arcs du graphe. C'est donc avec la représentation de ce graphe que nous allons implémenter le modèle. Les différents algorithmes de créations et de mises-à-jour du graphe sont présentés dans le chapitre 3.

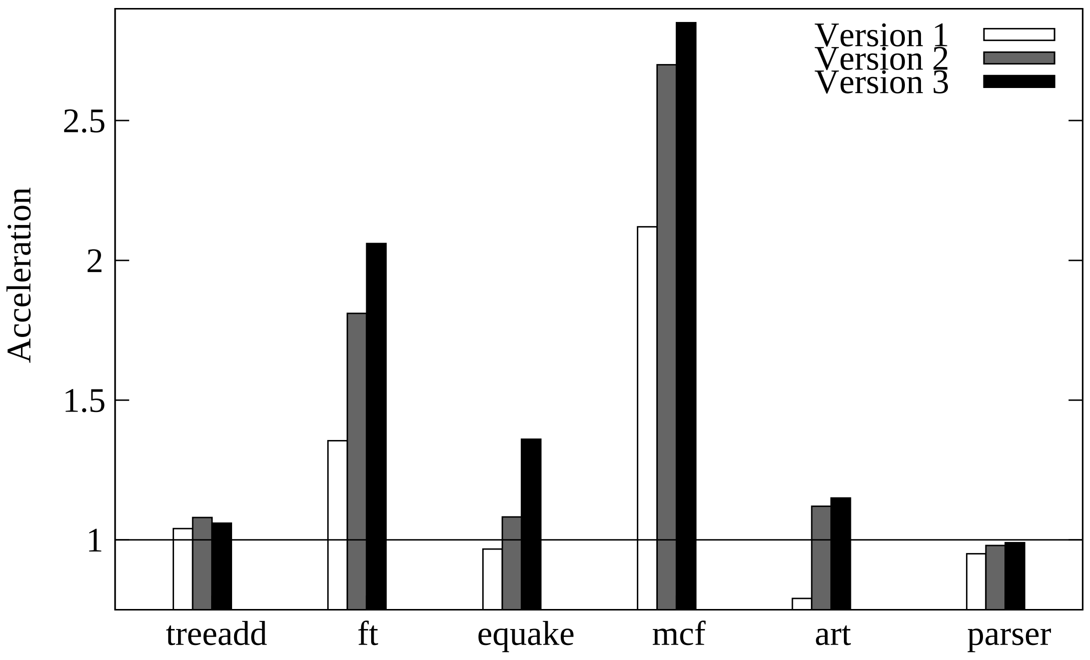
3.3 Performances

L'implémentation du modèle Esodyp a mené à l'écriture d'un certain nombre de fonctions qu'un programmeur peut intégrer dans son code afin de modéliser des comportements mémoire, les prédire et précharger des données. La figure 7 présente les résultats obtenus lorsque nous avons testé le modèle sur différents programmes. La figure de gauche présente les performances obtenues lorsque nous calculons l'accélération par rapport au temps d'exécution global du programme.

Puisque ces programmes ont été, par rapport au système présenté dans le chapitre 2, instrumentés à la main, une seule fonction est généralement modifiée pour contenir les appels vers le



(a) Optimisation des programmes



(b) Optimisation par rapport aux fonctions

Figure 7: Accélérations obtenues sur différents programmes

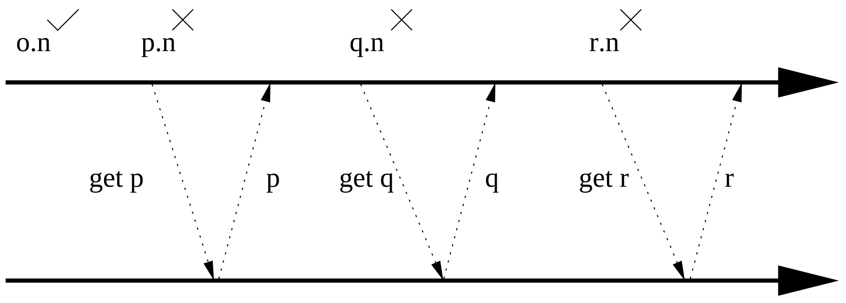


Figure 8: L'exécution du système de requêtes d'objets de Jackal

modèle. La figure de droite montre donc l'accélération obtenue par rapport au temps d'exécution de la fonction modifiée. Finalement, les trois versions représentent différents paramètres utilisés par le modèle et sont présentés en détail dans le chapitre.

3.4 Conclusion

Le modèle Markovien que nous avons présenté dans ce chapitre permet de créer dynamiquement un modèle et de s'en servir afin de prédire les futurs accès mémoire. Une comparaison entre deux représentations structurelles est faite entre un tableau et un graphe. Nous avons choisi la forme du graphe puisqu'elle représente la meilleure solution en terme d'utilisation mémoire, de temps de calculs et de recherche d'une prédiction.

En utilisant un certain nombre de fonctions, le programmeur peut insérer dans le code des appels afin de modéliser les accès et précharger les données pour limiter les défauts de cache. Enfin, sur plusieurs programmes, nous avons montré qu'il est possible d'obtenir des accélérations.

4 Jackal

La construction et la mise à jour du modèle Esodyp sont relativement coûteuses pour réduire la latence mémoire sur un processeur moderne. Par contre, dans le domaine de la programmation distribuée, le coût des messages est plus important. Le temps nécessaire pour accomplir les calculs nécessaires au bon fonctionnement du modèle est largement compensé.

4.1 Intégration du modèle Esodyp dans Jackal

Le chapitre 4 propose d'intégrer Esodyp dans un système de mémoire distribué nommé *Jackal*. La première partie des travaux présentés dans ce chapitre a été publiée dans [64] et un article présentant la deuxième partie est en cours de soumission [13]. Le système Jackal provient du département informatique de l'université d'Erlangen-Nuremberg et le travail contenu dans ce chapitre a été fait en collaboration de Michael Klemm. Le système utilise un compilateur du même nom pour transformer du code source Java en exécutable binaire. En ajoutant des vérifications d'accès, le programme teste avant tout accès si la donnée est présente localement. Si ce n'est pas le cas, un message est envoyé vers le détenteur de l'objet pour en obtenir une copie. Bien sûr, le programme est arrêté pendant cette attente. La figure 8 montre le déroulement d'un programme sur un système distribué. Nous voyons que le noeud du haut a besoin des objets o, p, q et r. Par contre, p, q et r ne sont pas présents localement donc le programme doit envoyer des demandes au noeud distant à chaque accès.

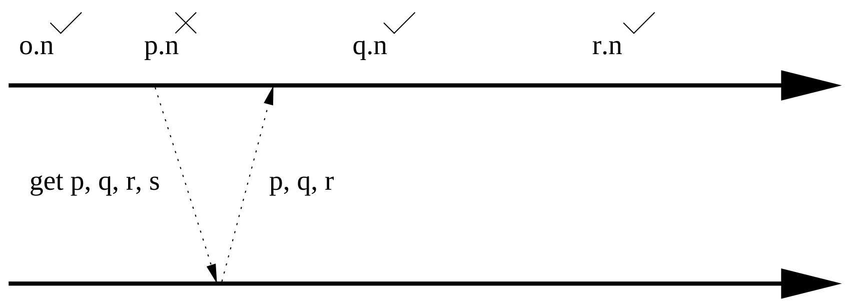


Figure 9: L'exécution du système de requêtes d'objets de Jackal utilisant un système de préchargement

Dans ce contexte, Esodyp a pour but de modéliser les accès mémoire afin de précharger les données. Si les prédictions sont correctes, le système ne s'arrête plus puisque les données sont toutes présentes localement. Par contre, l'intégration du modèle dans le système Jackal demande quelques modifications internes. Nous avons nommé le nouveau système *Esodyp+*. Jackal utilise un adressage appelé GOR (*Global Object Reference*) qui est un couple (*indice du noeud*, *adresse locale de l'objet*). Chaque noeud exécute une partie du programme sur une machine distante et se voit assigner un indice unique. Avec ce couple, il est donc possible de référencer tous les objets du programme global.

Une autre modification au modèle Esodyp consiste à faire produire au système de multiples prédictions à la fois. Esodyp est conçu pour prédire un seul élément à chaque appel. Dans un système distribué, afin de réduire le nombre de messages, il faut donc pouvoir prédire plusieurs éléments à la fois et envoyer ces demandes en un seul message. On peut voir le déroulement du nouveau système distribué utilisant Esodyp+ dans la figure 9. Lorsque le noeud du haut s'arrête lors du défaut de cache pour l'objet p, le prédicteur envoie une requête pour q, r et s simultanément, bien que s ne soit pas utilisé par le programme dans cet exemple (c'est donc une prédiction invalide). Lorsque le programme reçoit ces données, il peut continuer l'exécution sans interruption.

Le système Esodyp+ est comparé à deux autres prédicteurs : un prédicteur de pas nommé *Stride* à faible coût mais qui ne gére pas correctement les comportements complexes; et un prédicteur nommé *Delphi* qui utilise une table de hachage et les trois derniers éléments afin de calculer la clé de hachage. Quatre programmes parallèles ont été testés. Nous avons montré que le modèle Esodyp+ était capable généralement d'avoir la meilleure précision tout en gardant un faible coût de calcul.

4.2 Optimisation automatique dans le système Jackal

Esodyp+ a été intégré dans Jackal mais un défaut persiste toujours : pour que le système prenne en compte les prédictions, le programmeur doit ajouter manuellement les appels au modèle Esodyp+. Ceci veut dire qu'il doit avoir une connaissance approndie du programme cible afin d'instrumenter le code correctement.

La deuxième partie de ce chapitre présente un système dynamique qui évite cela en modifiant directement le système Jackal. Ce changement modifie la façon dont Jackal gére les vérifications des accès mémoire. Jackal sépare les accès mémoire en quatre catégories pour des raisons de performances. Tout d'abord, il différencie les accès en lecture et ceux qui modifient aussi la donnée. Ensuite, la gestion d'un objet et celle d'un élément dans un tableau sont différentes.

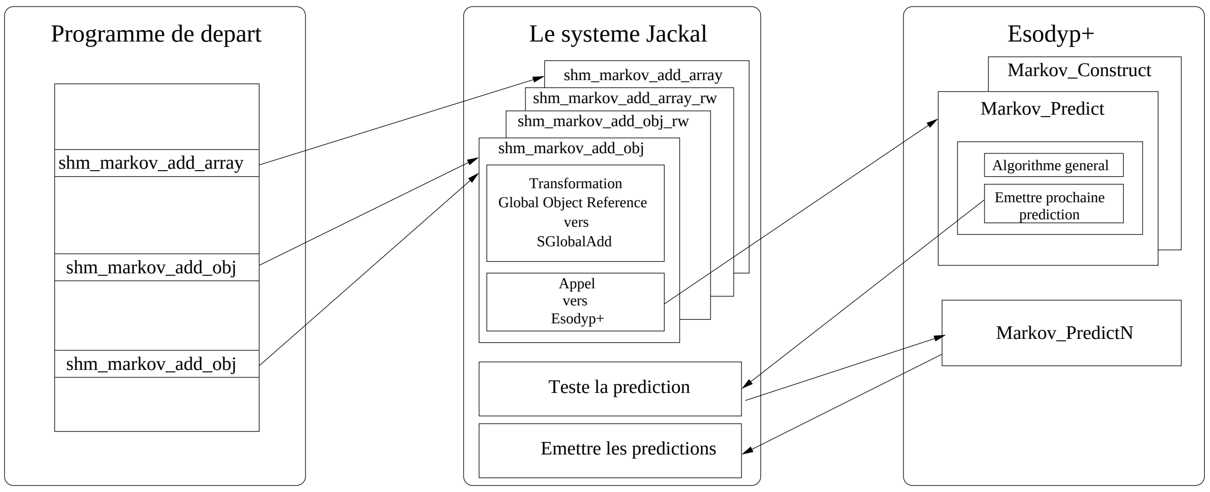


Figure 10: La séquence d'appels de fonctions entre le programme cible, le système Jackal et le prédicteur Esodyp+

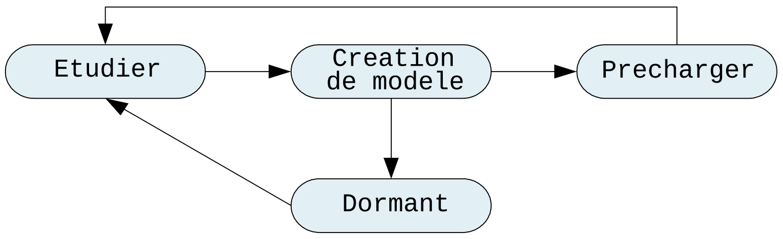


Figure 11: Automate fini représentant les transitions entre états des vérifications d'accès

Pour chaque vérification d'accès, dépendant de son type, un appel distinct est effectué.

La figure 10 montre les appels entre le programme cible, le système Jackal et le prédicteur Esodyp+. D'abord, lorsque le programme effectue une vérification d'accès, il passe par un appel qui transforme l'adressage Jackal sous forme de GOR à un adressage intermédiaire appelé *SGlobalAdd* que le modèle Esodyp+ pourra utiliser. Ensuite, il appelle le modèle Esodyp+ qui est en train de construire le graphe ou de mettre à jour le modèle et prédire. Si une prédiction est émise, Esodyp+ appelle la fonction du système Jackal qui vérifie si la donnée prédite est locale ou non. Si elle est déjà locale, ce n'est pas la peine de provoquer un préchargement. Par contre, dans le cas contraire, un appel à la fonction `Markov_PredictN` est effectué pour demander à Esodyp+ un certain nombre de prédictions. Cette liste d'éléments est ensuite envoyée aux noeuds concernés qui renverront en retour les objets prédits.

La dernière modification que nous avons faite au système Esodyp+ concerne la gestion dynamique de la distance de préchargement. Elle permet d'adapter le nombre d'éléments prédits pour réduire encore plus le nombre de messages envoyés. Par contre, plus la distance est grande, plus le nombre de mauvaises prédictions est possible. Selon le nombre d'objets qui sont renvoyés par le noeud distant, le nombre d'objets prédits est mis à jour.

Enfin, le système général utilise un automate fini pour gérer l'instrumentation des différentes vérifications d'accès. La figure 11 montre les transitions entre les états. Le système commence dans l'état *Étudier* afin de sélectionner les accès coûteux. Une fois qu'un accès a été exécuté un certain nombre de fois, l'accès est envoyé vers l'état *Création de modèle*. Dans cet état, un modèle Esodyp est créé pour le comportement mémoire de l'accès et, si le modèle est capable de

		Temps (en secondes)				Messages (en milliers)			
	Nodes	w/o	Esodyp+	DyCo	Dyn. *N*	w/o	Esodyp+	DyCo	Dyn. *N*
SOR	2	34.8	34.7	35.0	**34.5**	28.2	**11.5**	14.6	11.9
	4	18.6	**17.5**	19.2	18.0	84.1	**34.0**	44.1	34.7
	6	13.2	**12.2**	13.5	12.8	141.8	62.6	74.3	**56.7**
	8	11.0	**10.0**	11.4	10.4	199.2	85.2	104.6	**80.4**
Water	2	122.7	110.3	106.0	**93.7**	1713.5	1088.9	1114.6	**871.6**
	4	73.3	58.4	65.8	**56.3**	2941.5	1760.4	1843.3	**1401.8**
	6	68.0	55.6	58.0	**49.0**	3680.7	2183.6	2180.0	**1692.6**
	8	64.5	54.5	56.4	**45.7**	4299.6	2497.7	2612.9	**1984.3**
Blur2D	2	10.3	**3.5**	7.4	7.8	227.1	**68.3**	143.2	138.6
	4	6.9	4.0	4.3	**3.2**	394.5	182.0	159.4	**87.6**
	6	8.2	5.2	4.3	**3.8**	494.8	287.9	181.6	**142.0**
	8	10.0	7.9	5.2	**4.2**	589.2	403.6	228.7	**170.2**
Ray	2	41.9	**41.4**	41.8	42.9	9.2	**6.6**	8.5	8.4
	4	21.4	**20.9**	21.6	21.2	27.6	**19.5**	25.7	25.0
	6	14.9	15.1	15.1	**14.8**	46.2	**32.7**	43.4	42.0
	8	12.5	**12.0**	12.5	**12.0**	64.8	**46.0**	60.3	58.9

Table 1: Temps d'exécutions et nombre de messages (le meilleur est en gras).

prédire correctement, l'accès est envoyé vers l'état *Précharger*. Dans cet état, les accès appellent un même modèle qui provoquera des demandes de prédictions. Si ce modèle provoque trop d'erreurs de prédictions, les accès sont renvoyés vers l'état *Étudier*. Enfin, si le modèle créé dans l'état *Création de modèle* n'est pas convenable, l'accès est envoyé vers l'état *Dormant* et est désinstrumenté. Il sera réinstrumenté plus tard pour revérifier son comportement mémoire.

4.3 Performances

La table 1 montre les résultats obtenus. Les quatres programmes ont été exécutés utilisant 2 à 8 noeuds. Ensuite, les versions sans instrumentation (w/o), avec l'instrumentation du programmeur présenté dans la sous-section précédente (Esodyp+), sans modification dynamique du nombre de messages (Dyco) et celle avec la modification dynamique (Dyn) sont comparées. Les versions Esodyp+ et Dyco prédisaient 10 messages en avance.

Bien que le chapitre 4 donne plus de détails sur ces résultats, nous donnerons un résumé rapide. Pour le programme `SOR`, le nombre de messages est assez faible et la différence entre les différentes versions ne sont pas importantes pour 2 ou 4 noeuds. Pour une exécution sur 8 noeuds, la version *Esodyp* réduit le nombre de messages de 57% par rapport aux 60% de la version dynamique. Par contre, le système dynamique est un peu plus coûteux en terme de temps d'exécution puisqu'il faut instrumenter le programme et gérer les différentes vérifications d'accès.

Pour le programme `Water`, le système dynamique est nettement meilleur que les autres solutions et arrive à réduire le temps d'exécutions de 29% sur 4 noeuds et permet une réduction de presque 55% du nombre de messages. Le programme `Blur2D` profite du mécanisme dynamique pour limiter le nombre de prédictions faites en même temps, ce qui permet de limiter le nombre de mauvaises prédictions. Enfin, le programme `Ray` n'a pas assez de messages pour que le système dynamique puisse vraiment se mettre en place. En conséquence, le système ne compense pas son coût en terme de temps d'exécution mais aussi en nombre de messages par rapport à la version instrumentée (Esodyp+) qui sait tout de suite quelle zone de code optimiser.

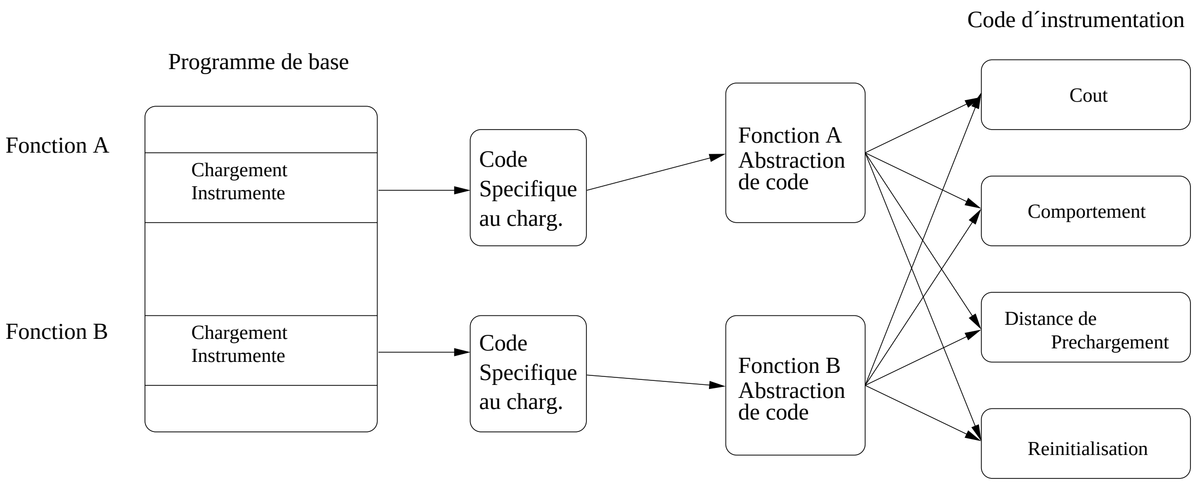

Code d´instrumentation
Programme de base
Fonction A
Fonction B
Chargement Instrumente
Chargement Instrumente
Code Specifique au charg.
Code Specifique au charg.
Fonction A Abstraction de code
Fonction B Abstraction de code
Cout
Comportement
Distance de Prechargement
Reinitialisation

Figure 12: Abstraction de code. De gauche à droite, le programme de base va dans le code spécificique d'un chargement. Ensuite, il saute dans le code spécifique de la fonction dans laquelle il se trouve et ensuite exécute un des modules d'instrumentation avant de revenir

5 Abstraction de code et intégration du modèle Esdoyp

Le chapitre précédent présente l'intégration d'Esodyp dans le domaine des systèmes distribués. Nous avons montré qu'il est possible d'utiliser Esodyp comme un filtre pour décider quels accès optimiser. Le chapitre 5 reprend le travail présenté dans le chapitre 2 et y intégre Esodyp. Au final, le système proposé modifie dynamiquement le code binaire d'un programme afin de précharger les chargement coûteux du programme. Un article présentant les travaux contenus dans ce chapitre est en cours de soumission [10].

5.1 Abstraction du code d'instrumentation

Malheureusement, l'intégration d'Esodyp demande des modifications du système précédent. Puisque le système nécessite à présent de faire une instrumentation plus complexe, cela veut dire que l'espace mémoire alloué pour chaque chargement doit être agrandi. Par contre, cela n'est pas envisageable lorsque le nombre de chargements devient important. Au lieu de cela, deux couches intermédiaires entre le code de base du programme et le code d'instrumentation ont été ajoutées afin de permettre d'intégrer le modèle Esodyp.

La figure 12 montre la nouvelle infrastructure du système. Elle est découpée verticalement en quatre parties. Par rapport au système précédent, un état est toujours associé aux chargements et l'automate est présenté par la suite. Nous appellons *module* le code associé à chaque état de l'automate.

Le code du programme se trouve sur la gauche de la figure et les chargements ont été remplacés au début de l'exécution par des sauts inconditionels.

Le code spécifique aux chargements est ensuite exécuté. Une petite zone mémoire est dédiée à chaque chargement, charge l'adresse de la structure associée et fait un saut vers une zone de code spécifique à la fonction.

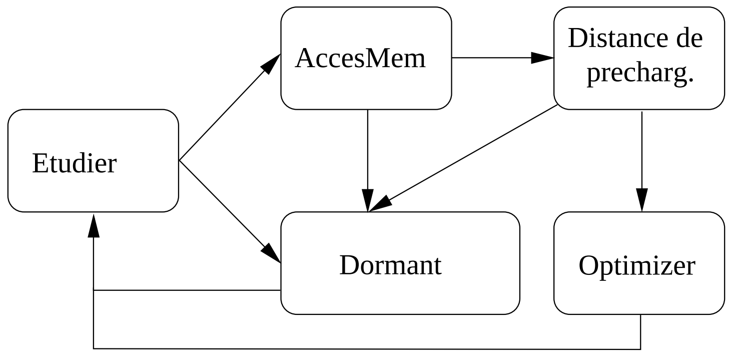


Figure 13: L'automate fini représentant les états et les transitions des chargements

Le code spécifique aux fonctions est un code qui commence par la sauvegarde de certains registres. Ensuite, utilisant les informations contenues dans la structure du chargement, l'état courant du chargement et l'adresse du code d'instrumentation associée sont récupérés. Enfin, un saut inconditionnel est fait vers ce module.

Les modules sont représentés par la partie droite de la figure. C'est par exemple dans ces codes que les calculs de latences sont effectués. Le calcul de la distance de préchargement est effectué dans le module *Coût*.

5.2 Les états des chargements

Les différents modules sont représentés dans la partie droite de la figure 12 et chacun représente le code d'un état que peut prendre un chargement. La figure 13 présente l'automate gérant les transitions entre les états. Puisque le code d'instrumentation est plus complexe, beaucoup d'opérations sont internalisées par rapport au système présenté dans au chapitre 2.

Les chargements commencent donc dans l'état *Étudier*. Lorsqu'ils sont exécutés un certain nombre de fois, la latence moyenne est calculée et, si elle est assez importante, le chargement est envoyé vers l'état *AccèsMem*. Comme dans le système précédent, si ce n'est pas le cas, le chargement est désinstrumenté pour limiter le coût du système global et son état devient *Dormant*. Par contre, une amélioration a été apportée au système : la plupart des changements d'état ne nécessitent plus aucune réécriture du code binaire. En effet, les états sont définis par un identifiant unique qui est chargé dans le niveau d'abstraction de la fonction. Cet identifiant donne l'indice de l'état et permet de décider quel module exécuter. De ce fait, ce changement d'état se fait en modifiant simplement l'identifiant.

Dans l'état *AccèsMem*, le chargement remplit un tampon avec les adresses chargées. Lorsque ce tampon est plein, le système choisit l'élément le plus présent et crée un modèle Markovien avec Esodyp afin d'étudier son comportement mémoire. S'il est composé de pas constants ou s'il contient un cycle de pas, il va vers l'état *Distance de préchargement* où il teste diverses distances. Sinon, il est envoyé vers l'état dormant et est désinstrumenté. Lorsqu'une distance est trouvée, il est enfin optimisé. Par contre, dans le cas contraire, il est désinstrumenté.

Dans le système précédent, chaque test d'une nouvelle distance de préchargement demandait l'intervention du deuxième *thread*.

Enfin, pour déoptimiser ou réinstrumenter les chargements *dormants*, le système utilise encore un deuxième *thread* qui modifie le code binaire. Par contre, tous les autres changements

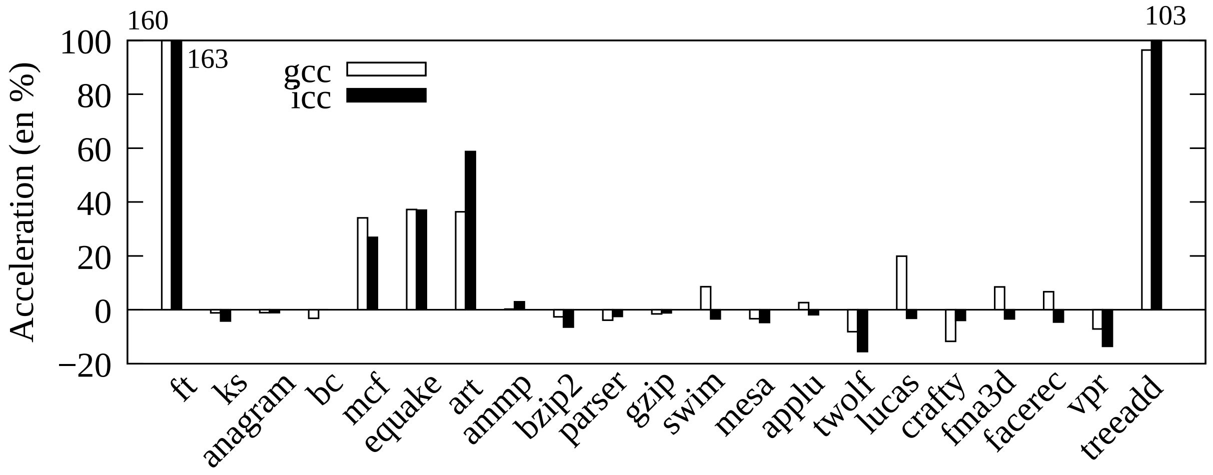


Figure 14: Accelerations obtenues pour les programmes générés par les compilateurs `gcc` et `icc`

d'états sont gérés en interne ce qui permet de ne plus attendre le réveil de ce deuxième *thread*. Ceci permet non seulement de rendre le système plus exact mais réduit sensiblement son coût.

5.3 Performances

La figure 14 présente les résultats obtenus avec ce système. Elle compare les résultats entre les deux compilateurs `gcc` et `icc`. Le coût du système a été évalué à 3% et les accélérations sont comprises entre 2% et 163%.

5.4 Conclusion

En conclusion de cette partie, le système que nous avons créé permet d'intégrer Esodyp dans un système dynamique et transparent. Utilisant le modèle afin de pouvoir décider quels chargements optimiser, le coût total du système est réduit au maximum. De plus, seuls les éléments propices à l'optimisation sont étudiés plus en détail.

Ce chapitre présente également une solution permettant une abstraction du code en ajoutant deux niveaux entre le code instrumenté et le code du programme cible. Ces abstractions permettent une écriture plus facile de l'instrumentation et, surtout, un code bien plus complexe peut être écrit car il est partagé par tous les chargements. Dans le cas contraire, il aurait fallu allouer plus d'espace par chargement, ce qui devient vite impossible du point de vue de l'allocation mémoire.

6 Conclusion et perspectives

6.1 Conclusion

Cette thèse présente des approches originales pour résoudre les problèmes de défauts de cache de données. Les systèmes dynamiques sont capables d'étudier, modéliser et optimiser des programmes cibles avec un coût réduit. Par contre, la plupart des systèmes se basent sur les blocs de base du code au lieu de se placer au niveau de l'instruction. Ceci ne permet donc pas de gérer chaque chargement indépendamment.

La première partie de cette thèse présente un système dynamique qui modifie directement le code binaire afin d'insérer des instructions de préchargement dans le programme. Chaque chargement du programme est associé à un état dans un automate fini. En utilisant un deuxième *thread* pour gérer les changements d'état, le système est capable de gérer les chargements et désinstrumenter ceux qui ne répondent pas favorablement à l'optimisation proposée. Sur un ensemble de programmes, ce système est capable de les accélérer de 2% à 143%.

Par contre, ce système présente quelques défauts. Premièrement, la sélection des chargements à optimiser est faite de façon naïve puisqu'il ne modélise pas le comportement mémoire avant de tenter une optimisation. Ce n'est seulement lorsque l'optimisation teste chaque distance de préchargement et ne réduit pas la latence du chargement que le système en déduit son incapacité à l'améliorer. Deuxièmement, puisque c'est le second *thread* qui gére les transitions d'état, il faut attendre le réveil de ce *thread* avant qu'un chargement change d'état. Enfin, à chaque transition, une modification du code binaire est nécessaire pour prendre en compte ce nouvel état.

Le premier point négatif, c'est-à-dire le fait que le système n'étudie pas le comportement mémoire des chargements, est abordé lors de l'implémentation d'un modèle Markovien. Nommé Esodyp, il se base sur une représentation sous forme de graphe pour comprendre le comportement mémoire actuel du programme en utilisant les précédents accès pour émettre des prédictions. Implémenté sous forme de bibliothèque de fonctions, le modèle peut être intégré par un programmeur pour accélérer les programmes.

Ce modèle a été étendu, et nommé *Esodyp+*, pour une intégration dans un système distribué nommé *Jackal*. Cette extension a permis d'analyser les messages envoyés entre les noeuds distants et prédire quels seraient les objets nécessaires aux phases de calcul du programme. Par exemple, Esodyp ne permettait que de prédire un élément à la fois. Or, dans un système distribué, il faut pouvoir prédire plusieurs éléments en même temps afin de réduire le nombre de messages global. En comparant le système à deux autres prédicteurs, il a été montré qu'Esodyp+ permettait de faire des prédictions précises et à faible coût par rapport aux deux autres prédicteurs.

Mais ceci demande toujours une connaissance du programme à optimiser puisqu'il faut intégrer des appels vers le modèle. Une modification dans le système Jackal a permis de transformer les vérifications d'accès afin de permettre à Esodyp d'être automatiquement appelé par le système sans l'intervention du programmeur. En utilisant un automate fini pour gérer les accès aux données et en utilisant une modification dynamique pour la distance de préchargement, le système final obtient généralement de meilleurs résultats que le système intégré par le programmeur.

Finalement, cette thèse présente une solution pour intégrer le modèle Esodyp dans un système d'optimisation dynamique. Ceci demandait de modifier l'infrastructure interne du système et deux couches d'abstraction sont apparues. Cette abstraction a permis d'intégrer des instrumentations plus complexes, allégeant le travail du deuxième *thread*. Par exemple, le changement d'état n'est plus intégralement géré par ce dernier puisque l'exécution du code d'instrumentation utilise un indice d'état qui définit quel code utiliser.

Cette abstraction permet aussi de réduire considérablement la mémoire globale utilisée par le système dynamique et permet d'intégrer Esodyp pour modéliser, lors de l'exécution, les comportements mémoires des chargements coûteux. Si le modèle décide d'optimiser le chargement, différents pas sont testés comme pour le système précédent. Par contre, puisque le système permet une instrumentation plus complexe, il est possible de tester plus de distances à un coût plus faible.

Ces optimisations ont finalement permis de rendre le système général plus complexe, plus précis et plus léger. Le coût moyen du système a été évalué à 5% de l'exécution des programmes et a permis d'en accélérer certains de 2% à 163%.

6.2 Perspectives

Cette thèse a aussi permis d'entrevoir la possiblité de différentes extensions. Ceci a été remarqué lors de l'intégration du modèle Esodyp dans le système Jackal. Le même automate fini ne pouvait pas être utilisé pour des raisons techniques et cela a demandé de modifier le comportement du système. Par exemple, un deuxième *thread* était difficilement possible dans ce contexte et, à la place, l'état *Dormant* posséde un test interne qui vérifie s'il est temps d'étudier à nouveau le comportement de l'accès.

Puisque les systèmes proposées sont entièrement logiciels, nous travaillons sur des versions pour les architectures x86 et embarquées. Un système d'optimisation sur une architecture embarquée pose des problèmes réels d'allocation mémoire. Le système d'abstraction de code présente par exemple de réelles applications dans ce contexte.

Une autre voie est l'abstraction du noyau du système. Ceci permettrait d'avoir le même code de base qui peut directement être placé sur différentes architectures. Pin [76] propose à ses utilisateurs d'écrire des *PinTools* qui fonctionnent sur n'importe quelle architecture prévue par Pin. Une telle séparation est intéressante mais demande probablement un langage intermédiaire qui serait compilé à l'exécution du système.

Finalement, le code du modèle Esodyp a été écrit dans le langage C. Il est donc possible d'utiliser un autre modèle mathématique pour sélectionner les chargements, comprendre et prédire les comportements mémoire d'un programme. Nous travaillons sur une abstraction supplémentaire qui permettra d'intégrer directement du code de haut-niveau que l'utilisateur pourra écrire. Ceci permettra d'écrire une vraie plateforme logicielle d'optimisation. Certes, les plateformes comme Pin, DynInst [18] et DTrace [19] existent, mais elles sont généralement orientées vers l'instrumentation uniquement et se base sur les blocs d'instructions. Il est difficile de trouver un système qui permet à l'utilisateur de définir parfaitement ce qu'il veut accomplir d'un point de vue de l'optimisation mémoire et qui gére chaque chargement séparément.

Introduction

Tasks handled by compilers tend to be more numerous and complex. From a simple text translator of a high-level programming language to a binary executable code guaranteeing the functional requirements, the compiler is now asked to handle various non-functional goals either exclusively or simultaneously. For example, the generated binary code has to:

- result in an execution time being as low as possible;
- be respectful of real-time constraints such as maximum execution time or fixed scheduling;
- minimize electrical power consumption;
- minimize memory consumption;
- result in a predictable execution behavior.

Moreover, all these objectives are often inconsistent or have strong interferences that may cancel, or even worsen, their impact. For example, the objective of execution time minimization is obviously in opposition with power consumption minimization, or with predictable behavior. Also the techniques used to reach them can be various and their choices can determine the impact of these contradictions.

All these aspects require a full understanding and full control of the behavior the software has when run on hardware. A piece of software, even if it has been built, analyzed, proved or simulated using the most advanced tools, will almost always have an unexpected behavior when confronted to the hardware that is executing it. Thus so-called *static* analysis and transformation approaches are limited by the ignorance of the impact of this confrontation. This is also true for the hardware side, since hardware is sometimes built without any knowledge of the software that might be run on it. But in this thesis, we remain on the software side, confronted to a fixed hardware platform.

An answer to this control over the behavior issue consists in opposing, or in assisting, static approaches with so-called *dynamic* approaches, i.e., approaches where program analysis and transformation occur during the execution. The analyzer or optimizer, that we can call more generally *compiler*, can access non-functional information only available at runtime, resulting from the confrontation between the software and the hardware, but also related to some non predictable functional evolution of the computation.

However, adding such a dynamic compiling process to the target software yields difficult conception issues in order for this process not to be in direct contradiction with the performance goals the programmer had established. For example, it is obvious that adding some compiling code that runs simultaneously with the target program yields an issue about increasing execution time if the objective is execution time minimization. It is the same for all the objectives cited above. However, many works dealing with dynamic compilation, as those presented in this manuscript, show that it is quite possible to reach these objectives by widely counterbalancing the negative impacts by the positive ones. For example, the additional memory consumed by the dynamic process can be negligible compared to the reduced memory consumption obtained due to it.

This thesis concerns the area of developing dynamic approaches for program behavior control. More particularly, the main topic of the works presented here concerns execution time minimization on a uni or multi-processor architecture, through memory access anticipation by data prefetching, and in a way that is transparent to the user. The developed processes include a dynamic behavior analysis phase, where memory access latencies are measured, a dynamic optimizing transformation phase if these transformations have been selected as useful, and where prefetch instructions are inserted into the binary code, a phase of dynamic analysis of the optimization efficiency, and finally a phase of transformation cancellation if the transformations have been observed as inefficient. Moreover, all these phases are applied individually to each memory access, and eventually several times when memory accesses show variable behaviors during the run of the target software.

We show that it is possible to conceive such a dynamic system which has a relative complexity and which is entirely software, i.e., which does not make use of any hardware functional specificity, which is efficient for many programs and which is not penalizing for the others. At our knowledge, our work is a first proposal of an entirely software dynamic system which does not use any code interpretation mechanism.

Chapter 1 presents the state-of-the-art concerning profiling mechanisms, prefetching techniques and dynamic instrumentation and optimization frameworks. In order to be efficient, dynamic instrumentation or optimization frameworks need to lower their overhead at a maximum. This is performed by profiling mechanisms that select the costly code regions and inform the frameworks. One solution to accelerate programs is to reduce data cache misses. A common technique to achieve this is to prefetch the data. A prefetch requests the predicted data before a program requires it. This prevents a data cache miss. Dynamic instrumentation or optimization systems require such techniques in order to reduce their overhead. This chapter presents various works concerning all of the above topics.

Chapter 2 presents a lightweight dynamic system that is able to modify the binary code at run-time. The system selects costly loads and tries to prefetch the data using a stride-based predictor. A stride-based predictor calculates the difference of addresses between two data accesses. This is efficient in the case of a constant memory traversal behavior. Each load is associated to a state in a finite state machine. A second thread is created to monitor the loads and update the state of each load. The chapter then presents the behavior of the system when applied on various benchmark programs and the implementation involved is detailed. The system is implemented on the Itanium-2 processor and has a low overhead of 5% while the speed-up on certain programs ranges from 2% to 143%.

However, the selection scheme of chapter 2 is not very effective since it does not try to understand the behavior of the memory accesses it tries to optimize. Instead, it simply tries to optimize a load and stops if it is not effective. Chapter 3 presents a Markovian model named *Esodyp* that is constructed and updated at run-time. This means that it must have the lowest overhead possible while maintaining a good accuracy. Furthermore, the system must be able to calculate the predictions quickly and depending on a variable number of past elements. It is shown that, a programmer can insert calls to Esodyp to achieve significant speed-ups on certain benchmark programs.

The creation of the Esodyp model lead to its integration into the Distributed Shared Memory (DSM) system named *Jackal*. The system compiles Java code into native binary codes and uses access checks to verify if the data is locally available. If not, the local node sends a request message and waits for its answer. Esodyp was extended into Esodyp+ in order to handle bulk predictions and the address representation of Jackal. Esodyp+ was then compared with two other predictors and it is shown that it performs well while maintaining a low overhead. Second, instead of adding calls to Esodyp+ in the benchmark programs, the chapter shows how the Jackal system can be modified to integrate a dynamic optimization system that handles the code transparently without the help of the programmer.

Chapter 5 then shows how to integrate Esodyp into a dynamic optimizer for the Itanium-2 processor. However, the system presented in chapter 2 is not directly extended since certain implementation reasons require internal changes. Using an abstraction level, the memory of the system is reduced while integrating more complex instrumentation. In the previous system, a second thread is used to change the states of each load. However, in this chapter, the system handles these changes internally, reducing the tasks of the second thread to a minimum. The end-result is a dynamic framework that is more precise while using more complex instrumentation techniques. Also, Esodyp is integrated to decide whether to optimize or not a particular load. The overhead of the new system is evaluated at 3% and the optimization accelerates programs ranging from 2% to 163%.

Finally, a conclusion and perspectives are presented.

Chapter 1

State-of-the-art

This chapter presents the state-of-the-art concerning profiling mechanisms, prefetching techniques and dynamic instrumentation and optimization frameworks. Profiling mechanisms give the optimizer, whether a compiler or a dynamic framework, the information needed to apply the correct optimizations. For example, this chapter presents prefetching techniques that can be used in order to request data in advance, limiting the number of processor stalls. Finally, we present dynamic frameworks that are able to profile and optimize programs during run-time without any prior knowledge of the target program.

1.1 Introduction

This thesis presents different techniques to profile, understand and optimize memory operations. Data loads for instance have been a common interest for many researchers since they are generally considered as being the bottle-neck of modern architectures. Since the gap between speeds of processors and memory accesses is growing every year, techniques must be found to limit the incurring stalls. This chapter presents a brief overview of the different possibilities in three major domains.

First, before being able to optimize a program, of the target program needs to be understood. This is called *profiling* and different techniques exist. Hardware profilers use architectural components in order to gather the necessary information. If time is not a factor, compilers can insert calls and special code into the program to provide the information by executing a modified and slower version of the program. If the code cannot be recompiled or if there are run-time constraints on the program, dynamic instrumentation frameworks can attach themselves to a running program and modify the code before detaching themselves. Finally, dynamic profilers exist and often rely on sampling techniques to lower the overhead. This last group of profilers are useful since they are perfectly adapted to combining a profiler with a dynamic optimizer.

Second, once the profiles have been processed, an optimization system must be defined. Prefetching techniques are well-known to handle data cache stalls. By loading the data ahead of time and if done correctly, the processor constantly has the information it needs to continue the execution of the program. We divided the prefetchers in three groups: stride, correlated and thread predictors. A stride predictor uses the recurring distance between two consecutive accesses in order to prefetch the data. This requires that the stride of the different load addresses remains identical. On the other hand, correlated prefetchers use more complex models handling many past events to understand the current memory behavior of loads. This second group has a higher CPU overhead since the creation and use of the model is more complex but these models are able to handle a bigger number of loads in a program. Finally, thread predictors are integrated into the execution stream and run concurrently with the program. However, they execute a simplified version of the code and, by reducing the computation code, this second thread executes faster than the base program. This means that the predictor directly accesses the data before the main thread, thus reducing the data cache misses.

Finally, we give a brief overview of the existing dynamic frameworks. These frameworks generally contain dynamic profilers and dynamic predictors in order to achieve speedups or limit their costs. Again, we can divide this group into two subgroups. The dynamic instrumentation frameworks offer the users a possibility to understand at run-time the execution and general behavior of different programs. The dynamic optimization systems are generally specialized in carrying out a specific task to optimize programs.

Though this state-of-the-art is not exhaustive, it should give the reader a general idea of the prior main contributions in each particular specialty.

1.2 Profiling

Different programming paradigms exist. Whether the program is interpreted or compiled, both interpreters and compilers optimize the code in order to achieve the best performance for the target program. This performance can be, for example, the amount of energy required, the quantity of memory used or the execution time. However, optimization schemes are numerous and deciding which to use and how is a complex problem. Certain optimizations can simply be achieved by analyzing the code (*static optimizations*), but others can only be done by understanding the behavior of the program during the execution, (*dynamic optimizations*).

In order to analyze the behavior of an executing program, the code is instrumented and different profiling goals are used. Prefetchers need to know what loads are considered delinquent. A delinquent load is a load that often leads to a cache miss. Code-layout optimizers need to detect frequently used code regions, also called *hot traces*. Depending on the information needed, profilers instrument regions of code, redirect function calls to maintain control and/or directly modify the binary code at runtime to achieve their goals.

We can classify the different profiling techniques in 4 groups:

- Hardware profilers that use existing components, extend them or even create new hardware elements;
- Compiler assisted profilers or post-compiler off-line techniques that use information provided by the compiler to achieve their profiles or modify the binary before execution to correctly instrument the program;
- Dynamic instrumentation frameworks that attach themselves to the program and modify it dynamically;
- Dynamic profilers used for dynamic optimizations which are lighter and incur less overhead than the previous group.

Hardware profilers are, by definition, dependent of or even integrated into the underlying architecture. For optimizing frameworks, they are useful since they generate a very low overhead.

Modern processors contain generally many hardware performance counters that count, monitor and check various events such as instruction/data cache misses, specific cache misses, bus information or *Translation Lookaside Buffer* (TLB) misses. The advantage of using hardware profiling schemes is the fact that the optimizer can concentrate on accelerating the program without having to instrument the code itself. However, this also means that optimizing frameworks cannot be used on other architectures as easily as totally software solutions.

Compilers, before generating code, generally use an intermediate representation in the form of a tree or a graph to represent the code and perform different optimizations. If integrated into the compiler, a profiler can modify the graph before code generation by inserting instrumentation and/or directly modifying the target program. The profiles are then studied by off-line techniques that generally extract as much information as possible and use complex or time consuming algorithms that could not be used during the execution of a program to define the behavior of a program and how to optimize it. For example, a loop recognition algorithm for memory access traces is presented in [25].

Post-compiler techniques directly modify the binary executable after the compiler has generated the code and before execution. These techniques sometimes rely on recreating the intermediate representation (IR). With the IR, it is possible to inline functions from different files, optimize certain code-layouts or modify global data structures in ways that were not possible when the compiler was generating each source file. However, modern compilers can also generate intermediate files in order to address these issues, *Icc* has for example an *interprocedural optimization* option. A problem is the register allocation scheme since it is done during the compilation phase. Post-link methods must rely on liveliness analysis to find free registers, allocate more registers if possible, use hardware solutions or find other techniques in order not to require extra registers.

Finally, dynamic solutions have low overheads which make them perfect solutions for dynamic optimization schemes. Though dynamic solutions with a high overhead exist, they are either used for off-line profiling or they do not profile the program during the whole execution. They use methods similar to post-compiler techniques but at run-time. Finally, dynamic solutions are able to pause, modify and resume the execution of the target program. Attached to optimization schemes, lightweight solutions are able to dynamically decide the modifications needed for a particular program.

The rest of this section presents each class of profilers in a more detailed manner.

1.2.1 Hardware Profiling

Hardware profiling consists in using the processor to extract hot-traces, delinquent loads or other information useful for optimizers. The hardware solutions can be hardware performance monitoring (HPM) using internal counters, extensions to current hardware components or even new elements that can be added to the processor. This subsection presents a few hardware profiling techniques.

When profiling tools are implemented for specific processors, they tend to use all the available hardware information. For example, the Itanium processor [53] contains hundreds of performance counters that can easily be used by a profiler [96, 73, 78]. Other proposed techniques extend the current architecture by adding a few simple components to the existing hardware [114, 28, 33, 63]. For example, a simple *dirty bit* or a cache flush mechanism can be added to the hardware to handle new algorithms. In [28], Conte et al. extend the branch predictor included in the processor with these mechanisms. A program is compiled containing a special identifier (e.g., a Unix Magic Number) signaling it contains a profiling table in the executable. At startup, the

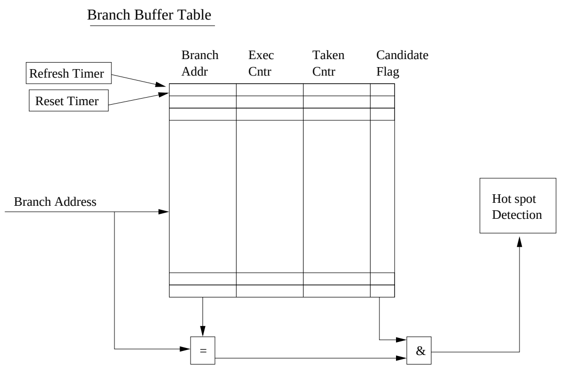


Figure 1.1: The Hot Spot Detector

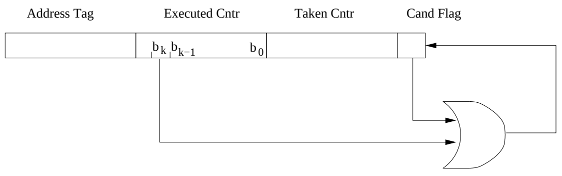


Figure 1.2: The candidate flag calculation

profiling table is loaded in memory along with the binary executable. The processor is then periodically sampled by the operating system in order to update an arc-based profile of the different executed branches. Finally, at the end of the execution, the updated table is rewritten in the executable for storage. Profiles from different executions are built at a very low cost and used to recompile the code accordingly. The advantage of such a system lies in the low overhead of the system. Since there is almost no slowdown, users can use the profiled version of the program without any of the problems that can arise with the difference in execution time. The common *compile-profile-recompile* paradigm is thus replaced by a *compile-use-recompile* solution which can be done without the knowledge of the user.

More elaborate schemes add completely new hardware components into the processor [114, 33, 82, 119, 120]. For example, Merten et al. [82] add a hot spot detector, branch behavior monitoring and a general system management system to detect and handle hot traces. Figure 1.1 shows the hot spot detector used by this method. It contains a Branch Behavior Buffer to store the branch addresses, execution counters, taken counters and whether or not the branch is a hot spot candidate, also called the *candidate flag*. Figure 1.2 shows how the candidate flag is set to 1 when the execution counter reaches a determined threshold. By using a logical binary *or* with the candidate flag and the b_k bit of the execution counter, the candidate flag is set to 1 when that execution bit is set to 1. Then, using a the binary operation *or* with the candidate flag, the value remains at 1. Once a candidate branch is found, the system starts a hot spot detection phase in order to determine if the candidate branch accounts for at least a certain percentage of the total branches executed. If this is true, the method sends an interruption to the operating system, letting it handle and decide how to optimize or modify the corresponding code of the program. Finally, since the branch buffer table has a limited size, multiple branches can be sent to the same line. These conflicts are resolved using two different timers. First, a refresh timer is used to periodically remove any non candidate branches from the table, allowing the profiling of current branches. Second, a reset timer periodically clears the whole table. This allows branches conflicting with candidate branches to be profiled. Though, hybrid systems are common, the interaction of the hardware profiling system with the operating system is not always needed. If the proposed technique can handle distinct simultaneous program executions efficiently, it can be a good idea to centralize the software side of the scheme and integrate it into the operating system. However, one can argue that it is preferable to have a separate software framework directly attached to the hardware in order to keep the whole system independent of the operating system.

Another solution called *Trident* [119] uses a hybrid approach to limit the overhead of the

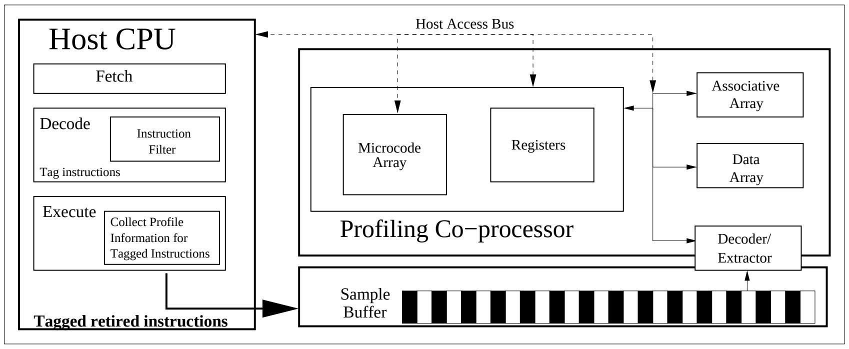


Figure 1.3: Co-Processor solution

framework. Using hardware components to detect hot traces, the system is able to keep a very low overhead since there is no need to handle this in software. Once a hot trace is detected, a hardware event *Hot trace* is created that will later be consumed by a helper thread. Once consumed, an optimized version of the trace is created and added into the code cache. In addition to optimizing the code, a hot value profiler is added to the code. This enables the hardware to monitor and detect frequently occurring values from load instructions. If such a load exists, a second type of event *Hot Value* is generated. When this event is consumed, the code of the trace is further optimized, using more aggressive techniques such as dynamic value specialization. This means that the same trace can be optimized twice and, using a garbage collector, the old traces are periodically removed from the code cache.

Zilles and Sohi [123] showed that using a co-processor to handle the profiling is also a solution. Figure 1.3 shows the proposal. While the host CPU is fetching, decoding and executing the program, the profiling co-processor receives samples of the available information. A filter is used at the decode stage to limit the quantity of information that would need to transit between both processors. At the execution phase, the information is collected and then sent to a sample buffer in the co-processor. The information is then sent to a decoder/extractor module. As the profiling program studies the profiles, the buffer is emptied and the data is sent to the different available memory arrays. The co-processor contains three different memory arrays: the microcode array for the co-processor program storage, the associative array for efficient matching between different profile entries using a hash-table and the data array for general data storage. Different registers are also available in the co-processor: constant registers set at the initialization of the program, value registers used to handle the profiled entries and optimized index registers. An index register has a configurable width that is set at program load time and can be used as a saturating counter or a circular buffer index. Finally, when the co-processor has enough information, it can transfer it using a host access bus. This last technique gives an application a chance to specialize the hardware profiling scheme. Finally, in multicore and manycore architectures, such a programmable hardware profiling core would be an interesting feature.

The low overhead incurred by hardware profilers gives them a clear advantage on the different software techniques. Since they can be directly attached to the processor and can directly monitor the different events and behavior of a program, they can retrieve most of the information at a very low cost. However, the cost of adding a profiler into a chip is expensive and generally hardware profilers use hash tables and sampling techniques in order to keep the memory needed for the profile at a minimum. This compromise can be a serious problem if the optimizers need very precise information in order to modify the target programs correctly. The rest of this chapter present the different software techniques available.

1.2.2 Compiler aided profiling

Probably the most renowned compiler-aided profiler is *Gprof* [44] which is an extension of the Unix program *prof* introduced in 1979. *prof* gave users a solution to determine the most time consuming function but did not give the call graph of the program. In 1982, *gprof* added that functionality. By using a compiler option, the compiled code included calls to the profiler at the beginning of each function.

These calls render possible the construction of the call graph. Since the profiler is aware of what function is called and where the instruction of the call is located, it can create the arc between the caller and the callee. The second information *gprof* retrieves is the time spent in each function. This is done by sampling the value of the program counter periodically and inferring execution time from the distribution of the samples within the program. At the end of the program, the profiler can provide a histogram of the location of the program at each polling period. Once the program is executed, a profile output file is written. The call graph arcs are condensed and the program counter histogram is written. An associated program then parses this file to output the call graph and the approximated time spent in each function.

Call graphs and function execution time are not always sufficient to decide what optimization should be used. Memory access regularities provide additional information and, if profiled correctly, these accesses can be efficiently handled for example with stride-based predictors. Wu et al. [116] use a compiler to add instrumentation around allocation, de-allocation, load and store instructions. However, the quantity of information can become overwhelming and compression techniques must be used. Using a horizontal and vertical decomposition of the stream of information and using the Sequitur grammar [87, 86] to compress the profile, the system is able to achieve a better compression than conventional compression using the raw data and the Sequitur grammar. In this work, two memory profilers are presented: a lossless profiler and a lossy profiler. The lossless compression is 22% more compact compared to conventional Sequitur compression, whereas the second profiler is able to compress at a 3 order of magnitude with a slowdown averaging a factor of 11.5 times the native execution time.

Another technique to limit the size of the profile output is to instrument only a subset of loads. For example, finding regular stride patterns is important when considering prefetching as a technique to optimize. Prefetching consists in requesting data that will probably be needed in the future by the program. One compiler-aided technique, presented by Wu [117], consists in profiling a set of loads of the program. Since profiling every load throughout the execution would incur a too high overhead, loads are called equivalent if they are inside the same loop, in control equivalent blocks and their addresses are different only by compile-time constants. The system also handles out-loop loads if they show a strong single stride behavior. Using a sampling method that does not continuously profile the loads of a program, the system is able to reduce the overhead. Once the execution is finished, the compiler analyzes the profile output file to determine what loads should be optimized.

Once the compiler has processed the profile, it can make decisions and prepare the new optimized code. Kim et al. [62] present a more elaborate scheme using Markovian predictors. Once the profiles are examined, Markovian predictors are created and then dynamically loaded into the prefetching unit of the CPU at the appropriate point during the runtime to drive the prefetching. This is more a hybrid solution since the profiling and calculations are done in software, whereas the hardware handles the predictions and prefetching. Another solution, presented by Rabbah et al. [92], consists in extracting the memory access sequence and, using a cache simulator such as Dinero IV [37], detect delinquent loads and then attempt to optimize them during a second compilation phase.

Compiler-aided techniques are practical when relatively simple optimization schemes are to be integrated into the program. Another direct application are programs that have a constant behavior and do not directly depend on different input parameters. However, when multiple optimization schemes are possible and when the mere order of the optimization phases can modify and influence the outcome of the process, compiler-aided techniques are not general enough. Iterative compilation is the logical next step to this technique. By iterating through the compiler and execution of the program, the system can find the best optimization possible for a target program. The next section briefly presents this field.

1.2.3 Iterative compilation

Iterative compilation can be considered as an extension of the *compile - profile - recompile* paradigm. Instead of a single profile version of the code, the technique resorts to multiple testing to find the right optimization sequence or optimization parameters [108, 2, 66, 91, 15].

Interactive compilation systems [45, 88, 66] can be seen as iterative compilation techniques. VISTA [66] (VPO Interactive System for Tuning Applications) is an iterative compilation system that gives developers an opportunity to decide which optimization sequence should be used for their programs. Also, optimization passes can be programmed by the user. Using basic programming structures such as *if* or *while*, a user is able to create an algorithm deciding what sequence of optimizations passes should be used. Finally, an exhaustive search or a genetic algorithm can be used to find the best sequence using different optimization passes.

Even when concentrating on a specific optimization domain, the number of possibilities is huge. For example, when considering loop nests, classical optimizations range from loop reversal, skewing, fusion to loop peeling. When optimizing these loops, the optimization sequence and parameters of each technique can modify the output program. However, iterative compilation can be a solution to test each possibility or at least a representative set in order to find the best version. For example, Agakov et al. [2] show how the polyhedral model, a mathematical framework, can be used to traverse the whole set of distinct, legal affine one-dimensional schedules for a program. This means that every legal combination of optimizations that leads to a new version of the program is tested. Of course, this depends of the program, the data input, the compiler and the architecture. The authors define a legal transformation of a program as a modification of the sequence of instructions while maintaining the semantics of the program.

Another factor to consider is the time spent finding the right optimization sequence. Though the best optimization pass sequence with the correct parameters can be found, the number of trials can be overwhelming. Fursin et al. [43] present a solution that accelerates the process. Programs often exhibit stable phases during a single run. A stable phase can be defined as a period in the execution where the behavior is constant, during a single run [70, 23]. During phase intervals with the same performance, the system tests, at most, 16 different optimization schemes and can decide what is the best optimization. Also, these schemes are only applied on the most time consuming sections of code. This means that to automate the system, a previous profiler would be needed to determine what code regions need to be optimized.

Though iterative compilation and compiler-aided profiling have a lot of things in common the process is not the totally the same. Compiler-aided profiling tries to retrieve a maximum of information since only one or a few runs will define how to optimize the program. Iterative compilation allows multiple runs to test each parameter and even run the program without profiles in order to check the duration of an execution without instrumentation.

1.2.4 Post-compiler Profiling

Though compiler-aided and iterative compilation techniques differ, they have one common point: the compiler. Both methods use the compiler to perform their tasks compared to post-compiler techniques. This latter solution instruments the binary after the compilation and adds code before the execution.

Profiling tools provide information such as memory accesses, data or instruction cache misses, branch information. Some programmers need specific information for specific programs. Instead of rewriting binary instrumentation tools, frameworks have been created to facilitate the work of the programmers [69, 93, 105]. For example, a tool called *PP* [6] instruments SPARC binary executables. PP is built on EEL (Executable Editing Library) [69], which is a C++ library used to edit binary executables. EEL provides an abstraction that allows its user to ignore the executable file format, binary instruction sets or the consequences of deleting, modifying or inserting code. A tool such as PP can examine and modify functions in any order and EEL handles internally the binary code generation of the new executable. Finally, EEL offers two further abstractions: control-flow graphs (CFGs) and instructions. To instrument the code, the tool uses the CFG as a map of the routine and can modify basic blocks, add code segments to blocks and edges or even delete instructions.

The authors show that path profiling can limit the quantity of instrumentation needed and lower the overhead incurred compared to other profiling techniques. The profiling system divides the control flow into different directed acyclic graphs (DAGs). Each DAG is associated to an array that contains a counter for each path existing in the DAG. A free register is set to 0 at the entry of the DAG and incremented or set at certain edges between the basic blocks of the DAG. When leaving the DAG to continue the execution of the program, the register value is used as an index in the counter table. By incrementing this value, the profiler knows at the end of the program which path was the most frequently used. This means that the system has two important properties: to be able to assign a unique number for each path and determine what path was executed when considering a path number.

Whole Program Paths [68] is an extension to this work. Instead of capturing path profiling algorithms, this systems gives a complete record of the entire control flow of the program. As before, the code is divided into DAGs but the different paths are linked together to give a full view of the execution. To limit the size of the profile output, the system uses the Sequitur [87, 86] grammar to compress the transforms the stream of acyclic paths into a context-free grammar.

Post-link optimizers [104, 100, 78] modify the compiled code before execution in order to instrument, profile and optimize programs without the help of a compiler and, compared to fully dynamic schemes, without the need of handling overhead considerations. However, the problems and solutions are often similar, though post-link solution are able to calculate more complex models and insert the corresponding code in the binary. On the other hand, these solutions are not always able to de-optimize in case their optimizations are hurting performance. This means that post-link solutions tend to be less aggressive than full dynamic solutions or rely on profile output files.

For example, ISpike [78] uses a Itanium post-link profiler. Like most Itanium dynamic optimizers, ISpike is based on the *perfmon* API [83]. To perform its optimizations, ISpike uses six profile types: I-cache misses, I-TLB misses, D-cache misses, D-TLB misses, load-miss strides and branches that are provided by the hardware performance monitors. Once the profiles are extracted and studied, ISpike performs different optimizations. For example, it modifies code layout to limit the number of I-cache misses, prefetches instructions to limit the misses at branch

Listing 1.1: Example of a DTrace script

```
syscall::read:entry
{
        self->t = timestamp;
}

syscall::read:return
/self->t/
{
        printf("%d/%d spent %d nsecs in read\n",
                pid, tid, timestamp - self->t);
}
```

instruction. But it also performs data relocation for statically allocated data and inserts prefetch instructions for stride-based memory loads.

1.2.5 Dynamic instrumentation frameworks

Dynamic instrumentation frameworks give a user the possibility to instrument programs during runtime [18, 19, 85, 76]. Some attach themselves to running programs, generally using the *ptrace* system call. These solutions also allow users to define how to instrument the programs by coding little pieces of code in customized languages or sometimes directly in C/C++. In this subsection, three systems *DTrace* [19], *Valgrind* [85] and *Pin* [76], which are more dynamic instrumentation tools than profilers, are presented. Since dynamically instrumenting binaries is often accomplished for profiling, we briefly present them here.

DTrace [19] is dynamic instrumentation system able to instrument both user-level and kernel-level software. DTrace is a framework that enables users to receive information dynamically. Using a C-like language called *D*, a user can, for example, request information on the use of the *malloc* function for a particular process. The information a user can retrieve is obtained by various *probes* provided by the framework. A probe is a location in the code or a particular activity the program performs that *DTrace* can bind a request to execute a set of actions. For example, these actions can record the stack, retrieve a timestamp or copy the arguments passed to a function. Once the request is issued, the framework dynamically modifies the process to install the needed probes and retrieves the requested information. Since instrumentation takes place only when there is a user requesting information, DTrace is able to instrument any program on demand without having to stop the execution and restart the program.

Listing 1.1 shows an example of a DTrace script that uses a thread-local variable to calculate the time spent in the system call *read*. The thread local variable $self \rightarrow t$ is instantiated when a thread requests usage of the $syscall :: read : entry$ probe. At the beginning of each call to the function *read*, the current *timestamp* is collected. At the return of the function, the current timestamp is retrieved. The time spent in the *read* function is calculated by performing a simple subtraction.

The second dynamic instrumentation system presented in this brief overview is Valgrind [85]. Valgrind is a dynamic binary instrumentation (DBA) framework. Valgrind, compared to DTrace, is defined as a core program that a user can complete to create a tool. The tool then executes the target program and instruments the target code one block at a time, in a just-in-time, execution-driven fashion. The binary code is disassembled and translated into an intermediate

Listing 1.2: Example of a Pin tool

```
#include <iostream>
#include "pin.h"
UINT64 icount = 0;
void docount() { icount++; }  // analysis routine

void Instruction(INS ins, void *v) // instrumentation routine
{
  INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}

void Fini(INT32 code, void *v)
{ std::cerr << "Count␣" << icount << endl; }

int main(int argc, char * argv[])
{
  PIN_Init(argc, argv);
  INS_AddInstrumentFunction(Instruction, 0);
  PIN_AddFiniFunction(Fini, 0);
  PIN_StartProgram();
  return 0;
}
```

representation (IR) for the analysis of the tool. Once the tool decides how to instrument the block, the IR is modified and the binary is rewritten taking into account the changes. Valgrind, like Pin [76], Dynamo [5] and Deli [35], does not execute original code but rewrites non-instrumented versions of the code and/or emulates portions of it.

Pin [76] has become one of the references for dynamic profiling. Currently supported by various architectures (X-Scale, IA-32, IA-32E and Itanium processors), Pin gives users a possibility to instrument and profile codes using C/C++ language written tools. Listing 1.2 shows a pin-tool example that counts the number of executed instructions in the program. As we can see, the tool has to initialize the Pin framework. It also adds a pointer into the framework to an instrumentation function that inserts a call to the function *docount* before each instruction. Finally, it asks Pin to call the function *Fini* before the end of the program. If we consider function *docount*, it simply increments the global variable *icount* and function *Fini* displays it.

These three dynamic instrumentation frameworks show that many techniques and variations are possible. The advantage of these techniques compared to the ones previously presented is the potential and freedom given to the user. By allowing a user to write the code in a high-level language, he can develop specific tools for each program or write generic tools for a given set of programs. This freedom gives the user the possibility to completely understand, profile and modify the code at run-time.

1.2.6 Dynamic Software Profiling

Dynamic software profiling, compared to heavyweight or off-line profiling, adds a decisive factor: execution time. Therefore, dynamic profiling needs to incur the lowest overhead possible. Lightweight profilers are different than their heavyweight counterparts. First, they are generally simpler in complexity and generally use a sampling method to limit the time spent instrumenting the code. Second, they are not off-line methods and are generally attached to a specific

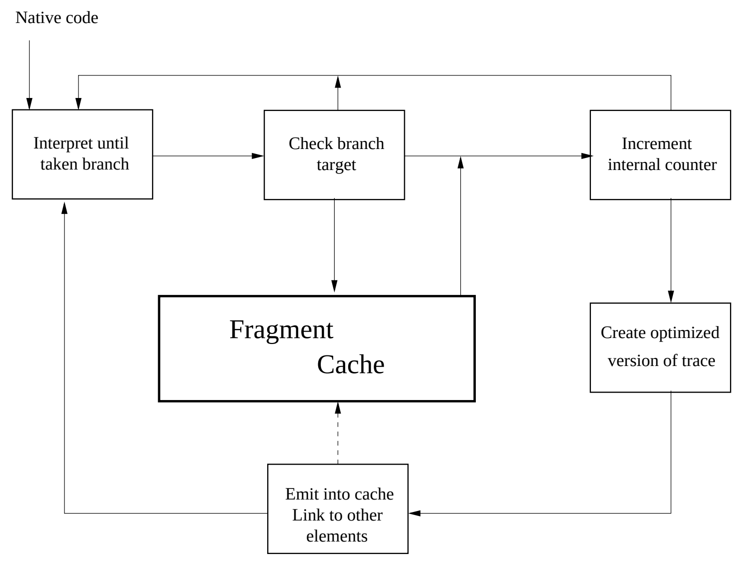

Native code
Interpret until taken branch
Check branch target
Increment internal counter
Fragment Cache
Create optimized version of trace
Emit into cache
Link to other elements

Figure 1.4: The Dynamo Framework

optimization scheme. Finally, certain solutions are totally dynamic, meaning that no prior instrumentation is needed and some solutions use a post-compilation instrumentation technique to insert the instrumentation into the code. The rest of this section presents different dynamic profiling schemes.

The Morph system [118] generates two files during the compilation phase. One file is the binary executable and the second one is an intermediate representation of the program that is used during the optimization phase. Using the operating system as a means to sample the behavior of programs and generate multiple profiles, Morph is able to gather profiles that can be used to specialize the code with relatively low overhead. In a second phase, when CPU activity is low, the profiles are analyzed and specialized code is created and integrated into the programs.

Other solutions are purely dynamic, meaning that the profiler and optimizer are loaded into memory with the program and retain no information about the behavior of the program afterwards [5, 16, 4, 49, 22, 35]. These solutions rely on quickly deciding whether to optimize a region of code or not and being able to assess their modifications.

Hirzel and Chilimbi [49, 22] extended the work of Arnold and Ryder [4]. The solution presented by Hirzel and Chilimbi used Vulcan [104] to modify binary executables. The code was modified in order to duplicate the functions present in the executable. One version of the function contains the instrumentation code to provide memory streams to the optimizer which incurs a high overhead. A second version contains only a few checks to be able to periodically return to the instrumented version. This function switching system is an efficient software solution to calculate stream samples instead of whole streams.

Dynamo [5] and DynamoRIO [16] interpret the code and dynamically create optimized versions of frequently used code fragments. A code fragment is defined as a *one-entry, multiple-exit instruction sequence*. Figure 1.4 shows the dynamo framework. The code is first interpreted until a branch instruction is found. By interpreting the code, Dynamo is able to control the execution of the program. If the branch target corresponds to an entry in the *fragment cache*, the program is redirected to that fragment. Otherwise, a counter associated to the branch target is incremented. Since the fragments contain only one entry-point, adding a counting mechanism to the branch instructions is sufficient to determine frequently used fragments. Dynamo does not use a second thread to maintain control and, at the end of a fragment, the execution returns to interpreter. Since the exit of a fragment leads to another entry point, the system could test to see if the corresponding fragment exists in the cache. However, if the current fragment branches to another portion of optimized code, it is useless to switch to the interpreter and back to the fragment cache. In order to limit the context switch between optimized fragments, they are dynamically linked together when a fragment is created. This means that, if the code remains in this cache, the code is no longer interpreted and the program can be executed at full speed. Since the code is even optimized, speed-ups are achieved. If the exit destination does not exist in the cache, the system directly increments the associated counter.

Ubiquitous Memory Instrospection (or UMI) [121] extends the DynamoRIO framework. The system uses sampling to profile the hot traces. Once DynamoRIO framework has selected hot traces, UMI samples the code and uses a fast cache simulator to determine delinquent loads. To profile correctly the trace, UMI first creates a clone of the trace. This clone enables the system to decide when to profile the trace. By filtering out certain memory operations and keeping potential delinquent loads, the framework is able to lower its overhead. Finally, instrumentation code is added at the end of the trace in order to decide when to profile the trace and when to execute it natively. Using recorded memory references and a cache simulator, the system is able to determine what loads are causing cache misses and UMI can try to optimize them. The overhead of the DynamoRIO system is averaged at 13%, the added UMI overhead raises this number to 14%. Once the delinquent loads are selected, UMI uses a stride prefetcher to accelerate the benchmark programs. To achieve this, the stride used by the different loads are determined by the profiler.

Deli [35], compared to Dynamo, is more a dynamic instrumentation framework than a simple dynamic optimizer or profiler. Using a specific API, a user can modify the default behavior of the DELI system. For example, a user can modify each instruction before it is executed since the system emulates the code before generating code fragments that are saved in a fragment cache. The user can also invalidate fragments or set callback functions. The callback functions are called at different events during the execution. Users can then directly modify the default behavior of the DELI framework to suit their needs.

Another dynamic optimizer is the Adore (ADaptive Object code RE-optimization) system [73]. Adore uses a second thread to decide what to optimize and perform the code modifications. Implemented on the Itanium processor, the system uses the hardware performance counters to find hot-traces. Using a hybrid phase detector to only optimize the code when the program is in a stable phase, Adore is able to optimize while maintaining a low-overhead. A stable phase is determined when the program remains in the same code region (using the *Program Counter register*), has a constant number of cache misses and number of cycles per instruction. More recent work, for example from Das et al. [32], defines this as a global stable phase detector and determines that calculating a stable phase for each code region gives better results. Since Dynamo interprets the code, it needs to optimize and send code fragments into the cache as fast as possible. Compared to the Dynamo framework, the overhead incurred by Adore is lighter and this means that less code needs to be optimized.

The profiling systems of these dynamic systems all have one thing in common: they are lightweight. In order to achieve the lowest overhead possible, sampling techniques, fragment caching, dynamic inlining and other optimizations are used. Once the overhead of the system is low enough, optimization techniques can be linked to the profiler. One of the most common data optimizations is data prefetching. The next section presents an overview of the different prefetching techniques that have been studied and implemented.

Caches	Information	Itanium	Itanium II
	Total size	16 Kbytes	16 Kbytes
L1 Instruction	Line size	32 bytes	64 bytes
(Internal)	Latency	2 cycles	1 cycle
	Size	16 Kbytes	16 Kbytes
L1 Data	Line size	32 bytes	64 bytes
(Internal)	Latency	2 cycles	1 cycle
	Size	96 Kbytes	256 Kbytes
L2 Data	Line size	64 bytes	128 bytes
(Internal)	Latency	6, 9+ cycles	5, 7 or 9+ cycles
	Size	2 - 4 Mbytes	6 Mbytes
L3 Data	Line size	64 bytes	128 bytes
(External)	Latency	21+ cycles	12+ cycles

Table 1.1: Cache level comparison between the Itanium and the Itanium-II processors

1.3 Prefetching

When a program is executed, it accesses data from memory. With the improvements in processor clock speeds, the importance of reducing memory stalls is becoming more and more important. To minimize the number of accesses in memory, modern processors copy the data into small caches in the processor. The number of caches available depends of the processor. Modern processors contain at least one small internal cache, considered as the first level and divided into two groups: instruction and data caches. The instruction cache contains the currently executed program instructions. Since most programs contain loops or recursive functions, the processor can execute the code without having to fetch the next instruction from memory. The internal data cache is the smallest but fastest data cache of the processor, except if we consider the different registers of the processor as a data cache. It only contains the current data and relies on bigger but slower external caches to handle less used data. Table 1.1 compares the different sizes and latencies for each cache level between the Itanium and Itanium-II processors. As we can see, the Itanium-II, while offering bigger cache levels, also reduces the average latencies.

When a data request is performed, the processor checks the caches first. The first cache level has the smallest size but is also the most efficient in terms of number of cycles to fulfill the check. However, if the data is not present, the processor looks for the data at the second level and so on. If the data is not available in the different cache levels, a *cache-miss* occurs and the processor retrieves the data from memory. During the retrieval and depending on the content of the cache levels and the replacement policy, the cache levels can also copy the data for a future reuse.

Prefetching consists in requesting data before the program accesses it. This ensures the presence in cache when the data is needed. To determine what loads are considered delinquent and what data should be prefetched, optimizers use profilers such as those presented in the section 1.2. This section presents the different techniques used to prefetch data once the delinquent loads have been detected.

Prefetching data generally consists in using a hardware component or using an instruction from the processor. Three factors are important when considering prefetching data: prefetch distance, prediction accuracy, outstanding prefetches. The prefetch distance and prediction accuracy generally depends on the information provided by the profiler. The outstanding prefetches is the number of prefetches that are not yet finished. Generally, a processor starts ignoring prefetch requests when the number of outstanding prefetches is too important.

The rest of this section presents different prefetching schemes. The reader may refer to [38, 109] for a survey of data prefetch algorithms. Generally, prefetchers can be described as stride, correlated or thread prefetchers. Stride prefetchers are the simplest prefetchers since very little information is required and they can start prefetching very early. They are useful for programs with regular stride memory accesses. They are easy to implement and the overhead incurred is low. For example, in the case of regular loops with regular array data accesses, a stride prefetcher is sufficient to achieve a speed up and optimize correctly the target program.

Correlating prefetching techniques are useful to programs with linked-list data structures or more irregular data structures. By calculating correlations between data memory addresses and even between different load instructions, the predictors are able to predict more accurately and handle cases where even a complex stride predictor is not efficient.

Finally, thread prefetching can be considered as a third class of prefetchers. Without studying the memory behavior of a hot trace, thread prefetchers strip nested loop code of calculations that do not contribute to the calculation of the next load address. By doing this and by executing on a second core (or processor), the prefetching thread is able to calculate in advance the data that the program will be using.

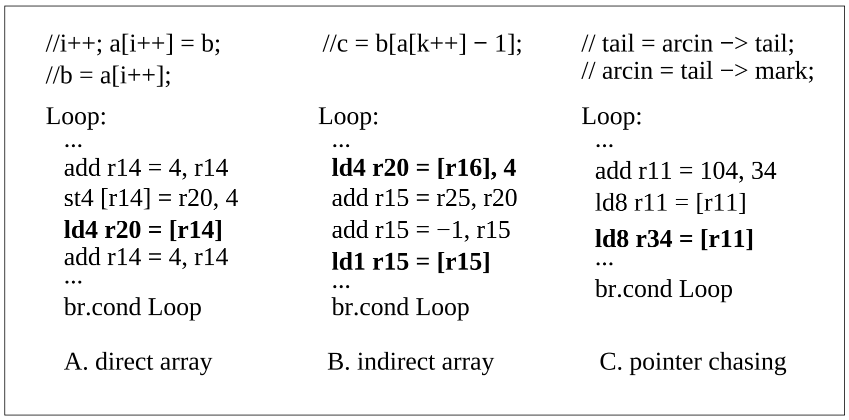
1.3.1 Stride Prefetching

When handling static data structures, the compiler can often insert prefetch instructions into the generated code [84, 57]. However, when considering complex traversals or dynamically allocated structures, the compiler cannot determine the actual memory access behavior of the program. Sometimes, the memory behavior can be represented by a simple stride, meaning that the difference between two consecutive addresses remains constant. Many optimizers and prefetchers have been implemented to handle these cases, some being more general than others. This subsection presents a few of these methods.

Probably the first stride prefetching scheme was the next-sequential predictors that prefetch the next cache line when a miss occurs [99, 20]. For example, if an access to a the cache line L results in a cache miss, the predictor prefetches the data that would be copied into the line $L + 1$.

As a hardware/software hybrid solution, ISpike [78] uses the hardware profiling system to detect the delinquent loads and retrieve samples of the memory addresses that are accessed by that load. Using this information, the system is able to select the loads that will be prefetched with a particular stride. If there are multiple strides detected for a single load, ISpike prefetches the most frequent two of them. Other techniques also are implemented on the Itanium processors or extend the architecture [73, 63]. For example, Kim et al. [63] add a few registers to create *informative loads* that provide information to hardware components. These hardware components can then be polled and used to define what prefetches should be executed. By creating two new prefetch instructions, the compiler can insert these special prefetches into the target programs. Using the information provided by the loads, the prefetch instruction are executed or not at run-time.

More complex solutions characterize the loads into different groups and handle the prefetching accordingly. For example, the Adore framework [73] concentrates its optimizations to loops containing stride based loads. By studying the code surrounding a load, the system characterizes the loads into three different patterns: direct array loads, indirect array loads and pointer chasing



```
//i++; a[i++] = b;
//b = a[i++];

Loop:
  ...
  add r14 = 4, r14
  st4 [r14] = r20, 4
  ld4 r20 = [r14]
  add r14 = 4, r14
  ...
  br.cond Loop

  A. direct array
```

```
//c = b[a[k++] − 1];

Loop:
  ...
  ld4 r20 = [r16], 4
  add r15 = r25, r20
  add r15 = −1, r15
  ld1 r15 = [r15]
  ...
  br.cond Loop

  B. indirect array
```

```
// tail = arcin −> tail;
// arcin = tail −> mark;

Loop:
  ...
  add r11 = 104, 34
  ld8 r11 = [r11]
  ld8 r34 = [r11]
  ...
  br.cond Loop

  C. pointer chasing
```

Figure 1.5: Data reference patterns detected by the Adore system

loads. Direct array loads represent a simple array accesses. Indirect array loads is an array access using a second array as an index array. Finally, a pointer chasing load is always characterized by the link-style structure it is attached to. Generally, a *next* pointer is included in the structure. More complex structures have multiple *next* pointers but, for simplicity, we will consider simple linked-lists. The pointer chasing code first loads the first element and then, iteratively, adds the offset to point to the *next* pointer field of the structure.

The instrumentation of the load is determined by its group. Figure 1.5 shows the different load patterns that the Adore framework handles. Delinquent loads are written using a bold font. For example, direct array loads contain a fixed addition (or subtraction) to the current address when calculating the next address. In this example, we notice that the value 4 is added to the base address three times between two executions of the load. Multiplying this number by a factor and adding it to the current address gives the system the possibility to prefetch elements in advance. For indirect array loads, the framework prefetches both data streams. Finally, for pointer chasing loads, the stride between the last two elements is calculated and multiplied by a factor. Stoutchinin et al. [106] showed that this works on link-list data structures but is not efficient for more general graph or tree traversals. More general work on linked-list data structures has also been done [24, 77] relying on the compiler to insert prefetches to the next nodes of the list, graph or tree as soon as the current node has been calculated or reallocating the data structure to render the traversal linear. If done correctly, the memory access behavior can be predicted using a simple stride. Roth and Sohi [95] present another solution for prefetching link-lists by adding jump pointers. These pointers, for example, link the current node to its grandson. This means that, instead of only being able to prefetch the next node, the jump pointer gives the chance to prefetch nodes further down the linked-list.

Other group categorizations are possible. Wu [117] presents one possibility by defining loads by their memory access behavior and not by their nature. This solution categorizes loads into 4 different groups:

1. *Strong single stride load*: the stride of the load is generally the same throughout the execution;

2. *Phased multi-stride load*: the stride of the load is not always the same but the different strides occur one after another and each stride a certain number of times. For example, 12 12 12 32 32 32 32 12 12 12 is considered as being a *phased multi-stride load*;

Type of stride	**Prefetch code**
Strong single stride	prefetch(P + K * S)

Phased multi-stride	stride = P - oldP
	oldP = P
	prefetch(P+ K* *stride*)

Weak single stride	stride = P - oldP
	oldP = P
	p = (stride == profiled stride)
	p?prefetch(P + K*stride)

General loads	No prefetching

Table 1.2: Prefetch code depending on the memory behavior of the load

3. *Weak single stride load*: the stride of the load is not always the same but one stride is more frequent than the others;

4. *General loads*: the remaining loads.

Since a stride prefetching technique is used, the last group is not optimized. However, the system does not need to optimize each load since the compiler can, by comparing the range of the different addresses and the configuration of the cache lines, prefetch effectively multiple loads with a single prefetch instruction. The stride of the load and the prefetch distance are calculated during the profiling. Instead of using the number of cycles in each iteration, the calculation relies on dividing the number of the *trip count* with the fixed threshold. The trip count is the ratio between the entering frequency to the entry basic block of the loop from inside the loop and the entering frequency from outside the loop. This can also be considered as an average number of consecutive iterations of the loop. The trip count threshold is the minimum value for a loop to be considered for optimization.

Finally, since the framework handles different types of stride-loads, three different prefetching codes are integrated into the program. Table 1.2 summarizes the different solutions for each load group. For a *strong single stride* load, P is the current address, S is the profiled stride and K is calculated as follows :

$$K = min(\lceil trip\_count/TT \rceil, C)$$

where TT is the $trip\_count$ threshold and C is the maximum distance accepted. For a *phased multi-stride* load, the optimizer must first know what the current stride is. This calculation is performed by storing the last address and calculating the current stride. K is calculated the same way but is rounded to a power of 2 in order to limit the overhead incurred by the prediction calculation. Finally, for a *weak single stride* load, a test must be added before prefetching since the optimizer only prefetches if the current stride is the most frequent one profiled.

Though stride prefetching can be considered as the most simple technique to calculate predictions, multi-stride predictors exist. For example, a simple stride prefetcher, after having noticed misses at $P, P + 32, P + 64$, would predict $P + 96, P + 128$. But if the sequence of memory accesses was actually: $P, P + 32, P + 64, P + 100, P + 200, P + 300, P + 332, P + 364$ and so on, the predictor would not be able to remain accurate. Multi-stride predictors are able to extract the information needed to modelize this. If we rewrite the sequence in a form of successive strides, we obtain: $0, 32, 32, 36, 100, 100, 32, 32$ and we would predict that the next strides would be $36, 100, 100, 32, 32$ and so on. Iacobovici et al. [52] use a hardware table to learn multi-strides and prefetch the data. Though the table is constructed to handle multi-stride streams, the system also predicts and prefetches single stride streams.

Zhang et al. [120] proposed an extension to their Trident system [119] by adding a self-repairing prefetching mechanism. The system adds a *Delinquent Table* (DLT) to the hardware. The system distinguishes two groups of loads called *Stride* and *Pointer* loads. For each Stride load, the other loads of the hot trace are examined. If another load uses the same live base register, the loads are grouped and optimized together. Instead of calculating a prefetch distance for the stride-based loads, the system starts by using a distance of 1 for every load. If the load is still considered delinquent, the distance is incremented.

Finally, if a delinquent load cannot be prefetched because it is not a stride or a pointer load or if it remains delinquent after testing different prefetch distances, the load is marked *mature* in the DLT using a flag. In this system, a mature load is no longer monitored by the hardware and the mature flag is never set back to 0. However, if the line containing the information of the load is replaced by another load, the hardware will monitor the load the next time it is added to the table. A simple flush of old mature loads could be added to periodically re-monitor the loads.

More complex stride predictors handle multiple streams at the same time. Al-Sukhni et al. [3] handle multiple streams and define one *Prefetching engine* per load. Since the system cannot allocate as many engines as there are loads in the program, it uses a confidence level system to recycle the engines with other loads. Using an affinity level to group different loads that show similar behavior, the system is also able to use the same engine for multiple loads. Finally, a density level is calculated for each load. This level can be defined as the ratio of loads performed by a particular load to the total number of loads performed by the program. A high density means that the load is frequently used compared to the other current loads and should be handled by an engine. The system uses a finite state machine to handle the confidence level of each prefetching engine and, when the confidence level is low, this means that the load handled by the prefetch engine is either poorly prefetched or has a low density and thus can be replaced by a higher density load.

Stride predictors, though at first simple and inaccurate in more general behaviors, have become more and more complex. Using multi-strided analysis and including techniques to correctly calculate the prefetch distance and the current stride, these systems are able to handle a good number of structure traversals. However, there are more general cases where even the most complex stride systems are unable to be efficient. In these cases, the stream must be carefully analyzed and correctly modelized. Correlated prefetching is a possible solution and the next subsection presents those systems in greater detail.

1.3.2 Correlated Prefetching

Correlated prefetching does not necessarily mean complex data structures and calculations. Collins et al. [26] present a simple correlated prefetching technique that can be used to remember pointer loads. A pointer load can be defined as a load of a memory address in the heap from a memory address in the heap. A pointer cache can be integrated into the processor to associate heap pointers to the objects in the heap they point to. When the L1 cache is queried, a simultaneous query is performed and the data can be sent from the pointer cache to the L1 cache. Though the system updates the pointer cache at each store of the program, a memory address provided by the pointer cache is still considered as speculative.

Another important aspect is the technique adopted to send the data to the processor. Prefetchers can directly use the cache of the processor or send the data to separate buffers that are queried at the same time as the normal caches. Jouppi [56] proposed a stream buffer that contained the predicted addresses of the considered stream. Once a sequence of addresses is contained in the buffer, the system then waits for the first element of the sequence before prefetching the data from the second cache level into the smaller first level. This implementation needs the current miss to correspond to the first element of a buffer before predictions are possible. Farkas et al. resolved this problem by adding a check to the other elements of the buffers. Manku et al. [79] provided another solution that was a mix between stride predictors and stream buffers. In this latter solution, a stride predictor and a stream predictor are used together to prefetch the data. In order to limit the number of prefetches, the stride predictor first checks if the stream buffer already contains the data. If found, the data is pulled out of the stream buffer and the corresponding stream is flushed. This leaves the subsequent prefetches to stride predictor. In general, this means that the stream predictor starts predicting until a frequently used stride is found. Then the stride predictor starts prefetching and the stream buffer can be used for another load.

Though complex systems are not always present in hardware because of the cost and limitations of the architecture, hardware prefetchers can closely be linked to the profiler. Lai and Lu [67] present a hardware pointer data prefetcher. The system distinguishes loads performing linked data structure traversals, called *pointer loads*, from the other loads, called *data loads*. For example, the system uses a table called the Target Register Bitmap (TRB) to detect if a load is a pointer load. Each register is assigned a unique index to the table. A pointer load register can be considered as the possibility that the address contained in the register represents another address. For each instruction, the system updates the table in order to keep track of pointer loads. For example, moving a pointer load register to another register sets the corresponding bit of the destination register to 1. This means that the destination register will be considered as a pointer load register since it is a copy of a pointer load register.

Another hardware component is called the address cache (AC) and keeps a history of past addresses. By updating the TRB and, once pointer loads are detected, quickly accessing the AC, the system is able to predict and prefetch the next elements in the data linked structure.

A more complex predictor is the Markovian predictor. Using previous elements to construct a statistical model, Markov predictors can accurately predict complex memory access behavior. Joseph and Grunwald [55] presented a hardware solution that consisted in adding a data/instruction prefetch buffer and a data/instruction prefetcher. Figure 1.6 shows the augmented processor design. The system adds two prefetching caches as external components of the chip. This processor design uses one internal cache level that is augmented by prefetch buffers. The prefetch buffers are intended to be concurrently accessed with the normal first level caches. This has the advantage of not modifying the behavior of the program if the predictions are not accurate. Markov models generally start with a training phase and, once the model has stabilized, the model can be used for prediction. The work presented by Joseph and Grunwald uses a *continuous* updating system instead.

Figure 1.7 shows the table management for the proposed system. Each miss reference address occupies a line in the table. Since the addressable memory is more important than the number of lines available, conflicts can occur. This means that, at each cache-miss, a test is made

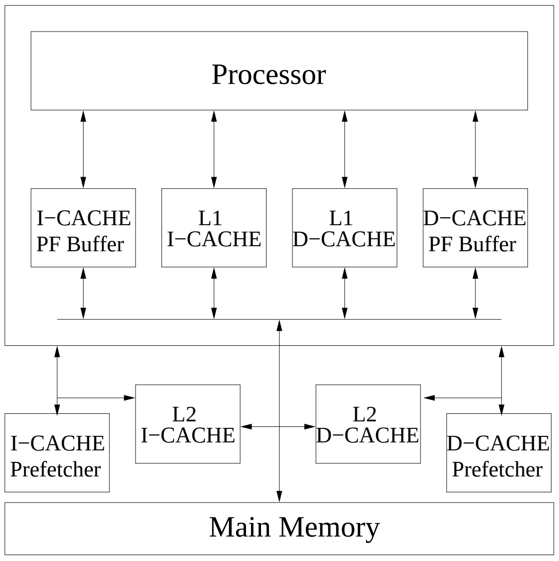


Figure 1.6: Hardware Markov predictor and prefetcher

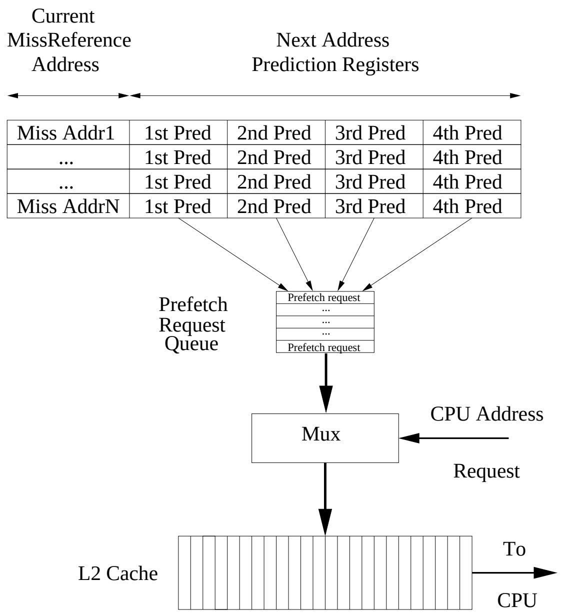


Figure 1.7: The table management system

between the current address miss and the corresponding line entry. If there is a match, 4 predicted addresses become eligible for prefetching. These predictions are sent to the *prefetch request queue* with different priorities. High priorities have a good chance of being prefetched, compared to low priorities that can easily be evicted by better predictions. When the processor requests data, the prefetching system is also queried and predicted elements are sent to the prefetch buffers presented in the figure 1.6. Hur and Lin [51] presented a hardware prefetcher, in which the global system also checks the current workload of the program and modifies the aggressivity of the prefetcher by issuing more or less prefetches at a given time.

Since Markovian models are relatively complex to build and use, some predictors use a stride-filtered Markov predictor [98]. The system first uses a stride predictor. If a load is not stride based, the Markovian predictor is used to predict the next accesses. Furthermore, the system uses stream buffers to predict in advance different streams of data. Using at the same time the stride predictor and the Markovian predictor, the system is able to directly take advantage of both predictors.

The problem with correlated predictors lies in the fact that each element is processed and finds its way into the prediction table. This generally pollutes the tables with noise and reduces the accuracy and efficiency of the models. A solution to this problem is to add a filter to the predictor. For example, Sharma et al. [97] use a technique called *Spectral prefetching* that identifies patterns and ignores random elements that can pollute the global stream of memory accesses. Using this technique, the predictor is then able to prefetch only repeating elements of the pattern.

The system divides the memory address space into tag concentration zones (TCzones) by using the lower order bits of the addresses as indexes to the zones. For example, if the lower bits of an address are equal to 5, the address is in the TCzone 5. This means that the number of TCzones depends on the number of lower order bits used to partition the address space.

The system uses an analysis phase where the data cache miss sequence is studied and processed. Once enough information is gathered, the analyzer passes the pattern to the correlator. This starts the second stage named the *update* stage. The correlator updates its internal array to contain the current predicted patterns.

The system actually goes from absolute address pattern to differential pattern recognition. It is possible, for example, that during very large memory structure traversals, there are too many elements for a predictor to be efficient and accurate. Using strides is a solution to lighten the model enough in order to detect a pattern. In order to modelize correctly different streams, the analyzer alternates the processing of the sequence. This gives the system a better chance to find a pattern, either in absolute form or in differential form.

Correlated prefetchers are more complex than stride prefetchers. However, they are able to handle general streams and extract the information needed to predict accurately. Because they are more complex, hash tables or sampling mechanisms are used to prepare the model and let it adapt. The next section presents thread prefetching that allow a second thread to execute a simplified version of the code in order to prefetch the data.

1.3.3 Thread prefetching

Stride predictors or more complex predictors all include prefetch instructions in order to accelerate a program. Another solution uses a helper thread to eliminate delinquent loads [61, 71, 113, 27, 112]. While the main thread is executing a heavy cache miss code region, a simplified version of the code is executed by a second thread. If this is done on a multi-threaded processor, the helper thread can be executed in advance. The data the helper thread uses is stored into the shared cache in order to reduce the number the cache misses of the main thread.

Using the Intel Pentium 4 processor with Hyper-Threading Technology [48, 81, 29], two logical processors are available simultaneously. This means that, while the main program is being executed, a helper thread can attempt to optimize the program. Kim et al. [61] present a solution using the Pentium 4 processor. Using the VTune program analyzer [30] to collect clock cycles and L2 cache misses, the system is able to identify delinquent loads. By analyzing the profiles, the compiler is able to identify delinquent loads included in loops.

Once a loop is designated as containing delinquent loads, the construction code of a helper thread is inserted. By removing code that is not necessary for the helper thread, the second thread can be executed faster. To be efficient, the remaining code consists only of the calculation of the addresses used by the delinquent loads. For example, every store to heap objects or to global variables are removed.

In addition to the construction of the helper threads, the system uses a lightweight monitoring system in order to monitor the performance of the helper threads. This dynamic profiling system enables the optimizer to decide at run-time whether to use the helper thread or not.

The previous system used a modified Intel compiler to insert code that would trigger the helper thread. Lu and al. [75] present a totally dynamic optimizer using many of the Adore [78] ideas. First, the system is loaded into memory using the *LD_PRELOAD* environment variable. Second, a stable phase and hot spot detector schemes are used before trying to optimize the code.

The system allocates multiple helper threads and the tasks of each thread are contained in a *dispatch table*. This table contains the function pointers to the different tasks and each thread is given a unique ID entry to the table. This has one major advantage compared to simply giving the function pointer to the thread: if the optimizer decides a task is hurting performance or if another task would yield a better speed-up, the pointer can simply be modified to point to a new task. Finally, once the threads are created, the system uses prefetches instead of directly loading the data with the helper thread.

1.4 Dynamic Frameworks

The previous sections have presented the different profiling mechanisms used by various hardware and software systems. Compilers can be asked to generate an instrumented version of a program. Some use a binary editor to directly instrument the binary post-link in order to modify the generated code and perform post-link optimizations. Others attach themselves to running programs, dynamically modify them and then restore the original code or create new optimized versions of the code once the profiles are retrieved. This last technique can be divided into two groups:

- The dynamic instrumentation frameworks such as Valgrind, DELI, DTrace and others;
- The lightweight profiling systems that are closely linked to the optimization systems they are integrated with.

This section presents in a more detailed fashion these dynamic frameworks. Whether they are instrumentation tools that incur high overheads but provide users with ways to perfectly define their needs or whether they are low overhead automated optimization frameworks, different techniques are used to implement them and enable them to handle the target programs correctly.

1.4.1 Startup

The ability to call a function of the optimizer and initialize it is one the first big issues when implementing a dynamic framework. Some systems modify the binary before the execution to call an entry-point of their system at the start of the execution [49, 22, 78]. This technique gives the system a chance to do some modifications of the binary off-line.

Other systems use the environment variable $LD\_PRELOAD$ which loads a dynamic shared library at the startup of a program [73, 74, 16, 121]. This has the advantage of being simple to install and the target program does not need to be modified in any way. Once the library is loaded into memory, these techniques generally implement the C-library entry point function called *__libc_start_main*. By loading the library and implementing this function, the dynamic frameworks ensure that their system will be called before the *main* function of the target program. The framework has then a chance to instrument the target program, emulate it and/or optimize it.

Finally, the Unix *Ptrace* library gives a dynamic framework the possibility to attach itself to a running program, instrument it and optimize it. Once the system obtains control over the target program, it can either keep it to supervise its optimizations, or detach itself later on. For example, the Pin framework [76] uses this solution to attach itself to the target programs.

1.4.2 Code execution

Once the dynamic framework is running, it needs memory space to store internal counters and optimized code. Certain systems modify the code by using non-conditional jumps to go from the original program to the optimized version [49, 22, 73, 16, 5]. Once the optimized code has been executed, the control flow returns to the original program.

Other systems emulate the code and, when hot traces are found, the traces are optimized and sent to a specific code cache [76, 35, 36]. These techniques have certain advantages on the first technique. First, the control of the program is completely handled since the emulation of the code keeps the control. There is little or no risk that the control of the application is lost. Second, using an internal representation of the code when optimizing the code, different front-ends can be used for different architectures. Resembling modern compiler internal structures, the front-end is the only piece of software that needs to be modified when porting the framework to a new architecture.

1.4.3 Code cache management

When rewriting the code and dynamically optimizing it, a memory location must be found to contain the new code version. If the framework emulates the code, then the code cache contains the code that is executed without emulation in order to limit the overhead. However, if the code executes the original code and then optimizes hot traces, it needs to insert code into the function. There is however a risk that the system overwrites the following function since binary executables tend to put one function after another in the executable file. The adopted solution generally consists in allocating a memory block to contain all the recent optimization codes. Code cache management systems are implemented to replace certain old optimizations by new systems.

Dynamo [5] flushes the code cache when multiple new optimized traces appear in the code cache. The system inserts counters at the beginning of each *one-entry-multiple-exit* instruction trace. When the associate counter is equal to a fixed threshold, the system optimizes the code and sends it to the code cache. Dynamo considers that, if a certain amount of new traces appear in the code cache, a new program phase has been detected. This means that the current traces in the code cache will no longer be useful and the system flushes the whole cache. Though this means that certain of the new traces might be flushed as well, they will be reinstrumented later on. The rationale is that a complete flush is very efficient in terms of overhead and outweighs the risk of reinstrumenting certain traces. Finally, if the code cache is full, a complete flush is performed.

A full flush is a practical *quick and dirty* solution to free cache space for new optimized traces. However, many traces are reinstrumented and sent back into the code cache using this technique. More elaborate techniques such as *Least Recently Created* or *Least Recently Accessed* can be used to limit the number of traces that are flushed and then sent back into the code cache. A comparison of these techniques was analyzed in [47].

Certain frameworks currently allow users to handle and modify the default behavior of the code cache management. Hazelwood and Cohn [46] extend the Pin framework [76] by providing more than 25 functions that can gather statistics about the code cache, test and modify the contents of the code cache. The extension also implements a certain number of callbacks that a user can define to handle different actions. For example, a user can rewrite the code cache replacement system.

Finally, an interesting contribution is a thread-shared software code cache [17]. Many code caches are not thread-safe and the simplest solution is to resort to thread-private code caches. The biggest problem with a thread-private code cache is scalability since, when using many threads, the total size of the code caches is too big. The presented solution is built on DynamoRIO [16] and uses different synchronization methods in order to maintain coherency between the different threads.

1.5 Conclusion

There are numerous solutions to profile, optimize and accelerate target programs. Depending on what the user wants and the technical possibilities, different solutions are available. Through a brief outline of different profilers, this chapter has shown that there exist different levels of detail or information that a profiler can gather. Of course, this level of information comes with a cost and compromises must be found depending on the different priorities. Users, without a need to quickly decide what optimization is useful, can use tools such as *Valgrind* or *Pin*, use the compiler to write an instrumented version of the code or go through an iterative compilation in order to generate an optimized binary.

Prefetchers are a common solution to optimize programs. These prefetchers can be instruction or data prefetchers. We have summarized the prefetching solutions that have been implemented using hybrid, software and hybrid techniques. Prefetching systems can be characterized in three domains: stride, correlated and thread prefetching. Again, depending on the goals and of the ressources available to the programmer or the system, compromises must be made in order to correctly choose between accuracy and efficiency in terms of overhead of the global prefetcher.

Finally, we presented the main issues when dealing with dynamic optimizers. Different strategies have been adopted to startup the frameworks, to gather free registers and to handle their code managements.

The work of this thesis starts by studying the implementation of a lightweight Dynamic Optimization System. Compared to most dynamic frameworks that consider and instrument basic blocks of code, the proposed system is load-based and considers each load independently. The system is totally software and only uses a clock register and a prefetch instruction, elements that exist on most modern processors. Finally, our system handles each load in a unified manner compared to other dynamic systems that classify loads in order to use certain optimization techniques. Though our solution uses the same prefetching technique for each load, it helps to reduce the complexity of the overall system while still achieving significant speed-ups.

The second contribution of this thesis is the implementation of a Markovian-like model called Esodyp that is as lightweight as possible. Compared to other Markovian systems, Esodyp can use N elements of the past in order to calculate the predictions. Furthermore, the solution issues predictions with almost no calculations using pre-calculated values. Though Esodyp was implemented to handle memory accesses, an extension called Esodyp+ abstracts the data given to the model, letting the user decide the internal representation of the data.

Finally, the Esodyp model is integrated into a dynamic optimization framework in order to decide what loads should be optimized. In order to use a complex model in a dynamic setting, a code abstraction technique is used. This abstraction reduces the amount of used memory but, with the help of Esodyp, limits the number of instrumented loads. This final version uses every element presented in this chapter. It starts by using a lightweight profiling method to decide what loads are delinquent. Once the loads are selected, the Esodyp predictor is used to check if a stride-predictor can be used to accelerate the program. This system shows that a data driven system can achieve speed-ups that are above more general optimization frameworks since the system was specialized in data cache optimizations.

Chapter 2

Performance Driven Data Cache Prefetching in a Dynamic Software Optimization System

The previous chapter showed various systems that profiled, instrumented code and prefetched data in order to optimize a program. This chapter presents an automatic and dynamic framework that instruments the binary code in order to detect delinquent memory accesses. Once a delinquent load is selected, a memory access stride is calculated and a prefetch distance is tested. If the combination of both elements leads to a lower latency of the load, the system inserts a prefetch instruction. However, in the case this worsens the latency, the load is de-instrumented and executed without instrumentation to limit the overhead. Finally, if the load is non-delinquent, it is also de-instrumented. Later, these loads are re-monitored in order to check if, at that point in time, the system can optimize them.

2.1 Introduction

In the previous chapter, we have presented software and hardware prefetching systems that contribute in hiding cache miss latencies [22, 60, 73, 74, 103, 122]. A predictor uses a prefetch distance to request the data sufficiently in advance to be in the data cache when the processor requires it. Hence a calculation is needed to decide what distance should be used for each prefetch instruction in order to maximize the positive impact of the prefetch.

This chapter presents a lightweight profiling strategy which does not use any specific hardware or any processor performance counter. Our dynamic framework is entirely software and aims to be portable across any modern processor architecture. This means that, though the core of the framework is dependent on the binary code used by the processor, the overall system can be easily reused on various architectures. The profiling mechanism considers each load independently. Moreover, instead of using code interpretation as it is done in many other systems [5, 35, 76], our approach is based on continuously instrumenting/de-instrumenting the native code with some monitoring or optimizing instructions. Once the number of executions of a load has exceeded a certain threshold, the load instruction is no longer instrumented, allowing it to run without any profiling overhead. If the behavior is relatively constant, then the load is not checked very often, letting the program execute it without any instrumentation. However, if it is more erratic, the profiler checks its latency more often. This solution helps to keep the overhead of the system low, while keeping a fairly good view of the general behavior of the program.

To illustrate the use of the lightweight profiling system, we implemented a simple stride-based prefetching system. The general framework is portable assuming there is at least a prefetch instruction provided by the target architecture. Since every load is monitored periodically, a prefetch instruction is associated to every load whose latency is above a certain threshold. The effectiveness of the optimization is ensured by comparing the latency between the prefetched load latency and the original latency. In order to correctly handle the prefetch distance, the optimizer dynamically tests different distances and, with the results provided by the profiler, adjusts it to the best value.

Since the number of existing prefetching systems is important, we will concentrate this introduction to dynamic optimization systems. Specific prefetching algorithms have been evaluated in numerous works proposing either software or hardware implementations. Also, since many details have already been given in chapter 1, this section will briefly present a few well known systems.

Dynamic optimization frameworks such as Dynamo [5] or, more recently, DynamoRIO [16] are transparent systems since the base program does not need to be compiled with specific options. They are dynamic native-to-native optimization systems. Dynamo is provided as a user-mode dynamically linked library. It interprets the base code at runtime searching for hot traces. Using a low threshold to detect hot traces, Dynamo creates an optimized version of the trace and patches the code accordingly. The system is divided into an interpreted code that is not optimized and the hot traces that run natively. To compensate the overhead of the system, Dynamo moves as much code as possible from interpretation to natively run. In a more recent work [35], a framework called Deli is used to build client applications that manipulate or observe running programs. It is shown how a user can define *on the fly* code modifications at low cost.

UMI [121] is a lightweight dynamic system built on DynamoRIO [16] that can be used to profile and optimize programs. DynamoRIO's internal framework discovers hot-traces and sends them to an internal instruction cache. When DynamoRIO selects a trace, UMI creates a clone that can profile the different loads contained in the trace. This instrumentation enables UMI to gather short memory reference profiles to simulate the cache behavior and select the delinquent loads. Using a sampling period and turning the profiling on and off, UMI keeps the overhead at an average of 14%, which is only 1% greater than the base of 13% incurred by DynamoRIO. Finally, UMI integrates a simple stride prefetching solution into the framework to achieve an 11% improvement on the tested benchmarks.

Lu *et al.* [73, 74] implemented the dynamic optimization system called Adore that inserts prefetch instructions into the instruction stream. Using Itanium performance counters and the profiling tool `pfmon`, the system handles hot traces and detects delinquent loads in loop nests. Loads are derived into three categories : direct array, indirect array and pointer chasing loads. By studying the code of the hot trace, Adore decides how to prefetch the required data. In [73], the optimizer uses a compiler option to reserve the registers needed for the inserted code. In [74], a modification of the bundles containing the `alloc` instruction enables the inserted code to use new registers. The prefetch distance is calculated using the average miss latency of the load and the number of cycles spent by one iteration.

Pin [76] is an instrumentation system allowing users to write C/C++ programs called *PinTools*. Pin keeps control over the application by never executing the original code but by executing an instrumented version of each *one-entry, multiple exit traces*. The instrumented version is built by a JIT compiler that can use predefined callbacks from the PinTool to insert calls or inline code that the user has requested.

Chilimbi and Hirzel's framework [22] samples the execution of the program to decide what portions of the code should be optimized. Using Vulcan [104], an executable-editing tool for x86, the framework creates two versions of each function. While both versions contain the original code, one of them also contains the instrumentation code and the other only contains some *checks* to decide whether to instrument again the function or not. Using a global hibernation phase, the overhead is lowered to achieve a speed-up on several SPECint2000 programs. Once a trace is selected and profiled, an analysis is done to find hot data streams. Finally, when the data streams are detected, a finite state machine is created to prefetch the data. The code handling this state machine is then injected dynamically into the program.

Trident [119] is an event-driven multithreaded dynamic optimization framework. Using hardware support to identify hot traces and helper threads to guide and perform optimizations, it generates low overhead. The software side of the framework consumes the hardware generated events and then decides whether to optimize or not the base program. In [120], Trident is extended to dynamically decide what prefetch distance has to be used. Using a *Delinquent Load Table*, the system monitors loads within a hot trace. It counts the number of executions of each load, the number of misses and measures the miss latency. As with Adore [73], stride and pointer loads are distinguished. By continuously monitoring the load latencies, the prefetch distance is adjusted until the loads are no longer delinquent.

Our system includes many aspects that are inspired by all of the above works. It is totally transparent by using the linux entry point *__libc_start_main*. Like Adore, it inserts prefetch instructions in the instruction flow and, like Trident, it uses a variable prefetch distance. But it also differs by being entirely software. The system does not use any of the processor performance counters, hence it is portable to any kind of processor. Moreover it does not handle only loads in loops, but almost every load in the program. For example, as it is shown in section 2.5 with benchmark program `treeadd`, often executed loads as those occurring in recursive functions are also handled. Moreover they are monitored individually, instead of using a hot trace detection scheme. Lastly, the system does not categorize loads into stride or pointer loads but handles all loads in a unified way. Our system can be compared to Chilimbi and Hirzel's system [22]: we add checks to detect delinquent loads but our system lets the code run natively compared to leaving lightweight checks in the code. The hibernation system is also almost identical but our system handles each load independently whereas Chilimbi and Hirzel's solution uses a global hibernation system. However, since our system does not use any interpretation or emulation of the code, our overhead is lower or similar to these different dynamic frameworks. Finally, our solution can probably be implemented in the different dynamic frameworks. However, the fact that they are almost all trace-based renders the implementation of the system load-based more difficult. A solution would be to ask the framework to reinstrument the trace when a change is needed but this would have a significant cost.

Table 2.1 summarizes the differences between the presented methods depending on different aspects of the frameworks. As we can see, the presented systems either use hardware counters or rely on code interpretation to select the regions of code that are optimized. Every other system is trace-based except our system that can be adapted to do trace-base selection with a minimal overhead as it is presented in subsection 2.2.3. However, even when using a finer monitoring grain, our system keeps a low overhead.

Our framework has been implemented for Itanium-2 machines. It involves several dynamic instrumentations of the binary code whose overhead is limited to only 5% on average. On a large set of benchmarks compiled with either `gcc -O3` or `icc -O3`, our system is able to speed

System/ Type	Profiling	Optimization	Execution	Granularity
Dynamo/RIO Software	Interpretation Instrumentation	Trace layout/ Code optim.	Interpretation/ Dynamic linking	Trace
Deli Software	Interpretation Instrumentation	Trace layout/ Code optim.	Interpretation/ Dynamic linking	Trace
Adore Hybrid	Hardware Sampling based	D-cache related optimizations	Dynamic instrumentation	Loop - Trace
Pin Software	Interpretation Instrumentation	Trace layout Code optim.	Interpretation Dynamic linking	Trace
Trident Hybrid	Hardware Sampling based	D-cache related optimizations	Hardware + Dyn. instrumentation	Trace
Our System Software	Instrumentation	D-cache related optimizations	Dynamic instrumentation	Load Instruction

Table 2.1: Summary of the differences between each solution

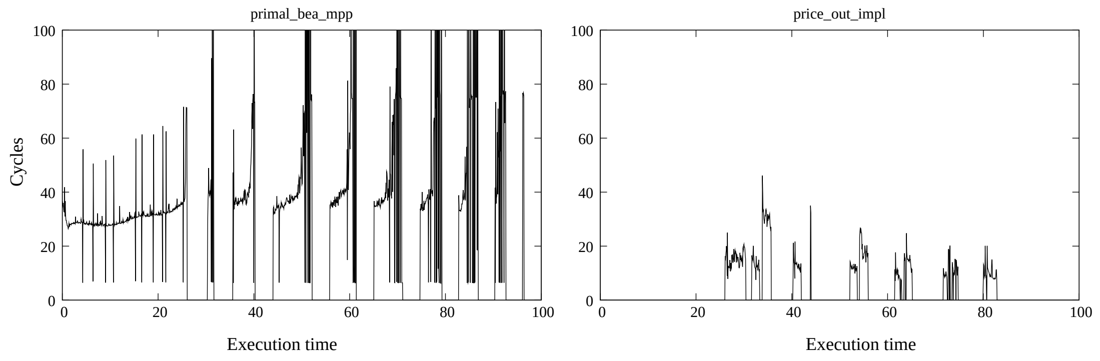


Figure 2.1: Average cycles for a load and the stop instruction in the primal_bea_mpp and price_out_impl functions in 181.mcf (CPU2000)

up some programs by 2%-143%.

The rest of the chapter is organized into five main sections. The algorithm to achieve a lightweight profiling system is presented in section 2.2. In section 2.3, the integration of a stride-based optimization system is explained. Our implementation is presented in section 2.4, where the general profiling framework is presented and technical issues like dynamic register allocation and code modifications are addressed. Finally, experimental and performance results are shown in section 2.5.

The work presented in this chapter was published in [12] and a journal version has been submitted [11].

2.2 Lightweight Profile System

While the program is running, it is necessary to be able to modify the code in order to profile its execution at runtime. The profile system gathers the number of times each load is executed and each average load latency. By gathering this information, the general behavior of the program can be understood and different optimization techniques can be chosen. Figure 2.1 shows the average latency for every load of the `primal_bea_mpp` and `price_out_impl` functions in the CPU2000 `mcf` program. The figure shows that the two functions are periodically called and have a high latency. We also notice that the two functions are almost always called at disjoint execution phases.

When an optimizer needs to know what code traces should be optimized, it relies on a profile of the program. This profiling can be achieved by compiling and executing an instrumented version of the program. By performing a first run of the program, information is extracted and, during a second compilation, specialized optimizations can be integrated. However, in dynamic optimization systems, this is generally done during the execution of the program. This means the optimizer must profile the program, detect hot traces and select likely candidates to the different optimizations schemes at the same time. The main advantages of this latter approach are the transparency to the user and the continuous adaptability to the variations of the program behavior. At the same time, the problems with this technique is the overhead incurred by such a method. Everything that slows down the program must be kept at a minimum since it would render the optimizations useless.

In this section, we present a dynamic profiling solution that does not require a profiling mechanism with two compilation phases. Adore [73] uses Itanium's hardware performance counters to achieve this. Using the hardware counters, Adore is able to keep the general overhead low and, by integrating a data-prefetching system, shows speed-ups on some of the presented benchmarks. Dynamo [5] and DynamoRIO [16] interpret the native code to detect the hot traces that will be optimized. These systems either use the hardware to achieve the profiling or emulate the code to retain control over the application.

Our system tries to be as lightweight as possible but remains entirely software. The profiler inserts a counting system for each load, considering each load independently. Chilimbi and Hirzel [22] have a similar solution but, in their solution, the code is always instrumented with at least a few checks to maintain control over the program execution. Also, the solution adopted by Chilimbi and Hirzel duplicates each function. Our system directly inserts unconditional jumps from the original code to the instrumented version. Though the overhead is obviously high when doing this during the entire execution time, this section shows how to reduce this overhead to an average of 5%.

2.2.1 Dynamic classification of loads

In order to lower the overhead, the system must not constantly profile the program. Instead, it must periodically interrupt the instrumentation of the code for the program to execute without any slow-down. For example, if we consider that a particular load has a constant behavior and its latency also remains constant, the system can stop the profiling and simply keep an average.

This means that, once the load has been executed a certain number of times, the profiler calculates an average and restores the original code. Then, it considers that the load remains at that average. Later, the load is reinstrumented to check if the latency has changed or if it remains the same. To profile and examine the behavior of the different programs, we defined 4 load latency groups that were adjusted experimentally:

- Under 8 cycles: these loads are non delinquent since they have the lowest latency;
- Between 8 and 15 cycles: these loads are also considered non delinquent but should be closely monitored since they might become delinquent if the behavior of the program changes;

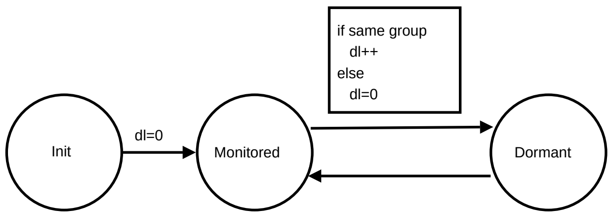


Figure 2.2: Profiler finite state machine representing the states and transitions of a load instruction

- Between 15 and 40 cycles: these loads are considered delinquent;
- Over 40 cycles: these loads are highly delinquent.

This division into four groups gives the profiler a solution to monitor the change of phases for each load. Figure 2.2 shows the finite state machine that decides the state of each load. At first, every instrumented load enters the monitored state and is associated a counter `dl` initialized at 0. This counter represents the confidence level indicating that the load behavior is relatively constant. When the number of executions for a particular load exceeds the threshold, its counter `dl` is incremented if the new calculated latency average does not result in a group change. If there is a change, `dl` is set back to 0. The *base poll time* is the minimum period of time during which the counters are not checked. When the load is restored, it remains dormant during 2^{dl} times the *base poll time*. This means that, during that period, the load is no longer instrumented. As a result, loads that have a constant average are rarely instrumented since the average does not often change. On the other hand, loads that show an erratic load latency behavior and often change groups are often instrumented since the profiler must keep a close watch on these loads.

Table 2.2 shows the number of latency changes for some benchmark programs compiled with the `gcc` and `icc` compilers at the `O3` level on an Itanium-2 machine. The results are averages of 3 runs. The second column of the table represents the total number of times a load changed groups. For example, 29 means that, after the initial group assignation of each load, the system noticed that, 29 times, a certain load was no longer in the same latency group. As we can see, the load instructions in programs `parser`, `ammp` and `mcf` often change groups. The third column shows the number of loads that actually changed from one group to another. For example, `mcf` has on average 135.33 loads that changed groups for a total of 416.67 group changes. This means that on average, each of the 135.33 loads changed 3 times of average latency during the execution of the program. The last column shows the total number of loads that were at least executed once before the last poll of the execution. The percentage of loads that change at least once of latency groups is 9% of the executed loads. Finally, we see that program `treeadd` has a different behavior depending on which compiler is used. Using the `gcc` compiler, the program executes 74 loads whereas only 19 loads are executed using the `icc` compiler. This difference has a direct impact on the other results for this program.

Though table 2.2 shows the total number of latency changes, it is important to consider the number of times each load changes groups. Figure 2.3 shows the percentage of loads that change groups once compared to loads that change more than once. This figure shows that on average only 47% of loads only change groups once. Furthermore, figure 2.4 shows the percentage of loads that change groups twice, 3 to 5 times or more than 5 times out of the number of loads having changed groups at least twice. For each of these three groups, the average percentage is

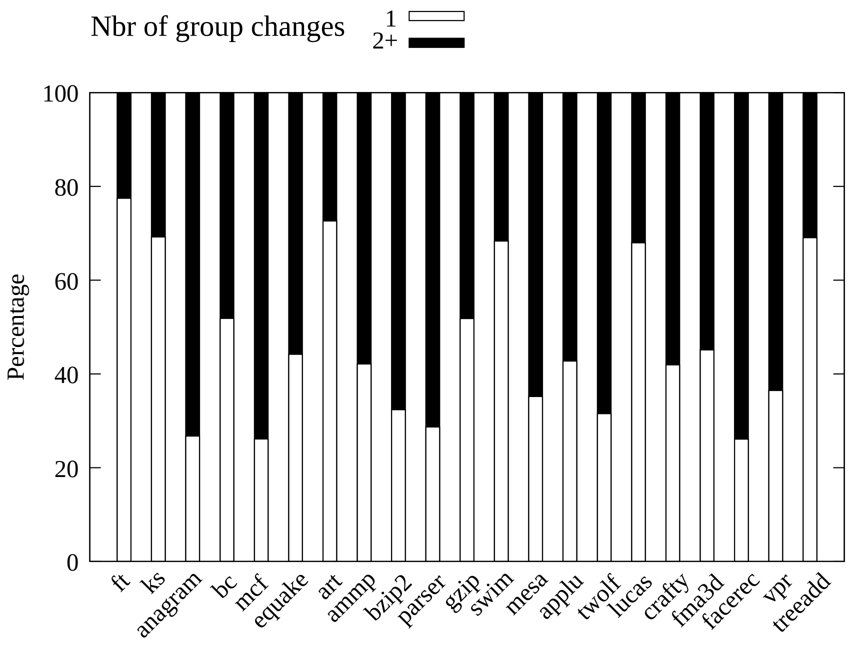


Figure 2.3: Percentages of loads that changed groups once compared to those that changed more than once using the `gcc` compiler

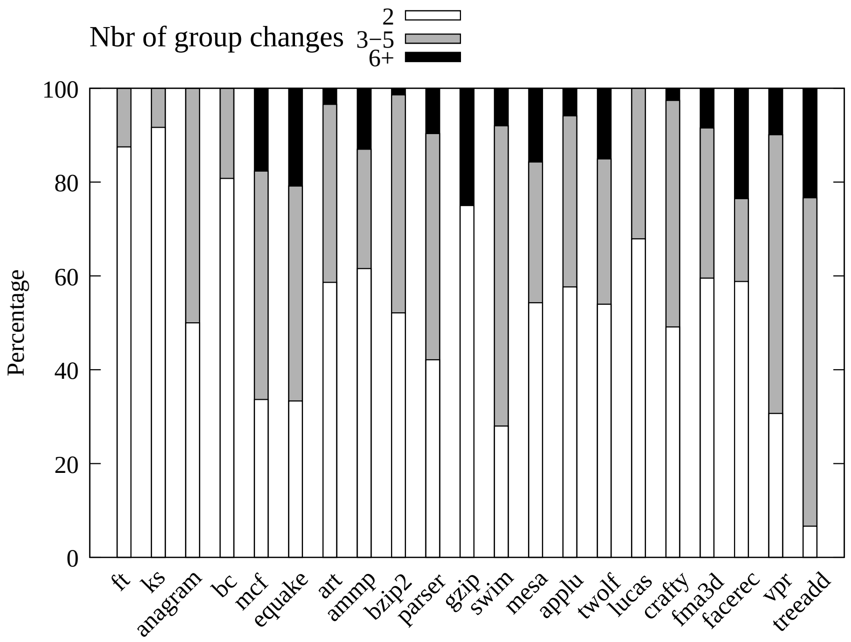


Figure 2.4: The percentages of loads changing groups twice, 3 to 5 times and more than 5 times out of the number of loads having changed groups at least twice using the `gcc` compiler

Benchmark	Latency changes		Loads changing latencies		Executed loads	
	gcc	icc	gcc	icc	gcc	icc
ft	29	35	23.67	22.67	108.67	74.67
ks	17.67	13	13	8.33	192	177.33
anagram	53.33	3.33	23.67	1.67	270	119
bc	28.67	35.67	18	22.33	889	856.67
mcf	416.67	283.67	135.33	91	396	344
equake	36	30.67	14.33	13.67	380	747.33
art	51.33	47.67	35.33	34	327.33	536
ammp	486	458	221.67	178.33	1595	1123
bzip2	77.67	71.67	35	37	750	866.67
parser	3864.33	2240.67	1457.33	727	5017	2545.67
gzip	45	49.33	27.67	26.67	730	827.67
swim	46	116	26.33	67.33	206	1154
mesa	190.33	135.67	72	54	1703	1883
applu	205.67	53	99	32	994	1146.67
twolf	438	507.33	165	232.33	3464.33	4720.33
lucas	126	2.33	84.33	0.67	469.67	858.67
crafty	1158.33	836	557.33	413.67	4180	3646.33
fma3d	420	203.67	201	114	3855	3451
facerec	52	69	8.67	25.33	81	1840
vpr	279.33	267.67	106	101.67	2913	3388
treeadd	38	1	27.33	1	74	19

Table 2.2: Load latency variations: number of latency group changes throughout the execution, number of loads that have changed latency groups at least once and number of loads that were executed at least once.

54%, 38% and 8% respectively. We notice that most benchmarks have a majority of loads that only change groups twice but programs such as `treeadd` or `vpr` have loads that change groups more often.

At each poll, if the number of executions of a particular load is over the threshold, the average latency of the load is calculated. If there is a group change, an associated group counter is incremented. If the counter value is high, this means that, throughout the execution, group changing loads were often tagged as currently belonging to the associated group. Figure 2.5 shows the percentage of each group counter. The proportion of non-delinquent loads seems quite low on the figure. However, most non-delinquent loads remain so throughout the execution of the program, therefore the associated counter remains low. This is especially true for fortan codes such as `swim`, `facerec`, `lucas` and `applu`.

2.2.2 Quality and Overhead of Lightweight Profiling

Though the system does not profile each load during the entire execution, the use of confidence levels helps in keeping a sufficient quality of the profiling method. If during a poll, a load changes state, then the system will rapidly check the latency to see if there is another change. For our experiments, the base poll time was set to 10 milliseconds and the maximum confidence level was set to 10. Thus if a load reaches the maximum level, its latency is checked every

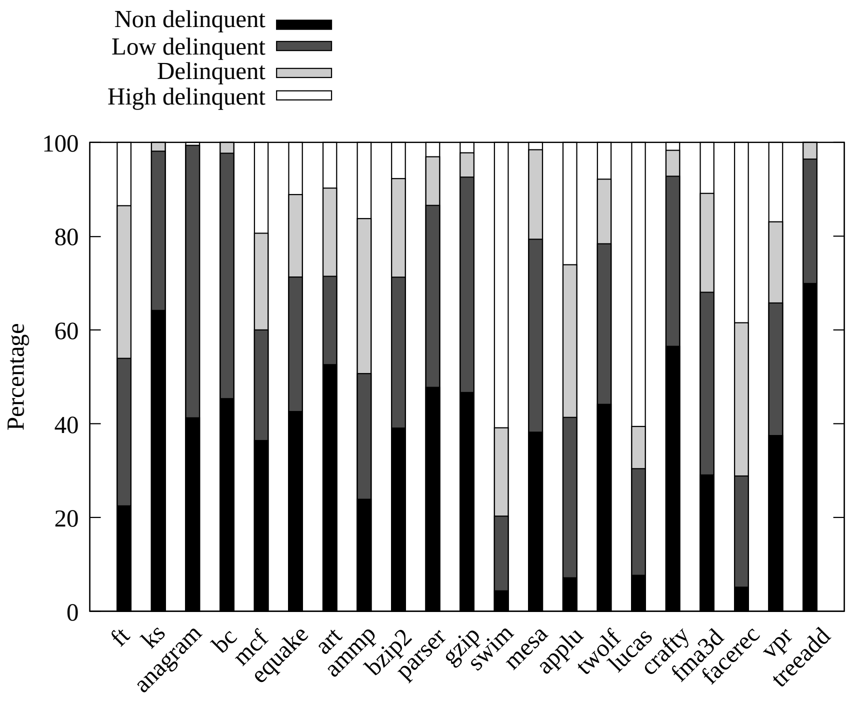


Figure 2.5: The percentage of each destination group for loads that changed groups using the `gcc` compiler

$2^{10} * 10 = 10240ms$.

However to reach such a level, the load would have been checked 9 times and would always have remained in the same state. In the worst case scenario, the load can change its state right after a poll. It would then last approximately 10 seconds for the system to adapt. Though this may seem as a long time to adapt, the code would not be instrumented during that period. This means that, during those 10 seconds, the program is not slowed down. Once the load is reinstrumented, the new average is calculated and taken into account.

The primary advantage of this instrumenting/de-instrumenting technique is the low overhead incurred. Since the program is generally running natively, it is rarely slowed down. Figures 2.6 and 2.7 show the overhead for the considered benchmark programs with the `gcc` and `icc` compilers. The execution times for the original program, the base profiling method and the lightweight profiling system are compared. The original program represents the case where the benchmark program is executed without any instrumentation. The base profiling method represents the case where the code is continuously instrumented during the whole execution. Instrumenting the code continuously results in an overhead of 84% on average. An average of 5% is obtained using the lightweight version which gives the system the chance to optimize the programs.

An interesting case is program `lucas`. In the `icc` case compared to the `gcc` version, the overhead is almost negligible. The `icc` compiler optimizes correctly certain regions of the code that `gcc` is not able to do. Since the delinquent and often executed loads that were generated by `gcc` were removed, the only loads that remain in the `icc` version are not often executed loads, which reduces the overhead of our system.

The second advantage is that the polling speed is specific to each load. Each load is polled

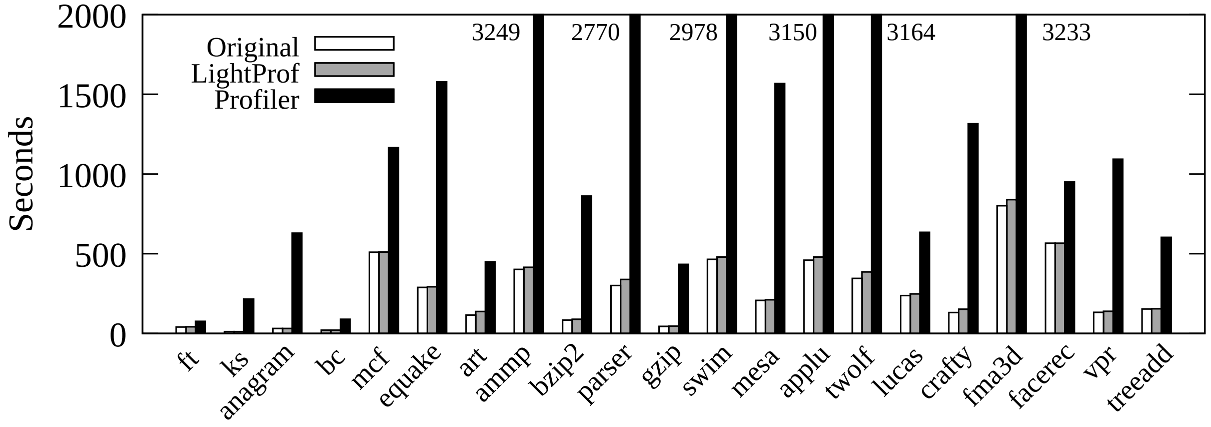


Figure 2.6: Comparison between the program without any instrumentation (Original), the base profiling and the lightweight profiling systems with the `gcc` compiler

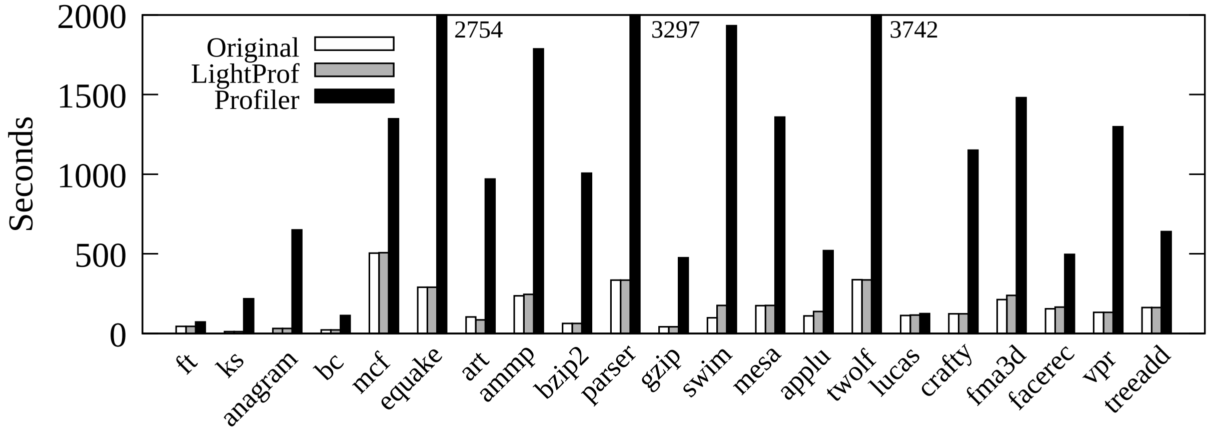


Figure 2.7: Comparison between the program without any instrumentation (Original), the base profiling and the lightweight profiling systems with the `icc` compiler

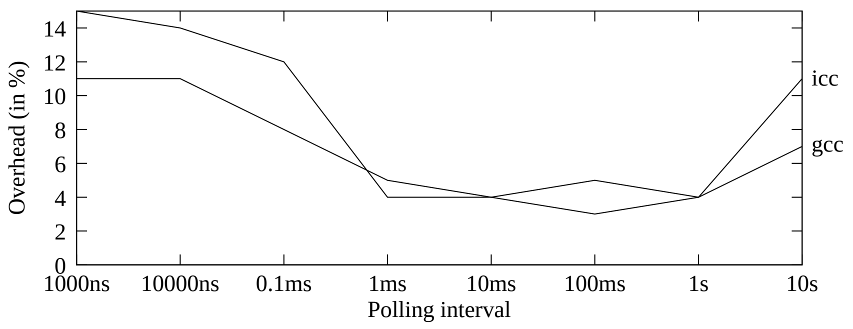


Figure 2.8: Average overhead depending on the lapse of time between two polls on the different benchmark programs

depending on its `dl` value. Hence a very constant load latency results in very few checks, whereas an erratic load latency is often polled to give a precise reading. Most dynamic profilers rely on profiling *one-entry multiple-exit* traces. These profilers are not practical when trying to instrument/de-instrument the code. Generally, the only possible solution to modify an instrumented trace is to invalidate it and ask the framework to reinstrument it. Our system handles each load independently and the profiler can halt the profiling of one load and reinstate it for another load.

Finally the frequency of the polling system is also dynamically modified. If a load is state changing, then the polling time is set to the base polling time. However, if no load changes state, then the polling period is multiplied by 2 until a maximum of 128 times the base polling time. Again, this is done to lower the system overhead.

Figure 2.8 shows the average overhead obtained depending on the choice of the base polling time for the different tested programs. As we can see, the lower the polling time, the higher the overhead, since the program will be spending more time checking the latencies of each load. However, if the polling interval is too big, the system is slower to respond to a change of phase and the overhead is also high, since the instrumentation code will remain longer in place. The polling interval of 10 milliseconds was chosen because it represents a good ratio quality of the poll/overhead of the system. Such an initial experiment can be done each time our system has to be implemented on a new hardware, in order to select a convenient polling interval.

2.2.3 Trace selection

Most dynamic optimizers use a trace selecting system instead of directly monitoring each load. For dynamic instruction optimization, it is often necessary to be able to select the frequently used blocks. Instead of handling each instruction independently, a block groups instructions together as a minimal functional unit. This allows a dynamic system to consider every instruction of a block as the same element and instrument or optimize it as a whole. In our system, this is not needed since the overhead is already low and we are interested in instrumenting each load independently. However, another important aspect to a dynamic optimizer is extra memory consumption. For example, on embedded systems, it is important to keep this consumption to a minimum. The lightweight profiling system helps in reducing the execution overhead but each called function is parsed completely and loads that are not often executed are still instrumented.

Loads that are executed only a few thousand times are not very significant and, if memory consumption is a factor, these loads should simply be ignored. Since the code must be instru-

Benchmarks	Number of loads instrumented			
	Without TS		With TS	
	gcc	icc	gcc	icc
ft	188	145	86	66
ks	313	280	120	94
anagram	384	134	265	82
bc	1841	1744	1358	1192
mcf	465	420	314	258
equake	399	1087	279	387
art	497	689	172	379
ammp	2700	2065	1104	909
bzip2	1139	1133	439	578
parser	5909	3084	4665	2415
gzip	1138	1176	535	539
swim	227	4503	95	2547
mesa	4484	4039	1579	623
applu	1319	4427	837	2760
twolf	8189	9913	3105	3162
lucas	670	3773	308	2482
crafty	6701	6247	3275	2326
fma3d	10813	10338	2045	4394
facerec	100	5496	74	2721
vpr	3834	4634	1569	1603
treeadd	81	23	64	4

Table 2.3: Comparison between the number of instrumented loads using the Trace Selector (TS) or instrumenting the whole functions with both compilers

mented to extract a profile of the current behavior of the program, a counting system can be implemented to instrument the blocks, also known as *traces*, of the functions instead of directly handling each load.

We define a trace as a *one entry-multiple exit* sequence of instructions. This definition enables us to simply insert the instrumentation code at the start of the block in order to retrieve the number of times the instruction sequence is being executed. Hence, multiple loads in a same block are monitored with only one counting system. This means that the memory consumption is reduced since we are globally reducing the size of the instrumentation code for each function. Once a block counter exceeds a fixed threshold, the loads of the block are instrumented as it is explained in section 2.2.

Table 2.3 shows a comparison between the trace selecting solution and the normal version regarding the number of instrumented loads. A direct implication is the approximate memory usage the system would need to instrument the whole code. The first thing we can notice is the reduction in the number of instrumented loads when using the block selector, which has a direct impact on the memory usage. Figure 2.9 shows the memory usage ratio between the two versions. This shows the factor by which the size of the instrumentation pool can be reduced and still be suitable to instrument efficiently the target program. There is slight increase of loads in the `icc` version of the fortran codes. Our system uses a technique called *dynamic*

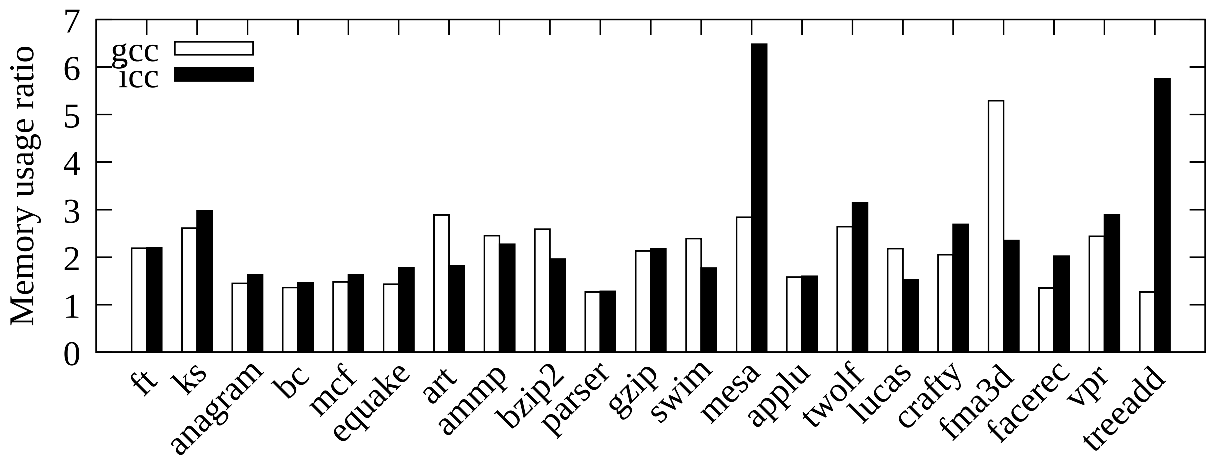


Figure 2.9: Ratio of the memory usage between using the trace selector or not

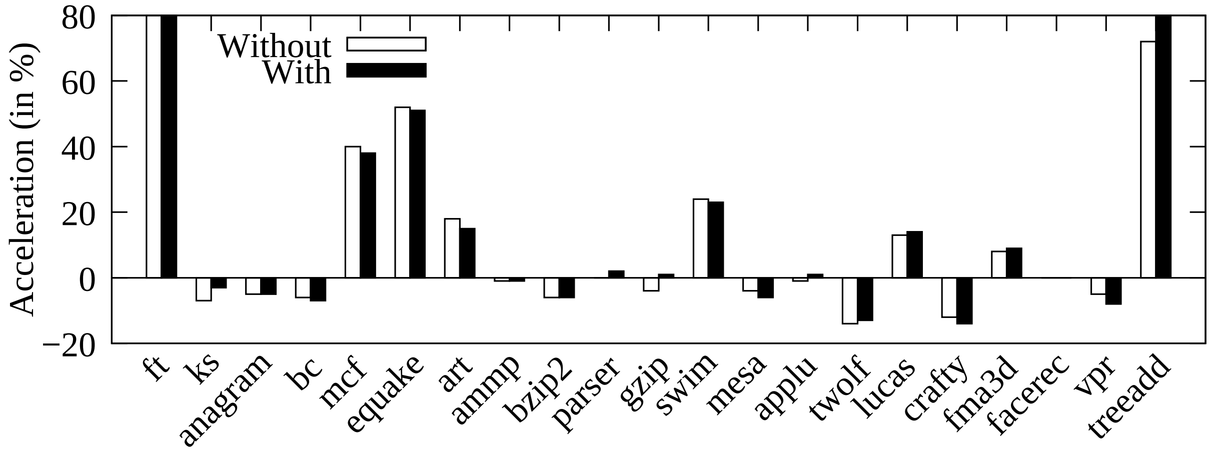


Figure 2.10: Comparison between the speedups achieved using the trace selector or instrumenting the whole functions with the gcc compiler

function instrumentation that consists in only instrumenting functions that are at least called once during the execution. This is explained in greater detail in subsection 2.4.7 and was not possible with these codes. If we compare the number of loads in the `gcc` and the `icc` versions, library functions are also included into the `icc` binary and are therefore handled by our system.

The drawback of such a system is the time spent to actually start optimizing the code. The accelerations achieved with both versions for the different benchmark programs, using the two compilers, are shown on figures 2.10 and 2.11. First, we notice that the results are similar. A slight slow-down is due to the time spent by the system to start prefetching the data for a particular load. Hence, we feel that on an architecture where memory is not a factor, the trace selection scheme is not essential in a per-load optimizing scheme. However, on an embedded system, the percentage of memory gain is an important factor and we have shown that hot-traces are selected with a 5% overhead. This means that, compared to code-interpretation solutions, our method keeps quite a low overhead convenient for instruction-based optimizations.

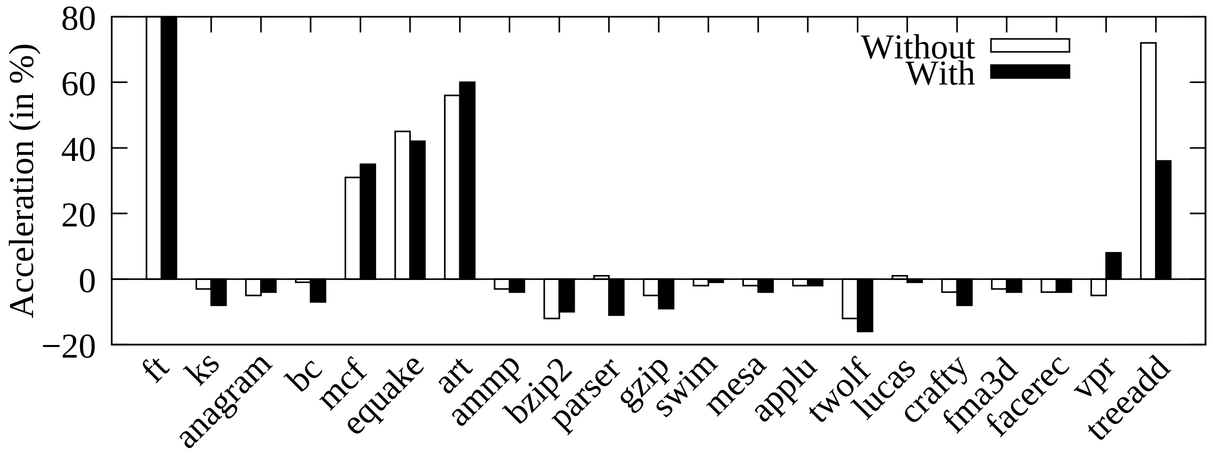


Figure 2.11: Comparison between the speedups achieved using the trace selector or instrumenting the whole functions with the icc compiler

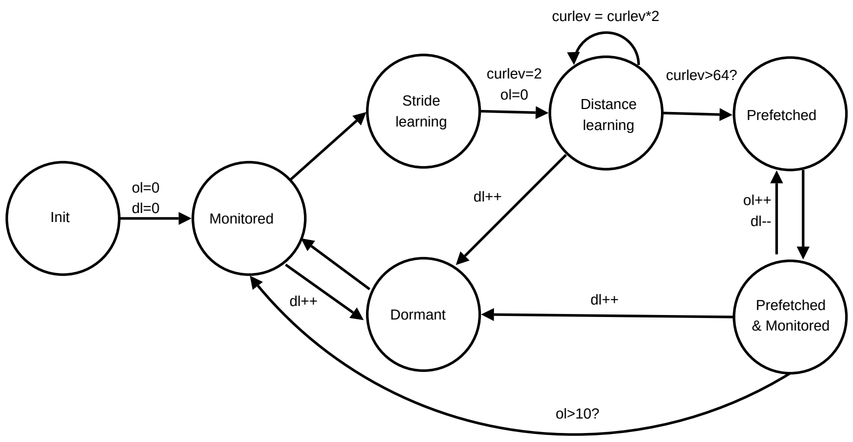


Figure 2.12: The finite state machine representing the states and transitions of the load instructions

2.3 Dynamic Optimization System

Various optimization schemes can be attached to a low overhead dynamic profiler. This section presents a lightweight solution for data prefetching. The section starts by presenting the augmented state machine and the general framework.

2.3.1 Optimizer finite state machine

The finite state machine presented in section 2.2.1 needs to be augmented to handle an optimization scheme. Figure 2.12 shows the new finite state machine implementing a data prefetching scheme. Four new states have been added and a new counter `ol` is associated to the loads.

Once the loads are instrumented, they enter the monitored state. When the number of executions reaches the threshold, depending on the average latency, they enter the dormant state or the stride learning state. The cycle between the dormant and the monitored states is the same as for the profile system presented in 2.2.

In the stride learning state, the instrumentation code segment contains stride learning code. This code retrieves the last accessed address, subtracts it to the new address and stores the stride in memory. Once the number of executions in this state reaches a threshold, the load enters the distance learning state.

Most prefetchers classify loads into different categories. For example, Lu et al. [73] distinguishes *direct*, *indirect loads* and *link list loads*, while Zhang et al. in [120] has three different categories named *stride*, *pointer* and *same object*. Our system handles every load without any distinction by simply calculating the last stride. If the stride is ineffective, the load quickly enters the dormant state where it is executed natively. On the next instrumentation, a new stride is calculated. Thus, if there exists a dominant stride in the access trace, statistically our system finds it.

In the distance learning state, the prefetch distance is tested with the calculated stride. The prefetch distance is the number of accesses before the speculated data is actually used. To calculate the best distance, prefetch code is inserted into the instrumentation code segment using different prefetch distances. The system tests every power of 2 between 2^1 and 2^6. A dynamic optimizer cannot test every possible prefetch distance since each test adds a certain overhead to the overall system. So we decided to use these powers of 2 distances since they are easy to calculate and still represent a wide range of distances.

Once the distances have been tested, the most suitable one is chosen and the load enters the prefetching state. In this state, the instrumentation code is removed and the code segment only contains the original code and the prefetch code. While the loads are in the prefetch state, the base code is accelerated by the prefetches. When loads are in this state, they generally accelerate the base program.

The load stays in that state for a duration of 2^{ol} times the base poll time. When the time has elapsed, it goes into a prefetched and monitored state where the effectiveness of the prefetch is evaluated. After a certain number of executions, the average load latency is calculated and compared with the base latency without any prefetches. As `dl` is the confidence level indicating that the load remains in the same state, `ol` is the confidence level characterizing a load remaining delinquent and being a good candidate for optimization. Such a candidate is defined if the prefetch mechanism is effective. Hence, if a prefetch is not lowering the load latency, the load goes to the dormant state and the `ol` level is reset. On the other hand, if the prefetch reduces the latency, `ol` is incremented and the load remains longer in the prefetch state.

A ping-pong scenario can occur in the case of a delinquent load which is never improved by prefetching. With no specific care, this load would always have its `dl` and `ol` levels at 1. To resolve this problem, when the latency is lowered by the prefetching, `dl` is decremented until it reaches 0 and directly incremented when entering the dormant state. In our ping-pong scenario, the load spends more and more time in the dormant state, reducing the overhead of its instrumentation. If, at another time of the program execution, the load benefits from prefetching, `dl` is slowly decremented as a confidence level indicating that this load is now becoming interesting.

Finally, the system periodically recalculates the base latency of each load. Since we do not always have the actual base latency, the system memorizes a value for it from the monitored state. This means that if a load was at first delinquent, its average was high at that moment. However, later on in the execution, it can become non-delinquent. To ensure that we eventually stop prefetching, when `ol` reaches a certain level, the load reenters the base monitoring state. If it is still delinquent it eventually reenters the prefetch state. In the case where it is no longer

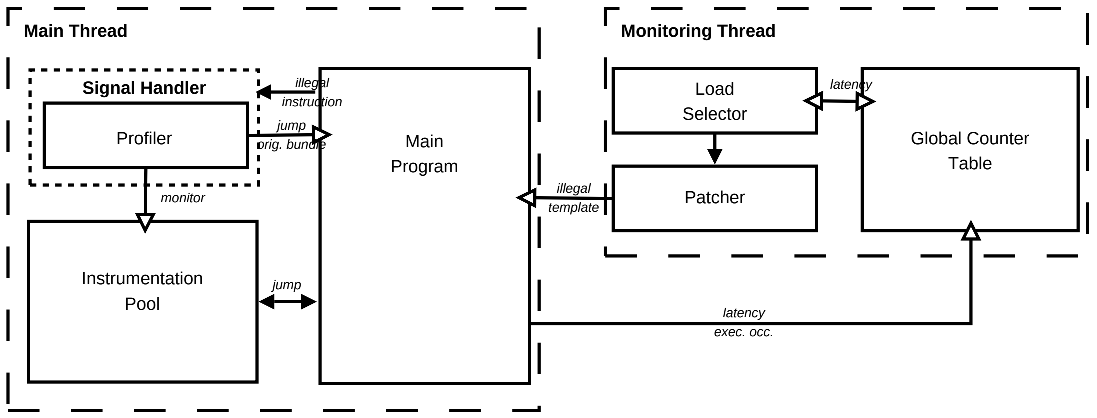


Figure 2.13: The framework

delinquent, it enters the dormant state.

2.4 Implementation Issues

The previous section presented the different modules needed to instrument, profile and optimize a program. This section presents the general framework of our system, the register allocation scheme used and details are given for dynamically modifying binary code on the Itanium processor.

2.4.1 General Framework

Figure 2.13 illustrates the profiling framework. The main program is instrumented to fill a *Global Counter Table* with load execution occurrences and latency information.

In order to modify a program binary, the read-only flag must be removed. Since the binary code tightly maps functions together, it is difficult to directly insert instrumentation code. A common solution is to insert an unconditional jump to another section of the memory, the *instrumentation pool*, where we can handle the code insertions at runtime and where an unconditional jump is used to return to the next instruction of the original code.

In the same way as in Adore, our system is implemented as a shared library on Linux for IA64 that can be automatically linked to the application at startup. A second thread, called the *Monitoring* thread is created at startup to handle the polling mechanism. The target program, whose binary is instrumented, is periodically polled in order to gather latencies and execution counters for each load. To allow a pre-compiled program to be monitored, we also modified the *libc* entry-point routine *__libc_start_main* to include our own startup codes.

2.4.2 Register allocation

Dynamic systems generally need to use a certain number of registers to achieve their goals. The solution to this problem depends on the architecture, but also on the amount of needed registers. In our current implementation, only four registers are needed to implement the instrumented code. For example, to allocate the needed registers on the IA64 architecture, a few solutions can be used as it has been presented in [31]:

- Compiler assisted: some compilers allow the programmer to request free registers. This was done in [73] where it has been shown that, if the number of requested registers is low, the impact is negligible. However, not all compilers have such an option and this is a huge drawback since it reduces the portability of the system.

- Liveliness calculation: it is possible to know which registers are being used and which are free by analyzing the program's control flow. For example in [76], Pin incrementally computes the liveliness information during execution, effectively reducing register spills. In our system, this would result in finding the four needed registers. However, it is possible that all registers are being used.

- Dynamic register allocation: this is the solution of Adore [74]. On the Itanium processor, register allocation is achieved using the *alloc* instruction. Modifying this instruction can give the system the required registers.

The third solution was chosen since it does not require the help of a compiler and has virtually no overhead compared to the liveliness calculation. However, there are limitations to this technique. First, if the last executed *alloc* has already requested every possible register, the system is unable to add its four needed registers. This was noticed for only two programs in our set of benchmarks (`ammp` and `mesa`) and concerns only 13% of the loads of these two programs. The second problem concerns functions that have more than one *alloc*. A control flow analysis would enable us to detect which *alloc* should be modified for the concerned load, but we have chosen to simply ignore these cases for the moment. This second case concerns 10% of the loads of the considered benchmarks. The last issue is when the load is in a function where there is no *alloc* instruction. This also represents 10% of the loads in the considered benchmarks and our system simply adds an unconditional jump at the beginning of the function in order to add an *alloc* instruction before the function's code.

The *alloc* instruction is modified by adding four output registers. If there is a function call, these new output registers are considered as scratch registers. This means that the output registers become input and local registers to the callee. If the caller wants to preserve them, a spill or a move to local registers is needed.

Finally, this solution is feasible in our profiler since we ensure that no branch instruction is executed during the profile of the load. To profile a load, the profiler needs to increment a value to count the number of times the load has been executed but also calculate the average latency of a load. If a branch instruction were to be executed, we would not be able to guarantee that the live-values of the dynamically allocated registers would be preserved. This is not a problem in our profiler, but a more general instrumentation tool without any control flow analysis would have to either also guarantee this or spill the registers.

Figure 2.14 shows the overhead of modifying the allocation instruction. In the general case, the benchmark programs respond very little to the register allocation modifications. In the worst case, using the `icc` compiler with program `twolf` resulted in a significant slowdown. On the other hand, benchmark programs `ft` and `mcf` actually obtain a speedup of 1% when modifying the instructions. Since most of the programs respond positively to this method, we feel it is a viable solution. However a better alternative is needed for programs like `twolf` where just modifying the allocation system results in a relatively high overhead compared to the other programs.

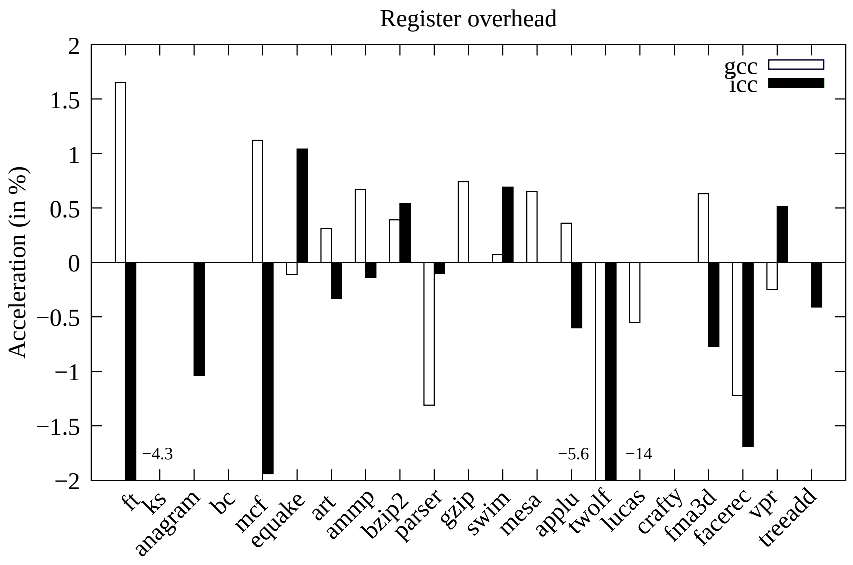


Figure 2.14: Overhead of modifying the *alloc* instruction to obtain the needed registers

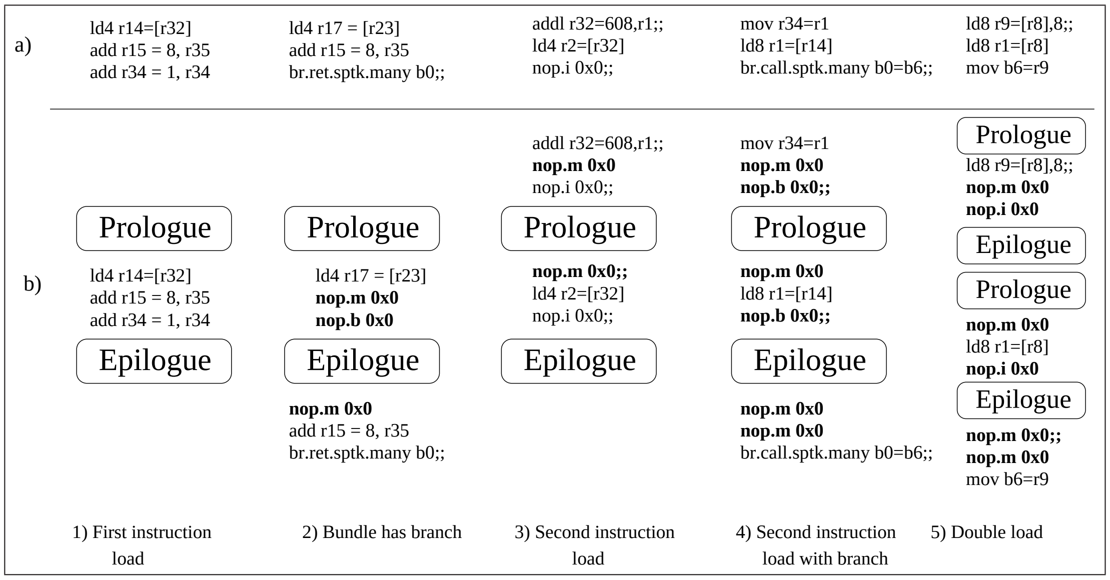
2.4.3 Code modification

The current system has been implemented for the Itanium [53] processor. On this EPIC architecture, instructions are packed into bundles. Each bundle contains three instructions that are divided into five types: Memory, Integer, Float, Branch and Extended. This complicates the insertion of unconditional jumps since, to instrument a load, the instructions joined with that load generally have to be copied into the *instrumentation pool* as well.

Each load is instrumented by defining a prologue and an epilogue section. The prologue consists in fetching the current counter and the accumulation value, incrementing the counter and checking the time counter register value. The epilogue starts with a *stop instruction*, then also checks the time counter register value. By subtracting the two time counters, a latency is calculated and, adding the result to the old accumulation information, we have an updated value of the accumulation of latencies for this load. By dividing the accumulator by the number of executions of the load, we are able to get an average of the number of cycles needed to complete this load.

The *stop instruction* is needed in order to ensure that, at the moment where the epilogue checks the time register value, the load has been completed. Since the dynamic register allocation scheme does not handle floating registers, the profiling system does not have a free floating-point register it can use as the result register of this instruction. Therefore, this instruction is a simple incrementation for an integer load and a store for a floating-point load. Though this will modify the calculation of the load latency, we see the latency as a qualitative value and not as a precise one. To compensate this fact, the threshold deciding if the load is delinquent is simply augmented. We feel that the time for one integer incrementation is statistically always the same during the program run.

For a floating-point load, we store the value of the load back into an area that has just been loaded by the profiling system. This ensures that the store will not dramatically slow-down the program. Finally, a special solution is used for *speculative loads*. Speculative loads contain addresses that, unlike normal load instructions, can be illegal. A check instruction *chk* is used


a)
b)
Prologue
Epilogue
1) First instruction load
2) Bundle has branch
3) Second instruction load
4) Second instruction load with branch
5) Double load

```
a)
1) ld4 r14=[r32]
   add r15 = 8, r35
   add r34 = 1, r34

2) ld4 r17 = [r23]
   add r15 = 8, r35
   br.ret.sptk.many b0;;

3) addl r32=608,r1;;
   ld4 r2=[r32]
   nop.i 0x0;;

4) mov r34=r1
   ld8 r1=[r14]
   br.call.sptk.many b0=b6;;

5) ld8 r9=[r8],8;;
   ld8 r1=[r8]
   mov b6=r9

b)
1) Prologue
   ld4 r14=[r32]
   add r15 = 8, r35
   add r34 = 1, r34
   Epilogue

2) Prologue
   ld4 r17 = [r23]
   nop.m 0x0
   nop.b 0x0
   Epilogue
   nop.m 0x0
   add r15 = 8, r35
   br.ret.sptk.many b0;;

3) addl r32=608,r1;;
   nop.m 0x0
   nop.i 0x0;;
   Prologue
   nop.m 0x0;;
   ld4 r2=[r32]
   nop.i 0x0;;
   Epilogue

4) mov r34=r1
   nop.m 0x0
   nop.b 0x0;;
   Prologue
   nop.m 0x0
   ld8 r1=[r14]
   nop.b 0x0;;
   Epilogue
   nop.m 0x0
   nop.m 0x0
   br.call.sptk.many b0=b6;;

5) Prologue
   ld8 r9=[r8],8;;
   nop.m 0x0
   nop.i 0x0
   Epilogue
   Prologue
   nop.m 0x0
   ld8 r1=[r8]
   nop.i 0x0
   Epilogue
   nop.m 0x0;;
   nop.m 0x0
   mov b6=r9
```

Figure 2.15: Transformations of five types of bundles

to decide whether the address was valid or not. Since the addresses used by speculative loads are not always legal and the processor can postpone their execution until a check is performed, our system adds a check to handle these cases.

Finally, the Itanium processor contains predicated instructions. Almost every instruction in the Itanium instruction set can contain a predicate. If the predicate is false, the instruction is executed but the internal state of the processor is not changed. However, since the instruction is executed even if the predicate is false, the system instruments these loads in order to try to optimize the delinquent ones.

Figure 2.15 shows the transformation made to five types of bundles containing loads.

1. The first example shows the simplest case where a load is the first instruction of the bundle. In this case, the prologue and epilogue are simply added before and after the bundle.

2. In the second example, there is a branch instruction in the bundle. To ensure that the code in the epilogue is executed directly after the prologue and the instrumented bundle, the branch is pushed after the epilogue. Of course, in the case of IP-relative branches, its destination must be carefully recalculated.

3. In the third example, the load is the second instruction of the bundle. In this case, to guarantee that the prologue code can be executed, the first instruction is moved in order to be executed before the prologue.

4. The fourth example shows a bundle where a load instruction is the second instruction and the bundle ends with a branch. This bundle must be divided into three separate bundles. The first instruction is executed before the prologue and the branch is pushed to the end of the instrumentation.

5. The last example shows a bundle containing two load instructions. In this last case, the first instruction is executed before the first prologue and then the two loads are separated

Lists	Usage
`new_instr`	monitored loads that have not yet entered another state
`used_instr`	monitored loads
`dormant`	unpatched loads that are in the original program state
`learn_stride`	loads for which the memory access stride is being learned
`pref_dist_learn`	loads for which prefetch distances are being evaluated
`optimized`	prefetched loads
`opt_instr`	monitored prefetched loads

Table 2.4: Lists associated to load states

Lists	Usage
`todo_instr`	loads that have to be patched to enter either the “stride learning” or “dormant” states
`todo_dorm`	loads that have to be patched to enter the “monitored” state
`todo_ls`	loads that have to be patched to enter the “distance learning” state
`todo_dl`	loads that have to be patched to enter the “prefetched” or “dormant” states, or reenter the “distance learning” state
`todo_opt`	prefetched loads that have to be patched to be monitored
`todo_optim`	prefetched and monitored loads that have to be patched to enter either the “dormant” or “prefetched” states

Table 2.5: Intermediate lists

by the epilogue of the first load and the prologue of the second load. Thus the latency of each load can be calculated separately.

The latency of a load can be calculated on most modern processors by accessing the internal clock register before and after the load. Though the Itanium processor is an in-order processor, using synchronization schemes would give the framework the possibility to monitor loads on out-of-order architectures. Since the number of executions that are monitored are negligible compared to the total number of loads executed, it would not be have a high impact on performance.

2.4.4 Data structure details

The optimization framework is directly built from the profiling framework presented in figure 2.13. Since many loads are going to remain in the same state during a poll, different linked lists are created. Table 2.4 shows the different lists that are present in the system.

When loads are instrumented for the first time, they are inserted into the `new_instr` list. Once they have been executed a certain number of times, depending on their delinquent nature, they are either inserted into the `dormant` list or the `learn_stride` list. The difference between the `new_instr` and the other lists is the frequency where they are traversed. Since the `new_instr` list contains every newly instrumented load, it also contains loads that are never executed. To lower the overhead, this list is only checked one poll out of ten. This raises the time it takes the system to notice that a load should change states but, in programs with many unused loads, this is compensated with the saving of the traversal overhead.

Each entry in these lists contains the address of the load bundle, the start and ending addresses of the associated code in the instrumentation pool, both counters `dl` and `ol` and the

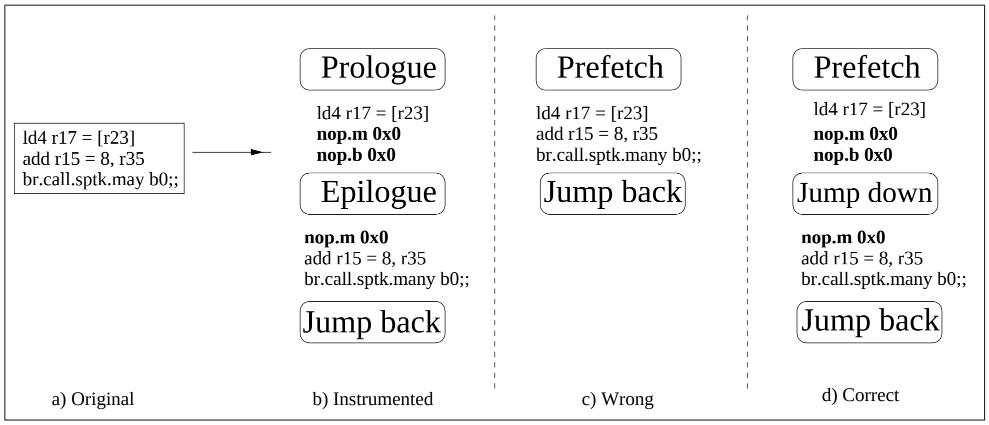


Figure 2.16: The original bundle (a), the instrumented bundle (b), an incorrect optimized version (c), the correct prefetch code by keeping the call branch at the same place (d)

index to the global counter table. So the state of the load is known according to the list in which the load structure is.

There are two groups of lists: sorted and unsorted. The new and used instrumented, the learn stride, the distance learning and the monitored prefetched lists are all unsorted since the monitoring thread has to check each entry. The dormant and the optimized lists are sorted since, when they enter those lists, `ol` and `dl` indicate how long they are supposed to stay. Sorting those lists consumes some time inserting the elements but, when the monitor checks those lists, only a few entries need to be tested and transferred to the other lists.

2.4.5 Temporary lists

The monitoring thread cannot extensively modify the binary code without a risk of interfering with the execution of the first thread. Hence, it simply modifies the template of a bundle containing a load that enters a new state. This code modification technique is explained in greater detail in subsection 2.4.9. When a load changes its state, it is first inserted into one of the temporary lists. Its bundle is replaced by an illegal instruction and the system waits for an illegal instruction to occur. When this happens, every load in the temporary lists are transferred back into the state lists. Table 2.5 explains the significance of these temporary lists.

2.4.6 Considerations about call branches

When handling bundles that contain call branches, a careful measure must be taken to prevent damage to the semantics of the program. Since we are modifying code during the execution, we must also guarantee that the bundles following a call branch are not wrongly changed. Figure 2.16 illustrates this issue by showing the instrumented code segment of a bundle containing a call branch instruction. When writing the instrumentation code, the branch must remain at the end of the instrumentation segment. If it is moved at the beginning, there is no guarantee that the call will return *before* the code is rewritten. This would change the program flow and generally results in a segmentation fault.

The solution is presented on the right side of the figure. By using a jump to the end of the instrumentation segment, we guarantee that, when the control flow returns from the call, the jump back to the original code is still valid.

Listing 2.1: Instrumentation code

```
movl r38 = @counter;    //Load the address of the associated counter
ld8 r35 = [r38], 8;     //Load the counter value and increment the address
                        //Now, the address is now at the accumulator
ld8 r36 = [r38], -8;    //Load accumulator and decrement the address
add r35 = r0, 1, r35;   //Increment the counter
st8 [r38] = r35, 8;     //Store new counter and increment address
                        //Now, the address is now at the accumulator
mov r37 = ar.itc;       //Get clock counter

ld8 r17 = [r25];        //The load

st8 [r38] = r17;        //Store the load to halt the processor
mov r35 = ar.itc;       //Get clock counter
sub r37 = r36, r37;     //Subtract before time from accumulator value
add r37 = r35, r37;     //Add after time to accumulator
st8 [r38] = r37;        //Store new accumulator value
```

2.4.7 Dynamic function instrumentation

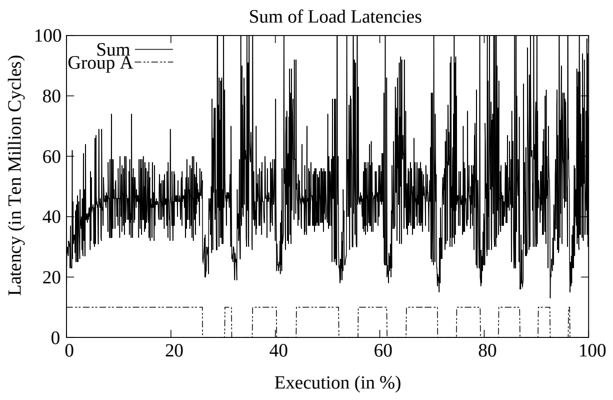
The Itanium architecture is based on explicit instruction level parallelism. This means that the compiler decides what instructions can be executed in parallel. The instructions are grouped into bundles. Each bundle contains three instructions and a template, which gives the type of each instruction. The template is 5 bits long and each instruction uses 41 bits. The total size of the bundle is 16 bytes.

At the start of the execution, the program calls an initialization function of our system. This function first fills an array with each function name, start address, size and the first bundle by reading the ELF header. To lower the overhead of the system, the binary code of each function is not parsed at this point but only at their first call. This is achieved by replacing the first bundle of each function by an illegal instruction. Additionally, the index to the function array is added to the bundle. This directly gives the system the index to the array containing the information regarding the functions.

When the illegal signal handler is called, the template of the bundle is checked. If it is the value used to define a first call to a function, the index is retrieved and the handler copies the first bundle back into place and calls the parser function. As explained in subsection 2.4.2, the parser first handles the register allocation issue and, if it is not possible to obtain the required registers, the original function is left untouched and the execution resumes. In the case of Fortran codes using the `icc` compiler cannot use this system since the associated signal handler is not able to handle the current version of the code correctly. However, if registers have been allocated, the function is parsed and the loads are copied into the *instrumentation pool* with the profiling code.

Another solution to this problem would be to replace the first bundle of a function by an unconditional jump to a handling code instead of using the signal handler. However, this would necessitate to write a trampoline between the original code and the parsing code to preserve the semantics of the program. This is more complex and since the signal handler is only called once, the overhead incurred is minimal.

Listing 2.1 shows an example of the prologue and epilogue assembly codes. In the middle of the code is the load that is instrumented. The instrumentation starts with setting the address of the associated counters. Each load from the original is associated to an *execution counter* and an *accumulator counter*. The execution counter is the number of times a load is executed



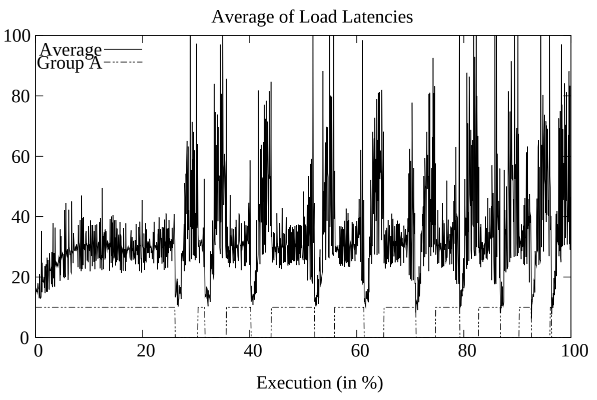


Figure 2.17: Left: Sum of load latencies (in ten millions cycles) with execution of functions from group B. Right: Average load latencies with execution of functions from group B

Group A	**Group B**
refresh_potential	compute_red_cost
update_tree	replace_weaker_arc
primal_iminus	price_out_impl
bea_compute_red_cost	
bea_is_dual_infeasible	
sort_basket	
primal_bea_mpp	

Table 2.6: Two groups of functions that share the program's execution

and the accumulator represents the sum of the different latencies of the load. The prologue starts with loading both counters using a post increment/decrement system from the Itanium processor. This technique allows, in one instruction, to perform the load from an address and then update the address. Once both counters are loaded, the execution counter is incremented and stored. Then, before the load is executed, an access to the clock register is performed.

Once the load is executed, a stop instruction is used to stall the processor until the data has arrived. In this case, a store instruction is used. Then, the second clock register access is performed. By subtracting the time obtained before the load and then adding the time after, we obtain the number of cycles between both values. Finally, the accumulator is updated after adding this value.

2.4.8 Polling mechanism and evaluation

Once the system has modified the first bundle of every function and set the signal handler, it creates a second thread to monitor the load activity. This second thread polls the different counter values that have been modified by each profiling code. At every wakeup, the profiler checks each load counter. If the execution counter of a load is over a fixed threshold, it stores the counter value and the accumulated load latency in the associated load-information structure.

The presented profile method enables to evaluate program bottlenecks regarding load latencies. For example, program `mcf` shows a cycle of two groups of functions given in table 2.6. Figure 2.17 shows the sum of latencies of every load in the program throughout the execution.

The dotted line represents the execution of functions from group A. When the line has a value of 10, it means that a function from the group A is being executed. We can clearly see that, during the execution of the group B, the sum of latencies is much higher.

Also, when comparing the average load latency of program `mcf` in figure 2.17 with the execution of the functions of group B, we see that the average is higher during the execution of the functions from group B. This means that an optimization system should clearly handle the three functions of group B in order to achieve a speed-up.

2.4.9 Dynamic code modification

While the profiler is instrumenting and de-instrumenting the code, it is possible for a modification to be made to a bundle that the main program is currently executing. Any dynamic code modification must be done in a way that preserves the semantics of the program. On the Itanium processor, the length of an atomic write is 8 bytes while the length of a bundle is 16 bytes. Thus the modification of a bundle cannot be considered atomic.

Two solutions are generally used. The first puts the main thread to sleep and modifies the code at that moment, the second writes an illegal instruction to stop the execution of the main thread and, when an illegal signal is received, the signal handler performs the modifications. Our system modifies the template to an illegal format before modifying the rest of the bundle. On the Itanium processor, each bundle has a template to indicate the contained instructions. This template is 5-bits long and there are 8 reserved values that cause an Illegal Operation Fault. Finally, the bundle's address is put into a special linked-list that represents pending modifications. To differentiate a modified bundle and the modification of the first bundle of a function rapidly, a different illegal template value is used.

When the main program encounters such a bundle, an Illegal Operation signal is captured and the modification of the pending bundles can proceed. Since modifying a bundle template can be done while the main program is executing the bundle, a safety mechanism is implemented. When the handler is called, two parameters are given to the handler: the address of the illegal bundle and the instruction that has raised the illegal instruction.

In the best case scenario, it is the first instruction that raises this illegal operation. In this case, the profiling system simply goes through the linked-list to complete the modifications and the program resumes its execution at that corrected bundle. In the worst case scenario, it is the second or last instruction that has caused the illegal operation. This means that when the program resumes its execution, it must not re-execute the first instructions.

To correct this problem, a safety net is implemented by replacing the bundle by an unconditional jump to a correcting patch code. Figure 2.18 shows this correcting patch. On the left, the original bundle executes only the first instruction. It is first replaced by an unconditional jump to a three bundle segment. The first bundle consists in the rest of the original bundle that needs to be executed. The second bundle has an illegal template that gives the handler the chance to finish the modifications of all pending bundles. Since the instruction stream is stopped at this new illegal bundle, when the handler function now returns, the instruction stream resumes at that bundle. To let the program flow continue, the system fills this bundle with nop instructions as we can see on the right of the figure, in order for the program to reach the last bundle. The last bundle of the patch code is a branch that returns to the bundle following the one that generated the first Illegal Operation.

However it is possible that at least one of the last instructions of the bundle contains a branch instruction. In this case, when the program executes them, it jumps back to the original code. The program then continues its execution until the next illegal operation. This case is

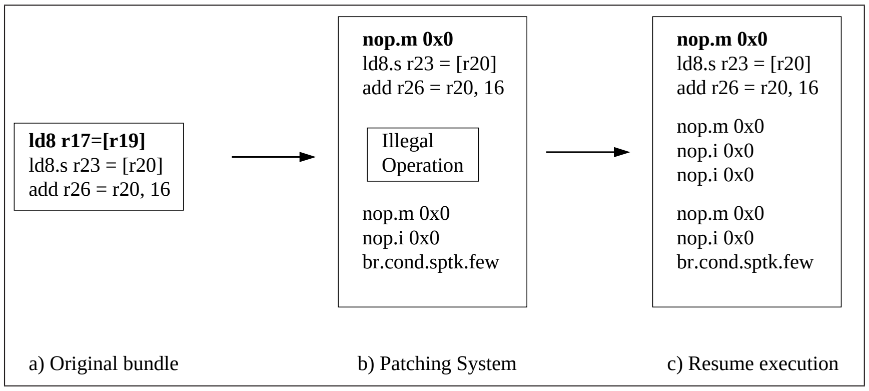


Figure 2.18: Original bundle that has executed only the first instruction (a), patching mechanism (b), once the modifications are finished the illegal operation is removed (c)

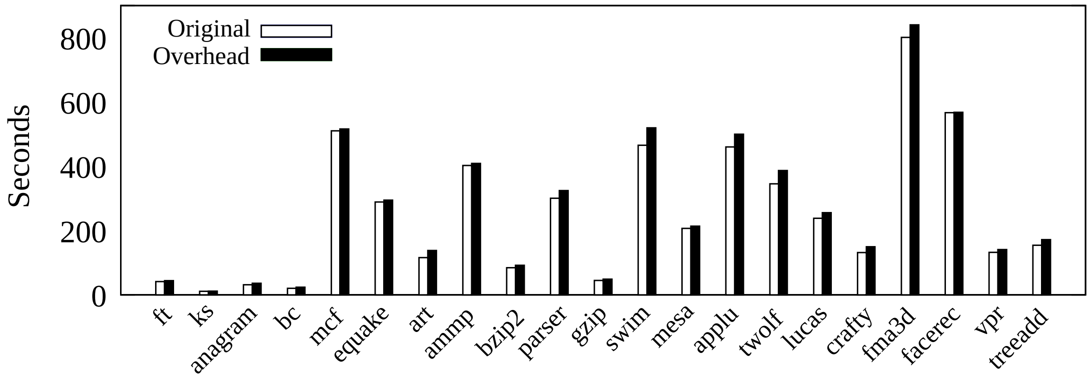


Figure 2.19: System overhead with the `gcc` compiler

extremely rare and is not an issue, since the next illegal operation allows the profiler to modify the pending bundles.

In the general case there is no branch in the bundle and the program raises the Illegal Operation in the patching segment resulting in the modification of the pending bundles. Then the program resumes its course through the branch instruction at the end of the patching segment.

2.5 Experimental and Performance Evaluation

In the current implementation, the minimum number of times a load has to be executed before changing its state is fixed to one thousand. The average latency over which a load becomes a candidate for prefetching is fixed to 15 processor cycles.

Our system was run with nine SPECFP2000 benchmarks, seven SPECINT2000 benchmarks with reference inputs, four Pointer Intensive benchmarks (`ft`, `bc`, `anagram` and `bc`) and one Olden benchmark (`treeadd`). Our test machine is a 2-CPU 1.3Ghz Itanium-2 rx2600 workstation. However, notice that our system is entirely run onto one unique processor. The operating system is Linux kernel version 2.6.9-42 with glibc 2.3.4. We used both compilers `gcc` version 4.1

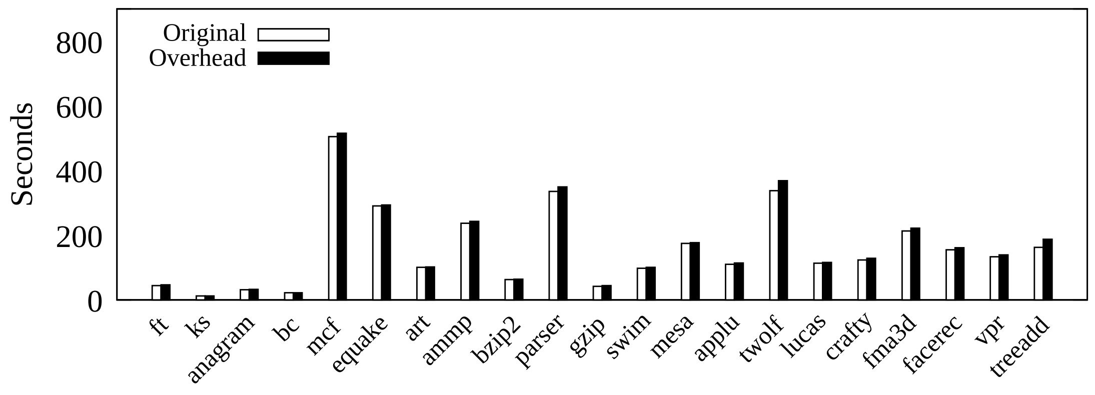


Figure 2.20: System overhead with the `icc` compiler

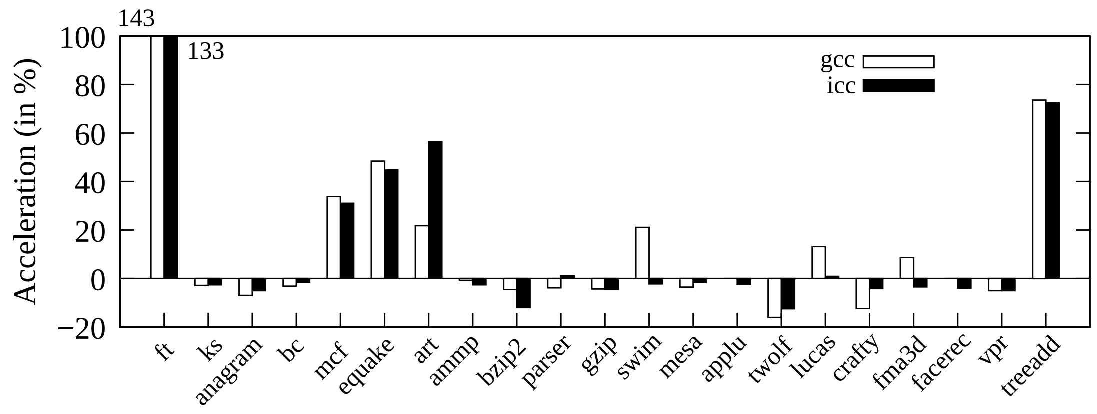


Figure 2.21: Performance Speed-Up of Dynamic Prefetching with `gcc` and `icc` compilers

and `intel icc` version 8.1 with `O3` to compile the benchmark programs. Notice that the `icc` compiler generates prefetch instructions with `O3` while the `gcc` compiler does not. Finally, every result is the average of 5 distinct runs.

2.5.1 System Overhead

To measure the system's overhead, we inhibited the insertion of prefetch instructions. In such a dynamic system, it is impossible to really calculate the overhead of the system since its decisions obviously differ when not inserting the prefetch instructions. But this gives an idea of the cost of such a system as a worst case scenario. As we can see on figures 2.19 and 2.20, though the execution times differ between the `gcc` compiler and the `icc` compiler, the overhead is on average 5%.

2.5.2 Performance Analysis

Figure 2.21 shows the performance of our system on the considered benchmark programs compiled with both `gcc` and `icc`. About half of the programs show speed-ups ranging from 2% to 143%. For programs showing no benefit, the performance difference is between -1% and -16%.

The worst case is program `twolf`, which already had a bad performance because of the register allocation scheme. After that initial overhead, the framework is unable to find a load to optimize efficiently. If we only consider programs that are being slowed down, the average slowdown is only slightly over 5%. Long duration programs respond better to the system showing significant speed-ups. In the case where there is no single interesting load, each load enters the dormant state for longer periods of times making the whole system slowly shutdown. For example, program `bc` lasts only 22 seconds whereas `parser` lasts more than 300 seconds. When comparing the overhead of both programs, we notice that `parser` is almost not slowed-down by the `gcc` code and even accelerated using `icc`. We think that `bc` does not have an execution time long enough for the system to counterbalance its overhead. Finally, the additional execution time is very low for the programs that are slowed down.

We notice very different results when comparing the execution of codes from the two compilers. For example, `art` responds much better with the `icc` version of the code. However, `equake` obtains a speed-up of almost 50% with `gcc`. The `icc` version also has a considerable speedup. This shows that, even if `icc` issues prefetches into the instruction stream, our system is still able to improve performance. This is especially true since if there is a load that is being correctly prefetched by the program, it will not be considered delinquent by our system. Therefore, the system can be considered as a prefetch helper and not a competitor.

For `mesa` and `ammp` the results can be explained by the fact that our system does not instrument the functions that use all the processor registers. This means that the biggest functions could not be transformed as it is explained in section 2.4.2.

Finally, for program `swim`, a big difference in execution time was noticed between the codes compiled with `gfortran -O3` and `ifort -O3`. With `gfortran`, the execution lasted about 11 minutes, compared to the 90 seconds of the `ifort` version. A significant speed-up was obtained using the `gfortran` compiled code as expected for such an execution time.

2.5.3 Prefetch distance

Since our system dynamically calculates the best prefetch distance for each load, it is interesting to compare this system with a fixed distance technique. Figures 2.22 and 2.23 show the results of this comparison for both compilers. The comparison with the distance learning solution is done using the best distance for each benchmark compared to our distance learning system. Since our system only uses power of 2 distances, our learning mechanism does not test every possible distance.

The numbers over the histogram groups are the best fixed prefetch distance found between 1 and 64. It can be noted that programs are quite sensitive to the used distance. For programs `art` and `treeadd`, a fixed distance can even slow-down the program instead of accelerating it. There is a cost of learning the prefetch distance and this explains the slight loss of performance when learning the distance dynamically instead of testing each distance off-line.

As we can see, each benchmark has particular distance for which the performance is optimal. In most cases, like `equake` for example, where the best prefetch distance is 35, the system could also include more distances in the learning phase. This would obviously slightly heighten the overhead but would give the system a better chance for such programs. However, with the current system, we feel the performance of the system is already significant with the actual learning mechanism.

Though our performance never dramatically outperforms the best fixed distance, in programs where the input modifies the behavior of the program or in cases where different programs need to be optimized by the same optimizer, our system performs and adapts correctly. Finally, the

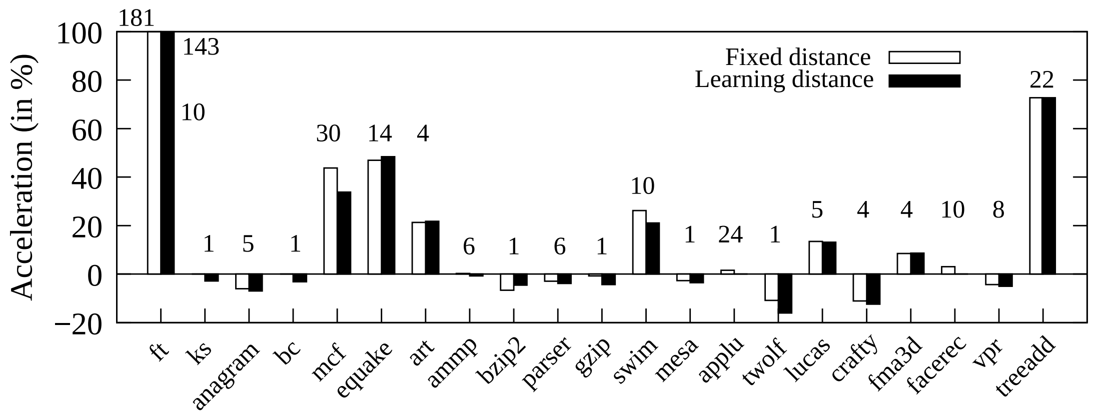


Figure 2.22: Effectiveness of the prefetch distance learning process with the `gcc` compiler

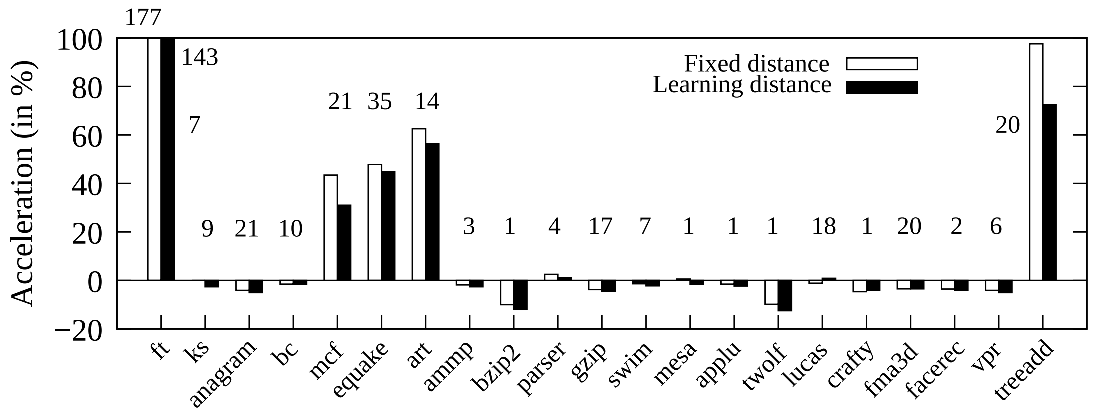


Figure 2.23: Effectiveness of the prefetch distance learning process with the `icc` compiler

Benchmarks	Distance changes		Loads changing distances		Optimized loads	
	gcc	icc	gcc	icc	gcc	icc
ft	3	3.33	2	2	0.67	2
bc	0	0	0	0	1	1
mcf	21.33	14.67	9	6.67	16.67	10
equake	27.33	18.67	10.33	2.33	16	4.33
art	20.67	10.33	9	4.33	9.67	4
ammp	20.67	11	8.33	4.33	21.33	15
bzip2	6.67	2.33	4.67	2.33	0.67	5.33
parser	128.67	68.67	64.67	32.33	155	80.33
gzip	0	0	0	0	0	1
swim	97.67	5	24	3	32.67	8.67
mesa	0	0	0	0	0	1.33
applu	79	5	33.33	3.67	71.33	8
twolf	27	8.67	10.33	4.67	34.67	32.33
lucas	36	0.67	22	0.67	68	0
crafty	4	3	2.33	2.33	40	30
fma3d	143.67	1.33	48.33	1	81.67	2.33
facerec	47.67	0	3	0	0.33	0.33
vpr	40.67	39.67	15.67	14	34	23
treeadd	7.33	8.33	3.67	0.67	5	0.67

Table 2.7: Prefetch distance variations

differences between fixed distance and distance learning techniques are important for programs `ft` and `treeadd` because their execution times are very low. For example, there is actually only a 2 second difference for program `ft`.

Table 2.7 shows the average number of changes in the prefetch distance during the executions of the different programs. Only programs having at least one load that changed its prefetch distance are shown. A change in the prefetch distance is defined when, after an initial distance is calculated, a better distance is found. For example, `treeadd` only has one load that is optimized for the `icc` version by the system but the best prefetch distance changes on average 8 times during the execution of the program. The prefetch distance can change on two occasions. First, when the load goes from the dormant state to the instrumented state. If it is still delinquent, a stride and prefetch distance calculation occurs. Second, if the load is continuously optimized, the load is reinstrumented to recalculate the current latency. When that happens, if the load is still delinquent, another stride and prefetch distance calculation occurs. The percentage of loads that have a prefetch distance change out of the loads that have been optimized is 42.6% across the different programs, and the average number of changes per load is 2.54 per program for every load that changes distance at least once.

Figure 2.24 shows the effect of modifying the fixed distance. The figure shows three programs `lucas`, `art` and `treeadd`. The programs were executed with prefetch distances ranging from 1 to 64. Only three programs are shown but the other programs show similar variations. Programs `equake`, `ft` and `swim` showed the same kind of sensitivity than `art` or `treeadd` when modifying the prefetch distance. On the other hand, the programs `mcf` and `fma3d` show little or no sign of change, like the program `lucas`, when we modify the prefetch distance. The remaining programs are always slowed down by our system.

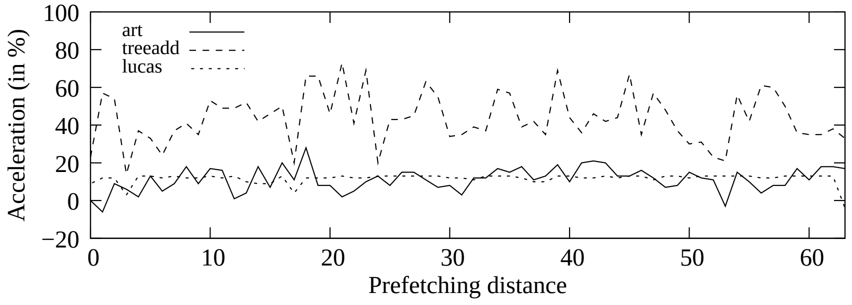


Figure 2.24: Acceleration depending on the fixed distance used ranging from 1 to 64 using the gcc compiler

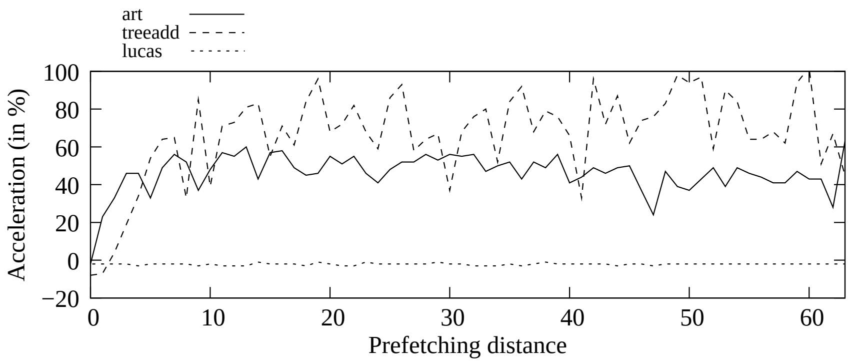


Figure 2.25: Acceleration depending on the fixed distance used ranging from 1 to 64 using the icc compiler

Figure 2.25 shows the effect of modifying the fixed distance using the `icc` compiler. With this compiler, only two groups of programs were found. Accelerated programs `mcf`, `equake` and `ft` resemble the behavior of `treeadd` and `art`. The other programs, which show a general slowdown resemble the behavior of `lucas`.

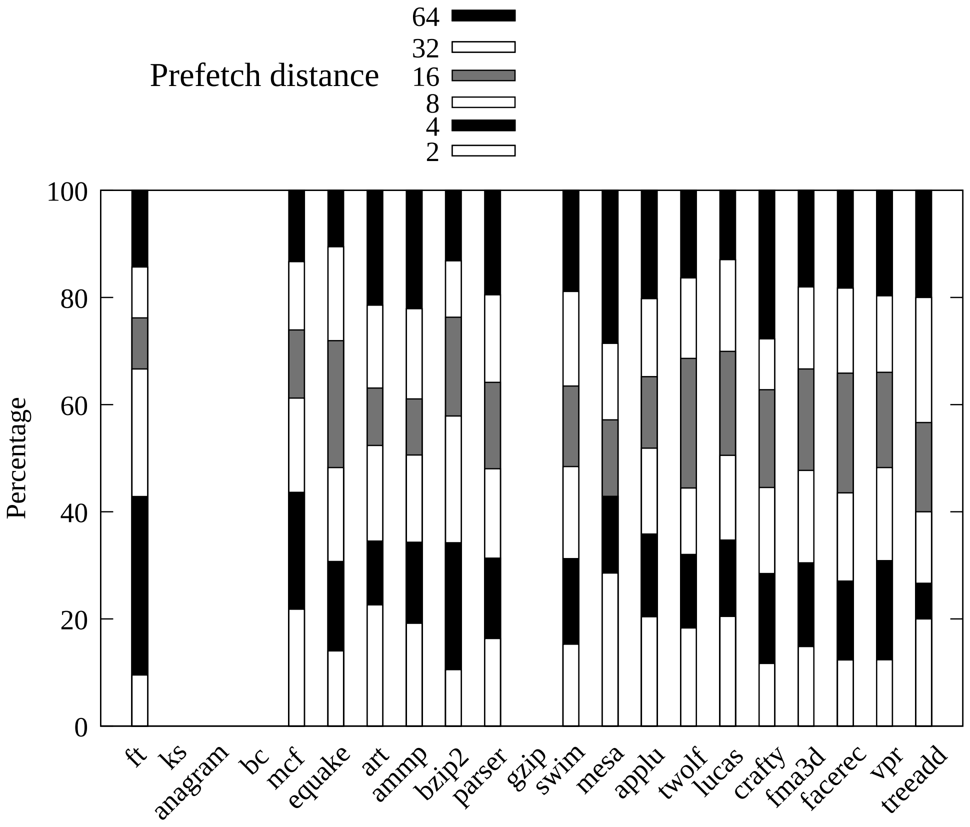


Figure 2.26: The distribution of prefetch distances used

The used prefetch distances by the system are equally balanced between the different distances used. Figure 2.26 shows the percentage of each prefetch distance used using the `gcc` compiler. The empty histograms are from programs where no efficient prefetch distance was found. As we can see, there is no benchmark that uses exclusively one distance. Each benchmark program used, at one time or another, almost every prefetch distance. This shows that a dynamic prefetching system must be able to use different distances in order to achieve the best possible speed-up.

Figure 2.27 shows the percentage of loads for which the system always assigns the same prefetch distance. For benchmark programs that have at least one load that was optimized, the average percentage of loads that always have the same prefetch distance is 56%. However, `mesa` contains only one optimized load and the system always maintains its prefetch distance. The loads from program `facerec` always change their prefetch distance.

Finally, figure 2.28 shows the distribution of loads that changed prefetch distance at least twice. Though most benchmark programs have a majority of loads in this case that change twice, `ammp`, `treeadd`, `facerec` and `swim` have more loads with a higher number of distance changes. Program `facerec` had the maximum number of prefetch distance changes with 35 changes.

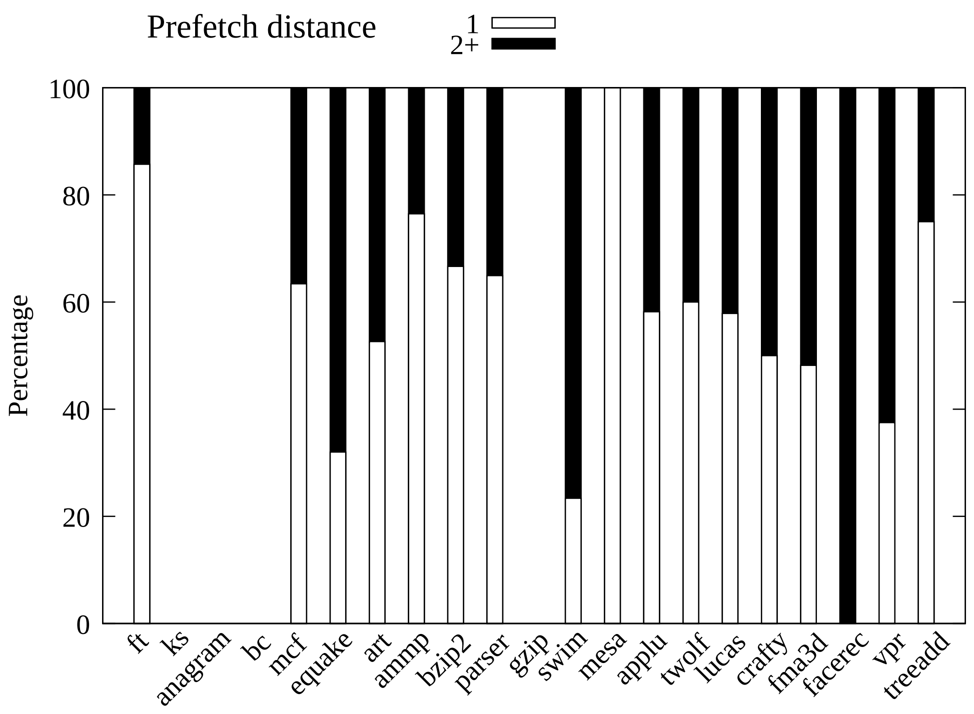


Figure 2.27: Comparison between loads maintaining the same prefetch distance and the loads using more than one distance

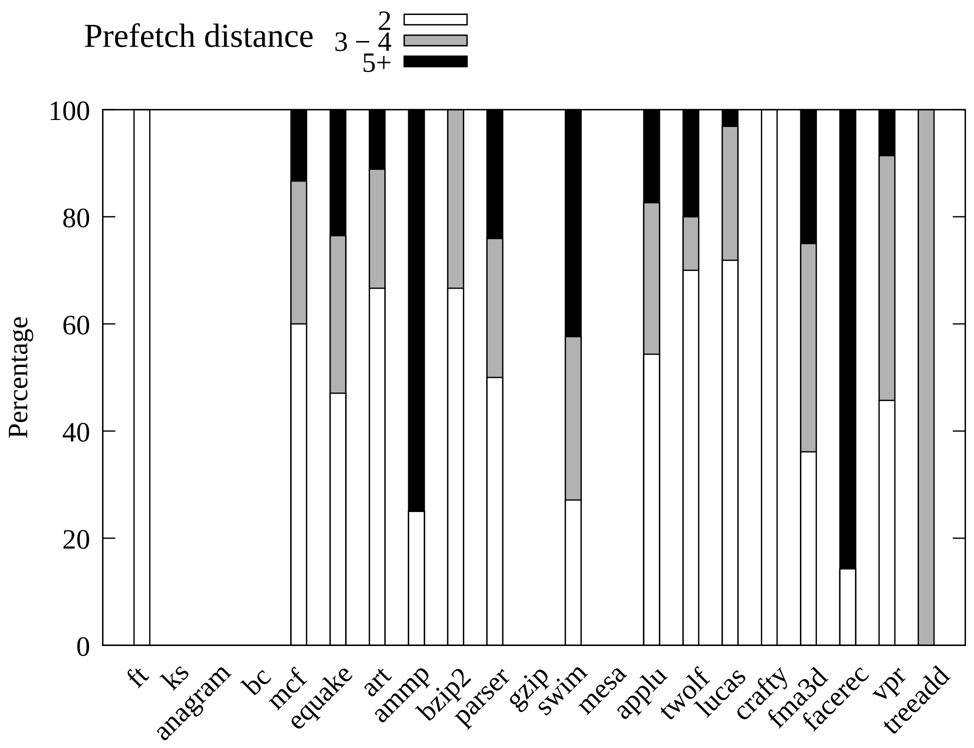


Figure 2.28: Comparison between loads which changed twice their prefetch distance, 3 or 4 times and more than 5 times

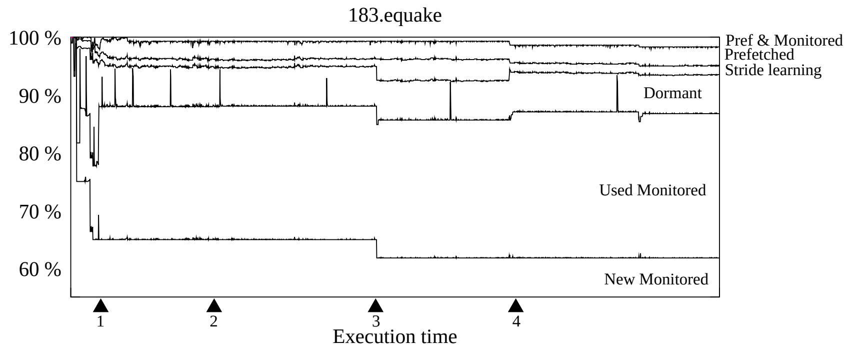


Figure 2.29: Evolution of the number of loads in each state for program `equake`

2.5.4 Behavior Analysis

Since every load is associated to a particular state, it can be interesting to analyze the evolution of the proportion of loads in each state. Figure 2.29 presents this evolution for program `equake`. The *monitored* state is divided into two groups: the *new monitored* and the *used monitored* loads, where the *new monitored* loads are those that have never been executed more that 1000 times. At a given time, the number of loads for a certain state is given by the interval between two successive curves. For example, if we look at the end of the execution, we can see that there are 65% of the loads remaining in the "new monitored" state, compared to $87 - 65 = 22\%$ in the *used monitored* state.

The four arrows under the time axis represent some interesting phase transitions. The first arrow shows the end of the initialization code and the start of the computation code. After the initialization phase, a lot of loads simply remain in the *used monitored* state since they are not being used anymore. As we can see, until the third arrow, the proportions of each group do not change. The second arrow shows a different peak that is generated by the monitoring phase. These peaks represent loads that are transfered from the dormant state to the monitored state. Since most of the loads are instrumented at the same time and are all sent to the dormant state for the first time, they all wake up together. Since they remain non-delinquent, they simply become dormant again but with a higher `dl` level. It is this incrementation that generates the increasing interval between two successive peaks.

The third arrow represents a real phase transition in the program's behavior. At that moment, the number of *new monitored* loads decreases. This means that the system has found new loads that have been recently executed enough times to change states for the first time. Since the proportion of *used monitored* and *dormant* loads remain constant, we can determine that the loads all went into the *stride learning* interval. This means that the program has entered a new portion of code and that the loads are considered delinquent. Finally, the fourth arrow shows the time where the loads have been filtered out into two groups: some return to the dormant state and some continue to the prefetch state.

2.6 Conclusion

In this chapter, we showed how to implement a dynamic analysis and optimization system, which is entirely software, that inserts prefetch instructions where it has been evaluated as effective by measuring the load latency. The system does not use any interpretation or emulation like other solutions, but dynamically modifies the target program to directly take into account the needed instrumentation and maintains a low overhead. The software implementation has been achieved by constantly optimizing the code in order to minimize the induced overhead. It has been shown on several benchmark programs that even if it is entirely software, significant speed-up is obtained. Moreover, our performance results look similar to results obtained in other works entirely or partially based on specific hardware.

However we think that the overhead could still be lowered by avoiding jumps to the instrumentation pool and by inserting monitoring and prefetching code in the place of existing `nop` instructions surrounding loads.

Finally, though the achieved speed-ups are promising, the prefetching scheme can be considered too as simple. Assuming each load might be stride-based, testing each distance in order to find a correct prefetch distance is time-consuming and not always applicable. A solution is to create a model that can help decide whether to optimize or not a particular load. The solution we propose, presented in the next chapter, is to use a Markovian model.

Chapter 3

Esodyp: an Entirely SOftware and DYnamic Prefetching system

The previous chapter presented a lightweight dynamic system to optimize programs by automatically detecting delinquent loads. However, the system does not try to understand the memory access behavior before trying to optimize the loads. This chapter presents a Markovian model that is created at run-time and can perform predictions at a low cost. A Markovian model uses past elements in order to calculate predictions. The chapter presents the theory behind Markovian models and compares two possible structural representations. Finally, an API is provided and tested in order to optimize and accelerate target programs.

3.1 Introduction

Handling data cache misses correctly can help to achieve significant speed-ups if the prefetcher is able to predict correctly the occuring memory stream. In chapter 2, we presented a simple stride predictor that handles one type of memory stream. However, there are certain limitations to this system. For example, before knowing if the data stream is strongly strided, it must test different prefetch distances. If the optimization framework had a mechanism to modelize the data stream, this would not be necessary. This chapter presents a solution that modelizes the data stream using a Markovian model.

Predictors usually consider information from previous events to calculate the probabilities of future events. Certain predictors, for example stride predictors, are relatively simple. However, more complex models are possible. Markovian models are generally considered as correlated predictors and are generally represented by finite state machines. Using statistical information about previous monitored events, they calculate the probability of future events.

Many works have used Markovian models to understand or predict the behaviors of programs [50, 62, 40, 34, 65, 41]. When considering data prefetchers, Markovian models can be integrated into the processor chip to handle branch predictions [21] or to prefetch data [55]. In the latter technique, Joseph and Grunwald use a hash table to handle multiple data streams and the solution is able to profile and prefetch data. However, the system only uses the last element in the sequence to calculate predictions. In our system, the model handles a variable number of elements to calculate the predictions. Finally, Markovian models can be complex to create and off-line techniques can be used [62]. Once the model is created, it can be loaded into the chip and used at the appropriate time.

Software solutions also exist. For example, Bartels et al. [7] present a solution predicting page-faults that uses multiple past elements to calculate a prediction. In this solution, a graph is also used to implement the model. However, a hash table query is needed at each call in order to find the correct prediction. Furthermore, the system only predicts the next page fault. In our system, the graph representation does not need a hash key calculation since the information is kept in the main structure and only the current element is needed to update the model. The predictions of our model also depend of a *prefetch distance*, meaning that our model can predict elements further in advance.

Tree representations are another common solution to represent Markovian models. Suffix trees are for example often used for pattern recognition algorithms [94]. Begleiter et al. [8] compare six prediction algorithms using tree representations of Markovian models. These systems represent the sequence of elements correctly but necessitate a complex calculation in order to calculate a prediction.

This chapter presents a dynamic solution to create a Markovian system prefetcher. The dynamic system must be able to gather the information during the execution of the program, create a model to prepare the predictions and issue prefetches. The model is created in the form of a graph in order to limit the memory and CPU overhead. Also, the system considers memory strides instead of considering the absolute addresses. A memory stride is the difference between two consecutive memory accesses. In many cases, this difference enables the system to handle large memory regions with a relatively small model. Furthermore, compared to hardware solutions [39, 55], the solution we present is totally software. This means that it is portable across different architectures and can be easily extended to other applications.

Though the implementation was intended for predicting and inserting prefetch instruction to reduce load latencies, chapter 4 extends this work to the *Distributed Shared Memory* systems. The current chapter begins with a presentation of the Markovian model in section 3.2 and presents briefly some theoretical aspects. The implementation aspects and decisions are discussed in section 3.3, where two potential data structures are studied and compared. Finally, section 3.4 shows two examples of how the model is used and section 3.5 shows the performance and evaluation of the model on different benchmark programs.

The work presented in this chapter was published in [9].

3.2 The Markovian model

At compile time, modern compilers use different optimization passes in order to create an optimized binary code specific to a certain processor. However, it is often difficult to assess what optimization should be used in certain cases. Dynamically allocated structures or data dependent behavior renders the application of some static optimizations impossible. However, a dynamic solution can decide at run-time if the optimizations are beneficial by directly testing their impact on the target program. Though generally more costly in terms of CPU overhead and more complex to implement than a static solution, it is a good compromise when a decision cannot be made at compile time.

Statistical information about the behavior of the program is needed for the dynamic optimizer. However, this information must generally be filtered in order to extract the important elements. One statistical method is the construction of a Markov model. A Markov model stud-

Element	Successor
A	B
B	C,D
C	B
D	B

Figure 3.1: A simple prediction table

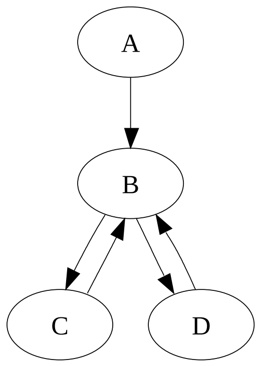


Figure 3.2: A simple graph

ies a sequence of information and extracts certain statistical information. The information can generally be described as a notion of sequencing. It can be the fact that instruction A precedes instruction B, as it can be that the memory block C is accessed after D.

3.2.1 An example

To understand a Markovian model, let $ABCBDBCBD$ be the considered sequence of elements. A description of the sequence could be:

- Though there is only one element A, after each element A is an element B;
- After each C or D is an element B;
- Finally, after a B is either a C or a D with equiprobability.

One way to modelize this would be to define each distinct element of the sequence and list the elements that appeared directly after it. Figure 3.1 shows one possibility of illustrating these predictions. The first column represents the current element and the second what the model noted as possible successors. As we can see, for the element B, there are two entries in the second column. This means that, if B is the latest element processed by a model that only considers the current element, it cannot predict accurately the next element.

Another representation is the figure 3.2 in which the nodes represent each element of the sequence and each directed-edge the direct order in which the elements appeared in the sequence. Since the graph can be considered as a state machine, we use the term *node* or *state* throughout this chapter.

The two presented models can be used to predict the future elements of the sequence. For example, let us assume we are using the table representation and the current element is C. The first column contains a C and the corresponding second column contains a B. The model would then predict that the next element will be a B. In a classical implementation, a hash table would be used with the key representing the current element and the associated hash value being the prediction.

3.2.2 Markov chains

Markov models exploit the theory behind Markov chains. A Markov chain is a sequence of random variables $x_1, x_2, x_3, ..., x_n$ with the property defined in the figure 3.3.

$$\boxed{Pr(X_{n+1} = x|X_n = x_n, ..., X_1 = x_1) = Pr(X_{n+1} = x|X_n = x_n)}$$

Figure 3.3: Recursive definition of the probability defined by the Markov Model

$$\boxed{\begin{array}{c} Pr(X_{n+1} = x|X_n = x_n, ..., X_1 = x_1) = \\ Pr(X_{n+1} = x|X_n = x_n, X_{n-1} = x_{n-1}, ..., X_{n-m+1} = x_{n-m+1}) \end{array}}$$

Figure 3.4: Recursive definition of the probability defined by the Markov Model of order M

This means that the element x_{n+1} is not dependent of the elements $x_1, ..., x_{n-1}$ but only of x_n. As we can see in the table 3.1 and the figure 3.2, the predictions are only dependent of the current element. This type of model only considers the current state to define the next. What happened before is not used to calculate the prediction of the next element.

A variation to the Markov model is called the *Markov chain of order M*. The order M defines the number of previous elements used to calculate the probability of the current element. Figure 3.4 shows the definition of the probability for the Markov chain of order M. As we can see, comparing it to figure 3.3, the probability of having a element x_{n+1} depends not only of x_n but also of $x_{n-1}, x_{n-2}, ..., x_{n-m}$.

Figure 3.5 shows the table representation when using the element sequence $ABCBDBCBD$ and an order of 2. As we can see, using the two last elements to calculate the next element is sufficient to achieve a perfect model where each row contains only one possibility in the second column. This means that, as long as the behavior remains constant, an order of 2 is sufficient to perfectly modelize this sequence without any error possible. Section 3.3.2 explains the different algorithms used to create the models in detail. Figure 3.6 shows the same model using the form of a graph. As the sequence is being processed, the model maintains a pointer on the current node. The nodes are represented by two letters: the last two elements that were noticed at their creation. For the current node, this represents the before last and last element of the sequence. The predicted element is the second letter of the node following the edge of the current node. This means that after the model has detected the sequence AB, following the edge, it can predict that the next element is C since the label of the next node is BC.

Finally we notice that the sequence produces a cycle in the model. Using this cycle, we are able to predict accurately the next elements of the sequence. For example, if we are in the state

Element	Successor
AB	C
BC	B
BD	B
CB	D
DB	C

Figure 3.5: A prediction table using an order of 2

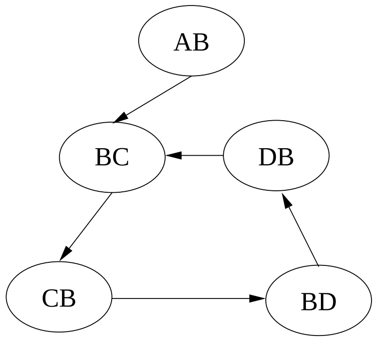


Figure 3.6: A prediction graph using an order of 2

Source	Destination
S_1	$d_{11}, ..., d_{1k_1}$
...	...
S_n	$d_{n1}, ..., d_{nk_n}$

Source	Destination
$S_{11}S_{12}$	$d_{11}, ..., d_{1k_1}$
...	...
$S_{n1}S_{n2}$	$d_{n1}, ..., d_{nk_n}$

Figure 3.7: The general definition of order 1 and 2 tables

BD, we know that the next 4 elements are $BCBD$.

Order of the model

Figure 3.7 shows the general definition of the tables of order 1 and 2. We can extend this order to the general number n and we can give four propositions. Let S be the sequence that the Markov model studies. Let us name the table representing the information at the order k as T_k. Finally, let us assume for the rest of this section that the length of the sequence is greater than $n+1$. This assumption is not restrictive since we are creating a model to predict the rest of the sequence. Generally, the construction of the model is done on a small fraction of the sequence and the predictions for the rest of the sequence.

Proposition 1. *If the transition $s_1s_2...s_ns_{n+1} \rightarrow d$ exists in the table T_{n+1}, then $s_2...s_ns_{n+1} \rightarrow d$ exists in the table T_n.*

Proof. Since $s_1s_2...s_ns_{n+1} \rightarrow d$ exists in the table T_{n+1}, this means that $s_1s_2...s_ns_{n+1}d$ exists in the sequence S otherwise the transition would not exist in T_{n+1}. Thus, we can extract the subsequence $s_2...s_ns_{n+1}d$ from S. Finally, if $s_2...s_ns_{n+1}d$ exists, $s_2...s_ns_{n+1} \rightarrow d$ must exist in T_n. □

Proposition 2. *If the transition $s_1s_2...s_ns_{n+1} \rightarrow d$ exists in the table T_{n+1}, then $s_1...s_n \rightarrow s_{n+1}$ exists in the table T_n.*

Proof. Since $s_1s_2...s_ns_{n+1} \rightarrow d$ exists in the table T_{n+1}, this means that $s_1s_2...s_ns_{n+1}d$ exists in the sequence S otherwise the transition would not exist in T_{n+1}. Thus, we can extract the subsequence $s_1...s_ns_{n+1}$ from S. Finally, if $s_1...s_ns_{n+1}$ exists, $s_1...s_n \rightarrow s_{n+1}$ must exist in T_n. □

Proposition 3. *If the length of the sequence is strictly greater than $n+1$, every transition $s_1...s_n \rightarrow d$ from T_n can be extracted by a transition from T_{n+1}.*

Proof by contradiction. Let us remind the hypothesis of the proposition $H1$ and assume by contradiction hypothesis $H2$:

- The length of the sequence is strictly greater than $n+1$ ($H1$)
- $s_1...s_n \rightarrow d$ exists in T_n but cannot be extracted from the order $n+1$ table ($H2$)

However, if $s_1...s_n \rightarrow d$ exists in T_n, this means that in the studied sequence $s_1...s_nd$ exists. If $s_1...s_nd$ exists then $\alpha s_1...s_nd\beta$ exists with α and β being possibly empty (or ϵ).
To finish this proof, let us consider each case:

1. If $\alpha = \epsilon$ and $\beta = \epsilon$, then $s_1...s_nd$ is the whole sequence. However, this means that the length of the sequence is equal to $n+1$ which is a contradiction with our hypothesis $H1$.

2. If $\beta \neq \epsilon$, then $s_1...s_n d\beta$ exists in the sequence. This means that $s_1...s_n d \rightarrow \beta$ exists in T_{n+1}. Using the proposition 2, we know that $s_1...s_n \rightarrow d$ must exist in T_n. This means the information can be extracted which is a contradiction with our hypothesis $H2$.

3. If $\alpha \neq \epsilon$, then $\alpha s_1...s_n d$ exists in the sequence. This means that $\alpha s_1...s_n \rightarrow d$ exists in T_{n+1}. Using the proposition 1, we know that $s_1...s_n \rightarrow d$ must exist in T_n. This means the information can be extracted which is a contradiction with our hypothesis $H2$.

□

These propositions show that if a predictor creates a Markov model of order n, it can extract the models of inferior orders. Algorithm 1 shows the solution to extract the information from a table of order $n+1$ to create a table of order n. The Table T_n can be considered as a set of transitions and the *Add* function can be considered as an addition to that set. This means that the addition is only performed if the element does not yet exist in the set.

```
Input: $T_{n+1}$: Markovian table of order $n+1$
Output: $T_n$: Markovian table of order $n$
foreach Transition $s_1...s_n s_{n+1} \rightarrow d$ in $T_{n+1}$ do
    Add $s_1...s_n \rightarrow s_{n+1}$ in $T_n$;
    Add $s_2...s_{n+1} \rightarrow d$ in $T_n$;
end
```

Algorithm 1: Extraction of T_n from T_{n+1}

Since all the information contained in the order n is contained in the order $n+1$, we know that if we use a higher order, no information is lost. This means that, a perfect Markov model has an order ∞ and has all the information available. Actually, for a sequence of n elements, a Markov model of order $n-1$ is enough since it can predict correctly the whole sequence. If it cannot predict anything, the model can always create a lesser order that might be able to predict an element. However, there are at least two reasons why a Markov model of order $n-1$ is not recommended. The first reason is the memory required to store the model and the second is the usefulness of such a model. Generally, a Markov model is used to predict elements in the sequence. However, if the model is being created during the sequence, no predictions can be made. For example, if we know that the sequence contains 50 elements, creating a model with an order 49 is useless since before predicting, 48 elements must be studied in order to create the first transition. For example, it would probably be best to create a model of an order of ten, construct the model using the twenty first elements and predict for the last thirty.

Finally, proposition 4 states that we cannot use a model with an inferior order to create a model with a superior order still representing the original sequence. This proposition says that, though the model can be accurate, it does not replace the original sequence. Since the history is limited, the model can create false predictions as long as there remains a node which has more than one son. If there is only one son per node, the model can be used to extract the whole sequence or patterns contained in the sequence. Of course, this is only true if the rest of the sequence does not include variations to the behavior studied during the creation of the model. The proof consists in taking an example and showing that the extracted sequence does not lead to the same Markovian model.

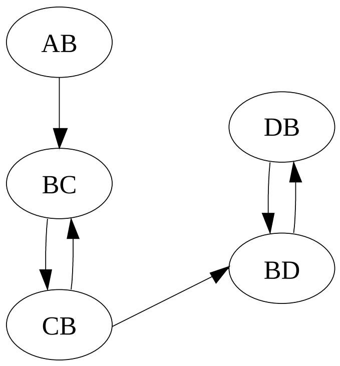


Figure 3.8: A order of 2 graph for the new sequence $ABCBCBCBCBDBD$

Proposition 4. *The Markov model of order n representing a sequence S cannot be extended uniquely to an order $n+1$ that represents the same sequence S without using the original sequence.*

Proof. Figure 3.2 shows the graph form of the model representing the sequence $ABCBDBCBD$. However, from that graph, we can create new sequences. For example, the following sequence $ABCBCBCBCBDBD$ can be extracted from the graph. This sequence, though created from the model is not equivalent since the Markov model of order 2 for that sequence is different compared to figure 3.6 as we can see in figure 3.8. Finally, the graph obtained does not represent the original sequence. This means that, without additional information or the original sequence, we cannot extrapolate the order 1 to the order 2. □

3.2.3 Dynamic representation

We have shown that a Markov model helps to understand and to calculate predictions using prior elements in the sequence. The order of the model defines how many elements are used to calculate the next element. Also, we have shown that the higher the order, the more precise is the model since information is never lost when incrementing the order. However, with only a model of a particular order, it is impossible to calculate a model of a higher order.

This means that, when creating the model, the right order must be used. If too low, the predictions can be inaccurate. If too high, the complexity and memory usage of the model can be unnecessarily too important. To use such a model in a dynamic framework, these factors must be taken into account.

Memory usage is an important factor. A model using too much memory has a risk of generating many cache misses. Also, on embedded systems, the amount of memory available is limited. A second factor is the complexity to calculate the prediction. Since a lookup into the model is done frequently, a fast algorithm must be used to quickly decide what to prefetch.

These two factors need to be taken into account when defining different parameters used for the creation of the model.

3.2.4 Predictions using a Markov model

When implemented into a dynamic framework, the model must be able to calculate the prediction with the lowest complexity possible. Moreover, if it prefetches data into the cache, the model

Element	Successor
A	B (1)
B	C (2),D (2)
C	B (2)
D	B (1)

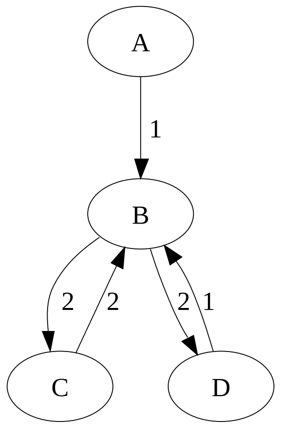


Figure 3.9: Extended versions of the models

must be able to predict elements further in advance. This means the model must predict n elements in advance and as accurately as possible.

We presented two model representations: one using a table and one using a graph. There is an additional information needed to be able to predict. If there is more than one choice possible from a given state, the predictor must know which is more likely to happen. This can be done by counting the number of times the model notices one transition compared to the other ones. For example, figure 3.9 shows the extended representations if we add this information into the models.

For the table representations, the numbers in the second column are the number of times the different transitions were noticed. This same number is also the label of the edges in the graph representation. This means that, if the last element was B, the model would not be able to decide between C or D since both appeared the same number of times. However, in a more general context, the label of an edge may be higher than the other edges. Therefore, this would be the most probable son and the best prediction.

3.2.5 Table vs. Graph

This subsection addresses the problem of deciding between both representations which one is the best. The system must:

1. *Use the least memory possible*: the table representation requires having an array big enough to hold all the information. Though dynamically allocated arrays are a solution, the constant reallocation can be prohibitive in terms of performance. The graph representation uses more memory since pointers need to be allocated for the edges between the nodes. However, each node is allocated when it is needed and the total size of the graph does not need to be calculated before the start of the construction.

2. *Have the lowest complexity possible when calculating the prediction*: when using a table representation, the system needs to calculate the position of the row containing the studied sequence. This can be done with a hash function but is still relatively costly. The graph version is faster since edges exist between nodes to directly give the next state of the model. The complexity depends on the number of edges leaving a node. However, optimizations

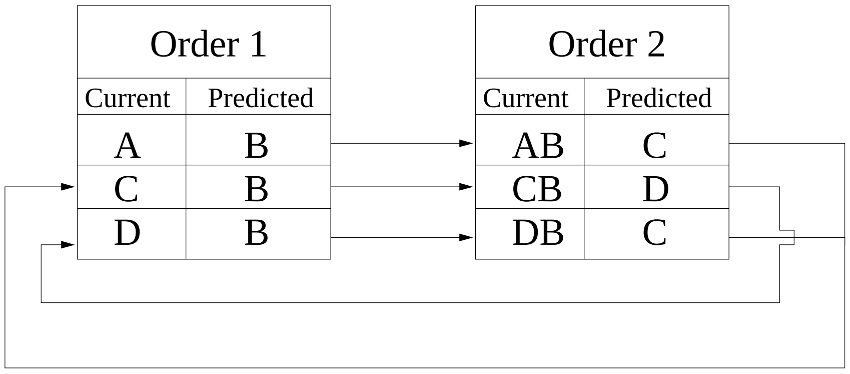


Figure 3.10: Optimized table representation

can be implemented to accelerate the search by keeping certain edges as *most probable edges*. Details of this technique are given in section 3.3.1.

3. *Be able to handle mispredictions*: when the model is unable to predict or the current behavior does not reflect the behavior the model noticed during its construction, it must still be able to predict quickly. In the table representation, if we only have the order n, the predictor must wait n elements before being able to predict correctly. However, if we ignore the memory usage, the model can build the tables of order 1 to n and use each order when appropriate. For the graph representation, it is not necessary since, while creating the graph of order n, we also create the orders 1 to $n-1$. Therefore, at a lower cost, we already have the different orders available for use when a misprediction occured since the graph representation does not necessitate more memory to include this information compared to the table representation.

3.2.6 Dynamic solution

Though solutions are possible to create a good and fast table representation, another problem is the memory allocation scheme used. An array is generally defined as being a contiguous memory block. This means that a general idea of the size of the structure before creation is required. Though a dynamic reallocation scheme can be used or a special data structure can be created to allow only portions of the table to be contiguous, the complexity of the model creation or the complexity of the lookup are augmented.

If we ignore this problem, a solution must still be found in order to predict effectively. Though hash tables are a solution to find an entry in a table, using a hash function at each prediction would probably be too costly in a software solution. Figure 3.10 shows one solution to optimize the time needed to make a prediction. Two tables representing the first two orders are augmented by links between different cells. These links represent the different predictions and, if the predictions were correct, one of the next states the model would be in. Therefore, instead of resorting to a hash function at each prediction, the predictor would first check the links.

If we suppose that the model contains a pointer to the cell that represents the current state of the model, the model can start by checking the different links of the current cell. Each time the model receives information about the program behavior, it can update this pointer to always represent the best state. The system starts by checking if the current behavior leads to one of the cells that are linked to the current one. If the check failed, a normal hash search is performed. However, if it succeeds, a call to the hash function is not needed since the predictor directly has the next current cell. Also, the links can be used to perform predictions from each cell. Hence, by using links as predictions of the next element in the sequence, the model can, at each call, predict and prefetch elements.

As we can see, there is no entry B in the first table and certain 2-order entries are also missing. This is done to minimize the memory usage of the model. If we consider the first column of figure 3.5, we notice that every couple finishing by a C or a D is followed by a B. This means that we do not need to know what preceeded a C or a D in order to make a prediction. This is why we can remove the entries from the second order. The removal of B is performed because, if the predictor does not know what preceeded the B, it cannot predict accurately. Another solution is to leave the B in the first order table and, if that is the current state of the model predict either C, D or even both.

Though different schemes or decisions can be made, augmenting the table representation with prediction links almost leads the predictor to a graph representation with the allocation problem of a table representation.

The graph

The previous section briefly presented a representation of a Markovian model using a certain number of tables. This section shows how some of the presented ideas naturally lead us to a graph representation to handle the model and predict efficiently. First, since a graph is generally allocated dynamically, each node can theoretically be added at any time.

Figures 3.2 and 3.6 showed two possible graph representations of fixed order Markovian models. One important factor for the predictor is to be able to stay at the highest order as long as possible, but if there is a misprediction or a slight change in the behavior of the program, it must be able to use the lower orders.

Having separate graphs for each order is not practical in a dynamic optimizer since too much memory is required. However, the previous section presented a technique using links between orders to lead the predictor from one order to another. This idea can be used in the graph representation.

Figure 3.11 shows this kind of representation on the sequence $ABCBDBCBD$. The square nodes represent the nodes from the first order. Links from the first order lead to second order nodes since we keep the information from the previous order. For example, if the sequence is AB, we have an edge from the node A to the node AB but no edge between the two nodes A and B. Once in the second order nodes, the predictor does not return to first order nodes unless a misprediction occurs.

This can happen if, for example, the current state of the predictor is the AB state. If the next element is D for example, the predictor cannot go to a state BD since there is no edge between both states. However, it could create such a link if needed and this could be used the next time the predictor is in the AB state.

However, maintaining and updating edges between the nodes of such a graph is complex and time consuming. In a dynamic system, this is not an option. Therefore, the predictor creates the graph and then stops inserting new nodes or new links. This, of course, makes the graph valid for a certain amount of time but, at a phase change for example, a new graph will be needed. The next subsections show a few extensions that were added to render the model adaptable to slight behavior changes.

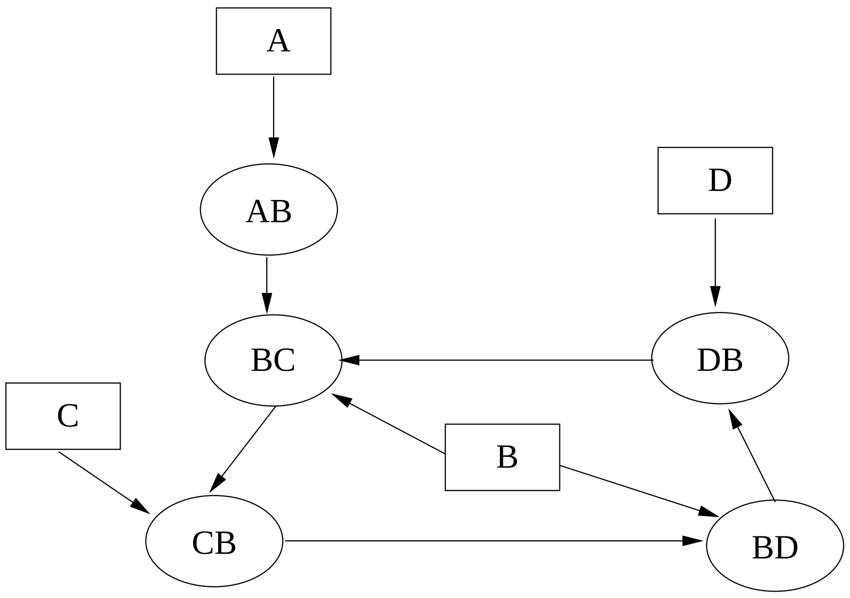


Figure 3.11: The order 2 graph representation including the first order. First order nodes are represented with square boxes

3.2.7 Prefetch distance

The prefetch distance is defined by the number of elements between the current element and the element predicted. The distance must be carefully calculated since a small distance results in the prefetch element arriving too late and, if the distance is too high, the system risks a cache line eviction before the data is needed. Indeed, when the prefetch data arrives, it is placed in a cache line that can be used for another line of data if that is what the replacement policy decides. This means that the prefetched data must arrive as close as possible to the time it is needed to limit these risks.

A drawback when using a higher distance is the drop of accuracy. Since each step in calculating the prediction adds a certain risk of error, the chances the predictions are wrong are greater if the distance is important. Table 3.1 shows three different traces and the results using the Markovian model. The first column shows the three traces that were repeated 200 times and the models were constructed using the first 30 elements of the sequence. The second and third column show the different parameters used. We used 4 different depths for the construction of the model. For example, 3 means that the model will consider the last three elements to predict the future access. For each trace, we compared different prefetch distances.

As we can see, the higher the construction depth, the greater number of nodes in the model. However, the higher order models are able to predict with better accuracy, especially when the prefetch distance is increased. As we can see, using a prefetch distance of 20, only the model of order 4 is able to predict correctly for each trace.

Another important consideration is the quantity of calculations made by the predictor. The time needed to finish the calculation depends on this distance. For example, for the Markovian model, if n is the prefetch distance, n nodes must be visited to calculate the predicted element. Since the goal of the model is to be as quick as possible, the solution is to keep the result in the current node. This means that, when a prediction is needed, the system already has a pre-calculated prediction.

Trace	Const. Depth	Pref. Dist.	Accuracy	Size
1-2-3-4-3-2-1-4-3-2-1-5-3-2	1	1	71.32%	5
	2	1	71.36%	14
	3	1	92.71%	27
	4	1	99.82%	41
	1	3	14.31%	5
	2	3	21.34%	14
	3	3	92.56%	27
	4	3	99.64%	41
	1	20	0%	5
	2	20	0%	14
	3	20	7%	27
	4	20	98.42%	41
1-2-1-4-1-2-3-1-4-5-1-2-3-1-5	1	1	53.30%	5
	2	1	86.56%	14
	3	1	93.6%	27
	4	1	99.83%	47
	1	8	0%	7
	2	8	46.38%	19
	3	8	53.05%	33
	4	8	99.33%	48
	1	20	0%	7
	2	20	26.27%	19
	3	20	32.88%	33
	4	20	98.53%	48
1-2-3-4-5-4-3-2-1	1	1	77.6%	3
	2	1	77.56%	8
	3	1	77.62%	15
	4	1	99.72%	24
	1	8	22.12%	3
	2	8	11.08%	8
	3	8	33.18%	15
	4	8	99.44%	24
	1	20	0%	3
	2	20	0%	8
	3	20	0%	15
	4	20	97.5%	24

Table 3.1: Different traces using the model. Each trace was repeated 200 times, the model was constructed using 30 elements of the sequence.

Listing 3.1: The SNode structure

```
//Prediction node
typedef struct snode
{
int stride;                //Information of this node
int predStride;            //Prediction
struct snodelist *next;    //List of the sons
}SNode;
```

Of course, since the labels of the edges are continuously updated, a new better prediction can appear. Using a node version counting system presented in section 3.3.3, the system is able to update the predictions when needed.

3.3 Implementation

Section 3.2 presented the theory behind the Markovian model. We compared two possible representations of a Markovian model. Since the memory used and the complexity of calculating the predictions with the table representation were considered too costly, the graph representation was chosen as the best solution.

This section presents the implementation choices that were made to create a dynamic graph representation of the Markovian model. The system was implemented to predict memory accesses. To achieve this, memory strides between consecutive accesses were modelized and stride predictions were performed. The rest of this section is as follows. First, the general structure of the model is presented. The information that is needed and the information available is considered. Since the model is kept in memory, it is possible that data cache misses are caused by the model. Spatial locality is considered to optimize the global system. Optimizations are then explained to reduce the overhead of the system and the construction and usage of the model is presented. Finally, the associated API is presented.

3.3.1 The Markovian structure

The markovian structure used by the graph is divided into different structures. Two main structures are used to create and handle the model. The first represents the nodes of the graph. The second structure is used to handle internal counters and the general state of the model.

Listing 3.1 shows the definition of the structure for the nodes of the graph. The contents of the node are the stride that represent the node, the precalculated stride and the next elements of this node. The *next* field of the structure is a linked list containing a pointer to a son of the current node, a counter which indicates how many times the model followed the edge and a pointer to the next element of the list.

The construction of the model requires a complex algorithm. To reduce the overhead to a minimum, the model is not continuously updated. When the model is not updated, it is used to predict elements. If the predictions are not accurate, the model is destroyed and reconstructed. The general structure is shown in the listing 3.2. This structure contains every element needed in order to handle the model, check the construction phase and the miss predictions.

The fields of the Markovian model are:

- *depth*: the depth of the model, i.e., the number of elements used to calculate the predictions;

Listing 3.2: The SMarkov structure

```
typedef struct sMarkov
{
int depth;            //Depth of the Markovian model
int nbr_err_max;      //Maximum of consecutive miss predictions
                      //tolerated before a flush
void* address;        //Current address,for the prediction we add this address
                      //to the predicted stride
SBinNode *root;       //Root of the binary tree
SNode **cur;          //Pointers on the graph

//Function that will be used by the programs that want to be optimized
void (*fct)(struct sMarkov*, void *);
int fct_ttlmax;       //Maximum Time to live for the construction function
int prefDist;         //Distance of the markovian model
                      //(i.e: number of jumps we make in the graph to predict
                      //in advance)
}SMarkov;
```

- *nbr_err_max*: the maximum number of miss predictions before a flush of the model is performed. A flush occurs when the number of current consecutive mispredictions exceeds the value in *nbr_err_max*. This flush resets the model and enables the system to construct a new model that will represent the current behavior of a program;
- *address*: the current address that the program has accessed;
- *root*: the root node of the binary tree linked to the graph representation;
- *cur*: the current node;
- *fct*: the function pointer to the state function entry;
- *fct_ttlmax*: the maximum number of elements used to construct the model, once this number is reached, the model is fixed and prefetching can start;
- *prefDist*: the prefetch distance used by this model.

Though most of the elements contained in the structure do not require more information, the *fct*, *root* and *cur* fields requires a little more attention. The signature of the pointer function *fct* states that two arguments are required. First, a pointer to the Markov structure in order to update the internal counters. Second, the address of the considered load. With these two elements, the function is able to update the structure and use the address to create the model or check the accuracy of the predictions.

Figure 3.12 shows the different states of the model. At first, the model enters the first call state. This first state sets the address field of the structure and sends the model into the construction phase. After *fct_ttlmax* calls, the state is sent to the prediction phase until the end of the execution. However, if there are too many consecutive errors, the model resets itself and the state is set back to the *First call* state. In order to achieve these state changes, the system simply updates the pointer function.

The model is accessed through the *fct* pointer function and, at each call, the system can update the model or use the model to predict and prefetch a data element. Since each node

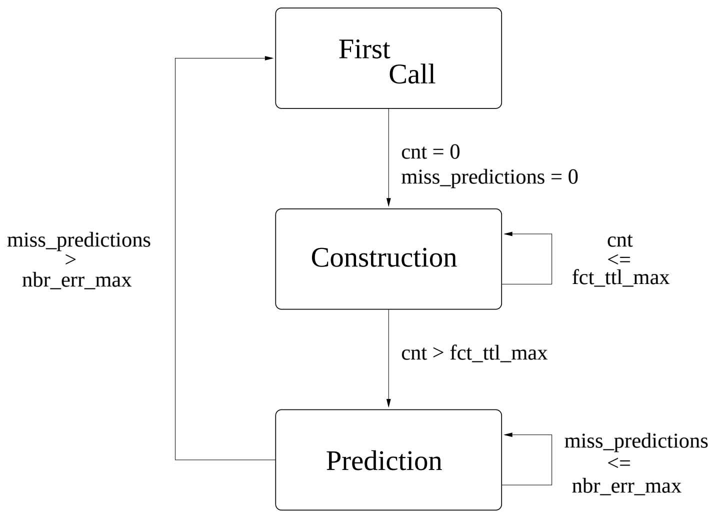


Figure 3.12: The state machine of the model

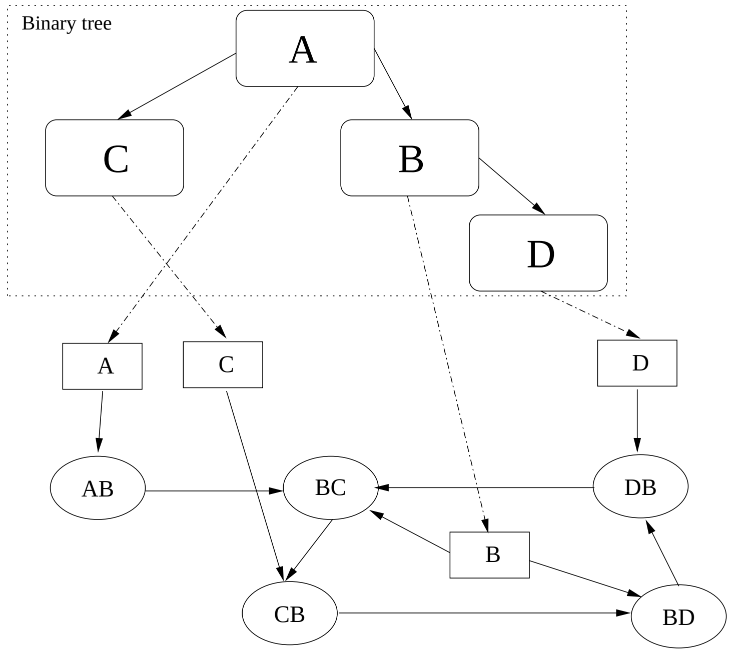


Figure 3.13: The binary tree, used to accelerate the search of the first order nodes

represents a certain state of the model, the model needs to store the current node when updating or predicting. The current node is stored in the field called *cur* of the *SMarkov* structure. For example, if the program informs the model that the next access is A, the model checks if there is a son of the node pointed by *cur* that is labeled A. If so, the label of the edge is incremented, and *cur* is updated.

However, when miss predictions occur, it is necessary to return to the first order states shown in the figure 3.11. This means that, whenever the model is looking for a son that the current node does not have, the system has to search the different root nodes to find it. A binary tree is added to limit the overhead as can be seen in figure 3.13. The root of the binary tree is the *root* field presented in the *SMarkov* structure. If the model finds the root node it was looking for, the model can continue predicting from that node. At each call, the model goes from the current node to one of the sons. This means that after a certain number of updates, the model will be at the maximum depth.

Spatial locality optimization

The model is constructed during the execution of the program. Nodes are created, edges are attached, internal counters are updated. Every access to the model may cause data cache misses that can slow down the program. To reduce the risk of modifying the behavior of the program, the system allocates a data block in which the whole model can reside.

The memory allocation system is handled internally using a pointer to define the next free block available and a free-space counter. If the counter reaches zero, there is no more memory available. The memory that we initially allocated is less than 300 kilobytes and can store a model of over 4000 nodes. This limit has never been reached in the considered benchmarks.

This allocation scheme has two consequences: spatial locality and a simple solution to flush the model. Spatial locality is the successive accesses to elements residing in the same memory region. Since the system now handles the allocation inside the block, it is able to concentrate the nodes of the model together, thus limiting the number of data cache misses. Second, when a model flush is required, the system can simply modify the *root* and *cur* pointers and the allocation pointer to flush the model.

Best and not so best pointers

The Markovian model contains a pointer to the node that represents the current state of the model. The edges of the node lead to the next strides that were monitored during the construction of the graph when it was in that particular state. Since a node can have a certain number of edges, a solution must be devised to limit the calculations needed to find the node with the correct stride.

Each node contains the edge that was the most followed. By doing this, the system is able to check the most probable edge before having to search the list of edges. Also, each node contains two other likely edges that are tested before the list. These tests limit the number of times the list of nodes must be checked. Listing 3.3 shows fields added to SNode structure presented in listing 3.1. The first element that is added is the *vis_probableNext* counter and *probableNext* pointer. If the stride contained in *probableNext* is the correct stride, the counter is incremented and the probable son becomes the current node of the model.

However, if the most probable son is not the one that the model is looking for, the model checks two other nodes *last* and *last2*. These two nodes act as a modified *Least Recently Used* (LRU) system. If one of the nodes corresponds, then the associated counter is incremented and

Listing 3.3: The new edges in the node structure

```
//We have the number of times we used the edge to the most used son
int vis_probableNext;
//and a pointer to that son
struct snode *probableNext;

//We then have a LRU system between the two last sons that followed this node
//And of course the number of times we followed their edges
int vis_last;
struct snode *last;
int vis_last2;
struct snode *last2;
```

a check is performed to see if it is now more probable than *probableNext*. If this is the case, the two pointers and two counters are interchanged. The full algorithm is presented in section 3.3.3.

3.3.2 Algorithms

The previous subsection briefly presented the implementation choices for the graph representation of the Markovian model. This subsection first presents the different phases of the construction of a Markovian model, and then presents the different algorithms used to construct, predict and update the model.

Example

Before explaining the algorithms in detail, we present the different steps required to construct a Markovian model using a simple example. The examples use the same sequence $ABCBDBCBD$ as before. Figures 3.14 and 3.15 show respectively the construction of the Markov models of order 1 and 2. The rest of this section explains the constructions of these two models to explain in depth how the model functions. First, we explain the first order since it is easier to understand.

The Markov model of order 1

To explain the construction process, we enumerate each step and present the modifications to the model for each processed element. Each enumeration is associated to the subfigure presented in the figure 3.14. For example, the enumeration a explains the construction process of the first element of the sequence and refers to the figure 3.14(a).

a) when the model receives the first element A, it creates a single node A. This node has no parent since the model does not know what happened before and no son since the model does not know what will happen next;

b) after the second element B is processed, a node B and an edge from A to B are created. This edge signifies that after an A the model noticed a B and the label means that it noticed this behavior once;

c) the third element C is then processed and a node C is created. As we can see, an edge is now attached from B to C for the same reasons as before;

d) when the fourth element B is processed, another node B is not needed since there is already one in the graph. Instead, we can notice that a cycle between B and C is created;

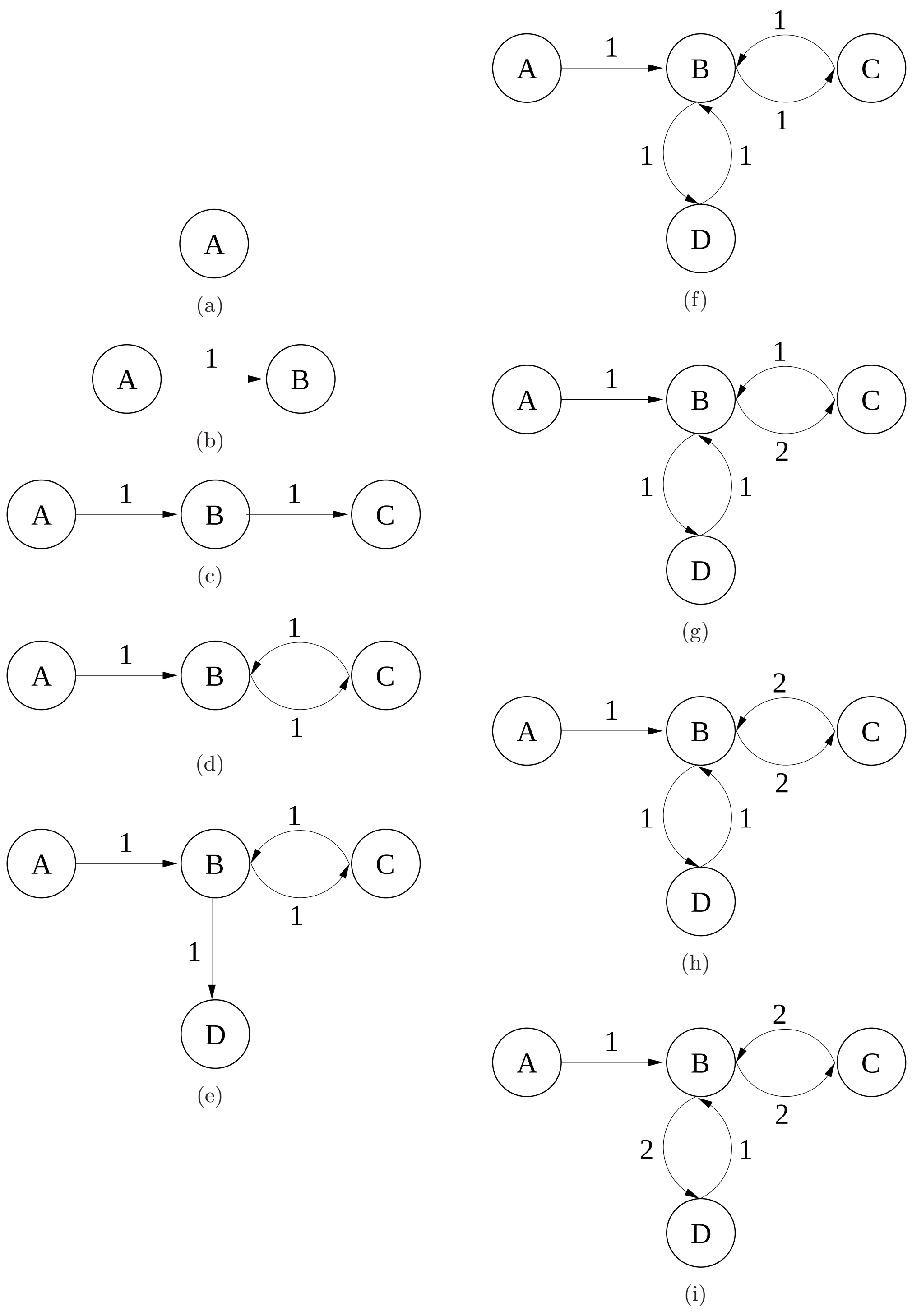


Figure 3.14: The construction of the Markovian models of order 1

e) D is the next element of the sequence and it is a new distinct element. This means that a new node D is needed and attached to the node B;

f) like figure 3.14(d), when the next element B is handled, a cycle between B and D is created;

g) the next element C is then handled and the model does not need to create a new edge or a new node. The current node is B and the next element is C. Since an edge exists from B to C, a simple incrementation is performed, meaning that this edge was used twice;

h) after the next B is handled, the labels of the cycle between nodes B and C are both at 2;

i) Finally, when the last element D is processed, the label of the edge from B to D is also incremented.

The Markov model of order 2

Let us now consider a Markov model of order 2. This means that, instead of only using the last element to calculate the predictions, the model considers the last two elements. Figure 3.15 shows the construction of this model. Using the same explanation technique as before, we enumerate the different steps of the construction.

a) after the first element is processed, we notice that the figures 3.14(a) and 3.15(a) are identical;

b) however, when the second element B is processed a difference can be noticed between figures 3.14(b) and 3.15(b). The edge between nodes A and B exists but a second node B was created. The difference between both nodes B can be defined as this:

   - The square node B, which represents a first order node, can be translated as follows: *From this node B, the model does not know what happened before or what is going to happen*;
   - The circle node B, which represents a second order node, can be translated as follows: *From this node B, the model knows that the previous element was A but does not know what is going to happen*;

c) once the model has processed the third element, two more nodes are created. The square node C and the circle node C are respectively the same type as the square node B and the circle node B in the previous figure. However, the edge from the circle node B in this figure to the new circle node C adds new information to the model. Indeed, the model now knows that if the last two elements were AB then the next element is likely to be a C. The destination of the edge is not the square node C, since this is a second order Markov model that uses the two last elements to calculate a prediction. The square node C is a first order node and this means that it is better to connect the B node to the circle node C because that node is a second order node. Finally, the circle node C is defined as *a C that was noticed after a B* which is precisely the case;

d) when the next element B is processed, a new node B is created and attached to the square node C as we can see in figure 3.15(d). The node is also attached to the circle node C since the model noticed that if the last two elements are BC then the next element is B. As we can see, every edge label is 1 since we followed these edges only once. However, a new square node B is not created since it already exists;

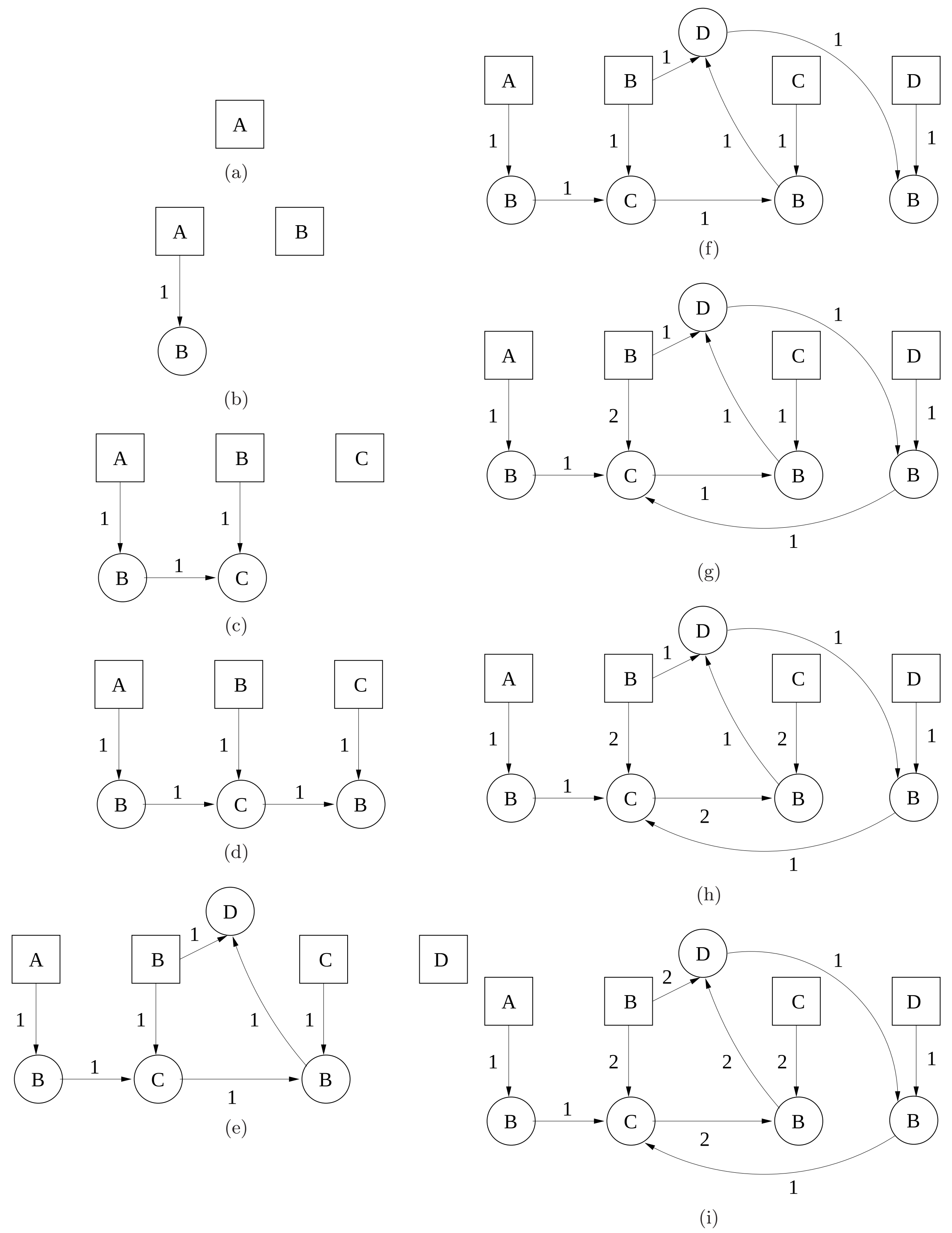


Figure 3.15: The construction of the Markovian models of order 2

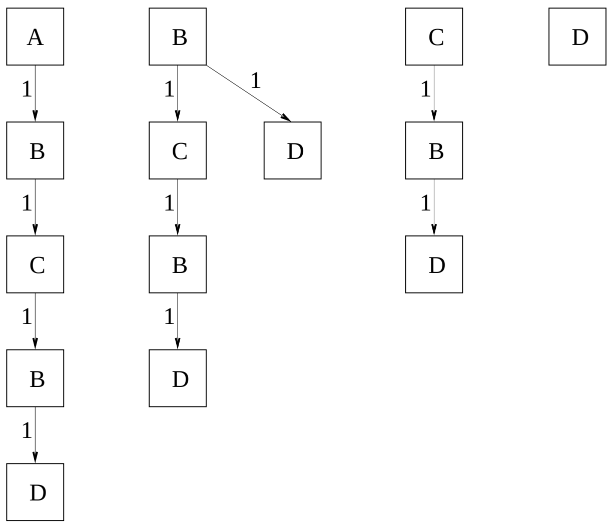


Figure 3.16: The construction of a higher order Markov model

e) after D has been processed, two more nodes are created the same way the model created the two C nodes;

f) the model then handles the element B, another node B is created and, at this stage, no more nodes will be created. We can now see that this representation includes both Markov model orders at the same time. Let us compare the figure 3.14(f) with figure 3.15(f), A, C and D have each only one edge leading to B and followed these edges only once. From the node B, there are two possibilities: C and D and these edges also have a label 1. However we have additional information in this second order model: we know that after the sequence CB the next element is D and after DB the next element is C;

g) when the next element C is processed, the last edge of the model is created. We also notice that the label of the edge from the square node B to the node C has been incremented. At this stage in the sequence, this is the second time the model noticed that, if only considering that the current element is B, the next element was twice C;

h) as for the handling of the last element, when the model processes the next element B, the edges are simply incremented;

i) finally, when the last element is processed the model is in the state shown in figure 3.15(i).

The construction algorithms

The previous section presented the construction of the graph with an example. This section presents the algorithm of the construction of the graph and explains in more detail certain aspects of the construction. The construction of the graph is performed in two different phases. First, it is merely the addition of nodes to the graph as we saw in the figures 3.15(a) and 3.15(b). During this stage, the graph is actually an accumulation of separate trees where each node has at maximum one parent. Figure 3.16 shows the construction of the model with an order superior to 5 and using the 5 first elements of the same sequence. Four trees were created since we had 4 different elements in the trace. Since the first node of a tree represents the first order nodes, a root of a tree can be defined as *without knowing what happens before this element, this is what*

Transitions				
Starting	A	B	C	D
	$A \to B$ $AB \to C$ $ABC \to B$ $ABCB \to D$	$B \to C$ $B \to D$ $BC \to B$ $BCB \to D$	$C \to B$ $CB \to D$	

Table 3.2: The transitions that can be extracted from the graph representation in figure 3.16

happens after. This means that, because there are 4 different elements in the trace, the model needs 4 different trees.

The root node B is the only node that has two sons. This is also logical since, if the last element was a B and we only have that information, then the next element is either a C or a D. Also, table 3.2 shows the different transitions that can be extracted from the graph representation. As we can see, we are able to extract every order inferior or equal to 4, since 4 is the highest depth of the graph.

```
Let cnt be the number of iterations this algorithm has been executed, initialized at 0
Let depth be the depth of the construction
Let cur be the array of pointers on the current nodes of the graph
Let elem be the last element monitored by the model
foreach node cur[n] in the array cur do
    if cur[n] does not have a son labeled elem then
        Create a node labeled elem and an edge labeled 1 from cur[n] to this node
    end
    The pointer cur[n] is now assigned to this son
end
if a root node labeled stride does not exist then
    Create a root node labeled elem and add it to the graph
end
Add a pointer to the root node labeled stride to the array cur
Increment cnt
if cnt >= depth then
    Finished, start second phase
end
```

Algorithm 2: Construction algorithm (First part)

Algorithm 2 explains the first part of the construction. This part is used during the first steps. The number of iterations in this part of the algorithm is equal to the depth or order of the model. At each iteration, the model is called and must process the new element *elem*. The model keeps pointers to a certain number of nodes stored in *cur*, called current nodes. At each step of the construction, the model checks each pointer, if the pointed node already has a son labeled *elem*, it increments the label of the edge and assigns the pointer to that son. However, if the node does not have such a son, the model creates a node labeled *elem*, attaches it to the pointed node via an edge labeled 1 and assigns the pointer to the new son.

The last part of this algorithm checks the different roots of the graph. Since the graph possesses different trees, each root is tested and if a label is equal to *elem*, a new pointer is added to *cur* and assigned to that root. If not, a new root is created and a pointer is also added to *cur*.

Finally, if the number of calls is sufficient, the algorithm is finished and the second part of the construction can be used. The number of calls is equal to the order of the model. This first part is executed at the beginning of the construction and we can see that, in the case of an order 1 model, this is only done once. Figure 3.14(a) shows this state. However, when using a second order model, two iterations are needed as we can see in the figures 3.15(a) and 3.15(b).

```
Let depth be the size of the array cur, it is also the order of the prediction model;
Let elem be the last element monitored by the model
Let cur be the array of pointers on the current nodes of the graph, index from 0 to
depth-1
Let cur[iter] be the pointer that is at the maximum depth
foreach node cur[n] from the array cur except cur[iter] do
    if cur[n] does not have a son with the element elem then
        The node cur[n] creates a son labeled elem with an edge labeled 1
    end
    The pointer cur[n] is assigned to this son
end
if cur[iter] does not have a son labeled elem then
    Create an edge between cur[iter] and the node cur[iter-1] (the value of this pointer
    has already been updated)
end
if there is no root node labeled elem then
    Create the root node labeled elem
end
cur[iter] is assigned to the root node labeled elem
Decrement iter (modulo depth);
```

Algorithm 3: Construction algorithm (Second part)

What defines the end of the first part of the algorithm is the fact that the maximum depth of the graph is now the depth of the first tree that was created. In the previous figures, this was always the tree on the left. We showed that in the case of a second order model, after two iterations, the left-most tree already has two nodes. If we look at this state, the model contains a tree that is defined by a root node A and a son B. However, there is also another tree that only has one root node labeled B. After the next element C is processed, the model adds a son to the square node B and this son is labeled C. We have shown that this node must be the same node as the one that is added to the root node B. If this was not the same node, then the trees would still be independent and the left-most tree would have a depth of 3 instead of 2. This would mean that the son of the node C would be calculated using the last three elements of the sequence instead of only the two last. This is why the trees can no longer be independent.

That said, the major difference between both parts of the construction algorithm resides in the fact that the model does not, at first, add edges between the different trees. It only starts to do so when the depth of the leaf nodes is equal to the order of the model. Once that is taken into account, the algorithm is quite straight-forward.

Algorithm 3 shows the second part of the construction phase. There is, at each iteration, a single element in the *cur* array that points to a node that we will call $node_{max}$ at the maximum depth. $node_{max}$ must not add an edge to a new unique son, since it would then create a tree with a height superior to the order of the model. However, the algorithm also shows that the node $node_{max-1}$ pointed by *cur[iter-1]* is a node with a depth equal to the order-1 (For simplification reasons, if iter is equal to 0 then we define that iter-1 is actually equal to depth-1). The algorithm 3 starts by updating every node except $node_{max}$. Once that is done, we know that the old $node_{max-1}$, since it has been updated, now has a son labeled *elem*. This son might be a new graph node or was created during a previous call. In any case, the pointer $node_{max-1}$ is now assigned to that son and it is a node whose depth is equal to the order of the model. Also, from the root of the corresponding tree to this leaf, the sequence of labels are the last n elements that the model has processed. The model then creates an edge if it does not yet exist from $node_{max}$ to $node_{max-1}$. Once that is done, the pointer *cur[iter]* is assigned to a root node labeled *elem*. Again, if this root node does not yet exist, it is created.

The algorithm requires that, at its entry, *cur[iter]* points to a node of depth the order of the model. Before the update this was *cur[iter-1]* (or *cur[depth-1]* if iter is worth 0). This is why, at the end of the algorithm, the counter *iter* is decremented modulo the order of the model.

3.3.3 Graph traversal

Once the graph is constructed, it can be used to predict elements and prefetch data. To achieve this efficiently, the model must be able to prefetch elements in advance using a parameter called the *prefetch distance* as fast as possible and must be able to check for accuracy. The prefetch distance is defined by the number of elements between the current element of the sequence and the one that is predicted. This section presents the prediction algorithms, the optimizing techniques to lessen the overhead and the solution to check the accuracy of the model.

Prefetching

The created Markovian model uses the strides between the different references that are provided by the program. As the model receives a sequence of elements, a pointer is maintained as the current state of the model. When the program accesses a data reference, it is passed to the model that either continues the model as previously explained or predicts a future data reference. This prediction is later sent to the prefetcher.

One solution to calculate the prediction is to follow the most probable son. Since the edge to that node is the most followed edge from the current node, the label of the node is also the most probable stride. However, since the predictor may want to predict two elements in advance or even further in the future, it would have to sum the different most probable sons and walk through a part of the graph to achieve this.

Even though this is a possible solution, it is not very practical in terms of performance. This is why a precalculated stride is used. A precalculated stride is included in each node and is recalculated when there is a change in the graph that affects the calculation. This way, when the model needs a prediction, it only needs to retrieve this value. However, different checks are needed to maintain the model in the correct state.

Algorithm 4 presents the update and the prediction mechanism. At the end of the construction, the model still has multiple pointers on different nodes of the graph. The predictor keeps the pointer on the node with the highest depth and updates its position at each call. The

```
Let cur_node be the current node
Let last_Stride be the last monitored stride of the program
Let pred_Stride be the predicted stride from the current node
Let probableNext be the pointer to the most probable next current node
Let last and last2 be the last two sons of the current node that were accessed after the current node
Let vis_probableNext, vis_last, vis_last2 be the number of visits to each corresponding node
Let sprobableNext, slast et slast2 be the labels of each corresponding node
if cur_node == NULL then
    Search for a root node labeled last_Stride
    Assign the result to cur_node (if the root does not exist, the result is worth NULL)
    Increment the miss prediction counter
    No prediction this time
else
    if sprobableNext == last_Stride then
        Increment the counter vis_probableNext
        cur_node = probableNext
        Predict the stride pred_Stride from the new cur_node and add it to the base address to make the
        prefetch
        Set the miss prediction counter to 0
    else
        Increment the miss prediction counter
        if slast == last_Stride then
            Increment the counter vis_last
            cur_node = last
            Predict the stride pred_Stride from the new cur_node and add it to the base address to make the
            prefetch
        else
            if slast2 == last_Stride then
                Increment the counter vis_last2
                cur_node = last2
                Predict the stride pred_Stride from the new cur_node and add it to the base address to make
                the prefetch
            else
                Search for the son labeled last_Stride if the node is found then
                    The current node cur_node is assigned to this son
                    Predict the stride pred_Stride from the new cur_node and add it to the base address to
                    make the prefetch
                    Update the label of the edge leading to this son
                    last2 = the new son
                else
                    Search for a root node labeled last_Stride
                    Update cur_node (NULL if no such root node)
                end
            end
        end
    end
    Update pointers last, last2 and probableNext
end
```

Algorithm 4: Update and prediction algorithm

program calls the model and provides the last accessed address. First, the model calculates the stride by subtracting this latest address with the last address.

Once this is done, it compares the stride to the different sons of the current node. First it tests the most probable node. If the stride matches the label of that son, we know that there wasn't a miss prediction since the parents of this node used this stride to calculate their own predictions. Therefore, the counter representing the miss predictions is set back to 0. Finally, a prediction is issued using the precalculated value associated to the node, the *vis_ probableNext* is incremented and then the current node is updated.

However, if the label of the most probable son does not match with the latest stride, we know a miss prediction occured and the miss prediction counter is incremented. The model then tests two possible candidates *last* and *last2*. If either match, the corresponding counter *vis_ last* and *vis_ last2* is incremented and the current node is updated. Finally, if the counter is superior to *vis_ probableNext*, the two pointers and associated counters are exchanged and the prediction of the current node is recalculated.

If none of the three labels matched, the model looks for another son with a matching label. If it exists, the pointer *last2* and the counter *vis_ last2* are assigned to this node. If not, the model looks for a root node labeled correctly and assigns the current node accordingly.

Prediction calculation

Once the construction phase is finished, the model starts predicting elements. At this stage, two possibilities can be adopted to calculate the predictions: pre-prediction or *just-in-time*.

The pre-prediction calculation is a traversal of the graph that calculates the predictions for each node. This is done at the end of the construction phase and just before any prediction. The advantage of this technique is to perform the calculation once and for all and, during the prediction, the model knows that every prediction has been calculated. This means that a test *is the prediction calculated?* is not needed. The down-side of such a technique is the overhead incurred. In the case where the majority of the nodes are visited periodically, this does not have a big impact since, for each node, a calculation of the prediction is needed. However, in the case where the graph has many nodes that are not used during the prediction phase, this can be a major issue.

The *just-in-time* technique uses a flag to signal if the precalculation has been performed. Before a prediction is made, the system checks the flag and, in case the calculation has not yet been performed, it calculates the prediction stride. The advantages and drawbacks of this method are the opposite of the pre-prediction calculation technique. Since the predictions are only calculated on a need-to-use basis, the problem of using the whole graph during the prediction phase is irrelevant. However, the continuous check of the prediction flag is a problem since it heightens the overhead during the prediction phase.

Finally, since the calculation phase is defined as being a minor phase compared to the length of the prediction phase, the pre-prediction technique was chosen for the calculation of the predictions. This was decided since a rare calculation that incurs a large overhead was deemed less costly that the possibility of frequent tests during a large period of time.

Node version number

Though the predictions are calculated before the prefetching phase, the model continues to update the labels of each edge. If an edge becomes the new most probable edge, a recalculation

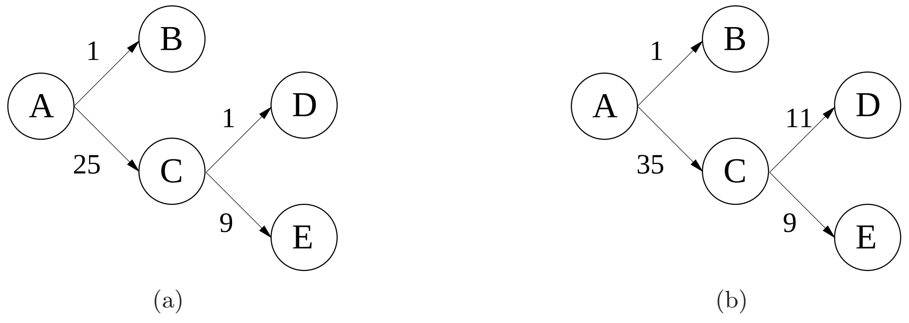


Figure 3.17: The evolution of the labels on each edge of the graph

is needed. Figure 3.17 shows the evolution of certain nodes in a graph. In the figure 3.17(a), we can see 5 nodes with different label values for the edges.

The most probable son for the A node is C. The most probable son for the C node is E. This means that if we were to predict two elements in advance from A, we would predict E. Furthermore, if the labels of the nodes were strides, the model would actually predict that the full stride is $C + E$.

Figure 3.17(b) shows the state of the model after 10 passes through this part of the graph, we notice that the most probable son of A is still C but the most probable son of C is now D. For C, when this happened, a change in the *nextProbable*, *last* and *last2* pointer values was performed. At that moment, C recalculated the prediction from its node. However, for the node A, this is difficult to test and handle. Recalculating the prediction at each step is not a solution since it is too costly in terms of CPU overhead.

The adopted solution is to use a general counter that acts as a version number. At each change of the most probable son, the node increments the counter and assigns a new *version* field in its structure. Before predicting something, a simple comparison between the versions of the current node and its most probable son is performed. If the versions do not match, the current node recalculates the prediction and updates its own version number.

This solution is not perfect in the sense that if we are predicting n elements in advance and there is a change of one of the most probable nodes, it will take multiple passes through the graph to update every node. However, this solution is acceptable since it is lightweight and a few iterations through the graph is minimal compared to the millions of accesses to the model.

The probableNext, last and last2 links

This section presents the use and decisions made for the three *probableNext*, *last* and *last2* pointers. The three pointers were added to the basic node structure in order to limit the number of times the list of sons has to be accessed. Though the list of sons is sorted, this would still be a long process at each step of the model and would not allow a speed-up when predicting.

The *probableNext* pointer represents the most likely son because its associated edge has the highest value. Whenever another edge has a superior label value, the model exchanges the pointers.

If the new most probable node is the *last* pointer, the model simply exchanges both counters and pointers. However, if it is the *last2* pointer, a cycle between the pointers is performed: *probableNext* becomes the *last* pointer, the *last* pointer becomes the *last2* pointer and the *last2* pointer becomes the new *probableNext*.

In the case that none of the three pointers match the current stride, the model compares the

Listing 3.4: API of the Esodyp framework

```
SMarkov* Markov_Initialize();                      //Initialize function
void Markov_Reset(SMarkov *m);                     //Reset the model
void Markov_Clear(SMarkov *m);                     //Clears the model
void Markov_Set(SMarkov *m, int v, int mask);     //Set certain attributes
void Markov_SetAddress(SMarkov *m, void*);         //Sets the address field

//Output graph to .dot format
int Markov_OutputGraphDot(SMarkov *m, const char *fname);

void Markov_DisplayInfo(SMarkov *m);               //Display information
```

Type	**Definition**
Construction Depth	The depth or order of the model
Maximum error	The value of the number of consecutive miss predictions before the model resets
Construction time	The maximum number of elements used to construct the model
Prefetch distance	The distance the model uses to predict in advance

Table 3.3: The different parameters a user can define

stride with the other sons of the current node. If a node is found, the *last* pointer becomes the *last2* pointer and the son becomes the new *last* pointer.

The last scenario occurs when none of the above solutions has succeeded. In this case, the model must find a new current node that is not one of the possible sons. The different root nodes are checked and, if one of the root nodes corresponds, it becomes the current node. However, before losing the pointer to the current node, a link is added between the current node and the root node.

We have stated that no new edges and no new nodes would be created during the prediction phase because it would add too much overhead. Though this is true, this new link type is a solution to limit the number of miss predictions. Since this new link is added via the three pointers and not added as a real son of the node, the model does not really add an edge but simply limits the number of times a check would have to be made through the sons of a node.

Though not perfect, this solution gives the possibility of slightly adapting the model in the case where the sequence in the construction phase was not a perfect representation of the sequence being processed during the prediction phase.

3.3.4 The API

The previous sections have presented examples of Markovian models, the construction and prediction phases through example and algorithms. This section presents the API that is used to perform and use the Markovian model. We have called the system *Esodyp* [9] for *Entirely SOftware and DYnamic Prefetching system*. Most of the code of the model is internalized, leaving the user with a few simple functions to set up, use and delete the models.

Listing 3.4 shows the different functions that a user can call. Here is a brief summary of their purpose :

- Markov_Initialize: this allocates the memory block necessary for the model and sets the default values;

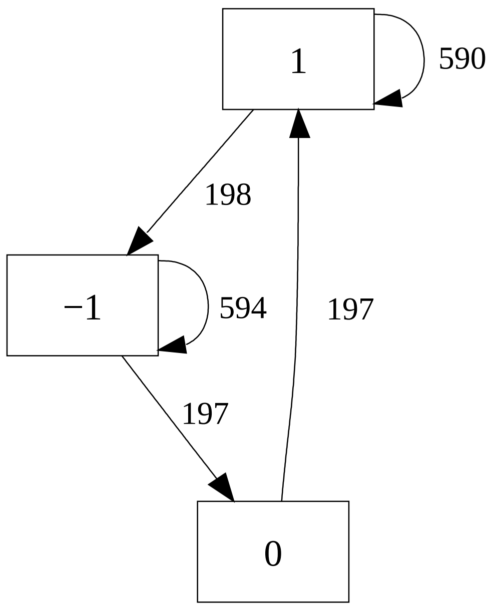


Figure 3.18: An example of a graph produced by the dot program. The dot file is created by Esodyp

0	16	20	8	12	24	**80**	16	20	8	12	24	**44**	16	20	8	12	24	**180**	16	20	8	12	24

Figure 3.19: A complex sequence

- Markov_Reset: this function resets the graph and sets the model back into the construction phase;
- Markov_Clear: this function destroys the graph and frees the memory used by the model;
- Markov_Set: this function is used to define the values of the different parameters of the model. The different parameters that a user can define are explained in the table 3.3;
- Markov_SetAddress: this function modifies the *address* field in the structure of the model, a detailed explanation of the purpose of this function is given in section 3.3.5;
- Markov_DisplayInfo: this function displays information regarding the model, it gives, for example, the number of nodes created, the number of resets;
- Markov_OutputGraphDot: this function creates a .dot file that represents the current graph. Figure 3.18 shows an example of a graph that the model can output.

3.3.5 The *Markov_SetAddress* function

The purpose of the *Markov_SetAddress* function must be explained in detail. This function does not add accuracy or directly optimize the model. However, it does limit the number of nodes in certain cases. To illustrate its use, figure 3.19 shows a trace of strides that can be modelized by Esodyp. As we can see, every 6 elements, the pattern is repeated except for the first access. This can happen when the program periodically performs the same behavior but on different

0 16 20 8 12 24 **0** 16 20 8 12 24 **0** 16 20 8 12 24 **0** 16 20 8 12 24

Figure 3.20: The simplified sequence

Listing 3.5: Initialization of Esodyp

```
SMarkov *m1;
...
m1 = Markov_initialize();

Markov_Set(m1,&j,M_CONSTTL);
Markov_Set(m1,&k,M_DEPTH);
Markov_Set(m1,&m,M_ERRMAX);
...
```

sets of data. Such a trace can have an arbitrarily big number of different strides between two sequences. This means the model will contain many nodes that are very rarely used.

In order to limit this, we can ask the model to ignore this address by calling the function named *Markov_SetAddress* prior to the first element of the pattern. This way, when the first element is processed, a stride of 0 will be calculated and this will be the case for each first pattern element of the trace. Figure 3.20 shows the simplified trace that is obtained. When modelizing this trace, the nodes that represented the different strides between the last element of the pattern and the first element of the next sequence are set to 0, which limits the number of nodes in the graph. For example, using a Markovian model of order 4 creates a graph with 65 nodes for the first sequence and 23 for the second.

3.4 Examples

This section presents two examples of how Esodyp is used and how the system reacts to different benchmark programs. The two programs that are presented are the *treeadd* program from the Olden benchmark [89] and the *181.mcf* program from the SPEC2000 benchmark [101].

3.4.1 treeadd

The *treeadd* program creates and traverses a binary tree. We modified the code in order to traverse the tree multiple times. With this change, 90% of the program is concentrated in the *TreeAdd* function. Our model was of course deployed in this function.

Initialization

Listing 3.5 presents the initialization of the Esodyp model. The initialize function allocates and sets different default values in the model that we can modify with the *Markov_Set* function. There is only one data stream that will be optimized in this program, therefore there will be only one model.

Call to the model

Listing 3.6 shows the *TreeAdd* function. First the function recursively calls itself passing the address of the left son, then itself again passing the address the right son. At the end, it returns

Listing 3.6: Call to Esodyp in the TreeAdd function

```
SMarkov *m1;

...

int TreeAdd (t)
     register tree_t    *t;
{
    ...
    m1->fct(m1,t);

    tleft = t->left;
    leftval = TreeAdd(tleft);
    tright = t->right;
    rightval = TreeAdd(tright);
    value = t->val;
    return leftval + rightval + value;
}
```

the sum of the values in the current node, the left son and the right son. To optimize this function, we need a predictor to prefetch the different nodes of the binary tree. This can be achieved by inserting the call $m1 \rightarrow fct(m1, t)$.

By doing so, at each call to *TreeAdd*, the model is informed of what tree node has been accessed. Internally, our system constructs the model, prefetches and checks the prediction accuracy.

Once the model is set and the different calls are placed, Esodyp is able to create a model and predict dynamically. However, this means being able to know where to insert the model and what parameters to use. This can be achieved by a profiler and an off-line training system or a dynamic framework that can locate hot traces in the code.

3.4.2 mcf

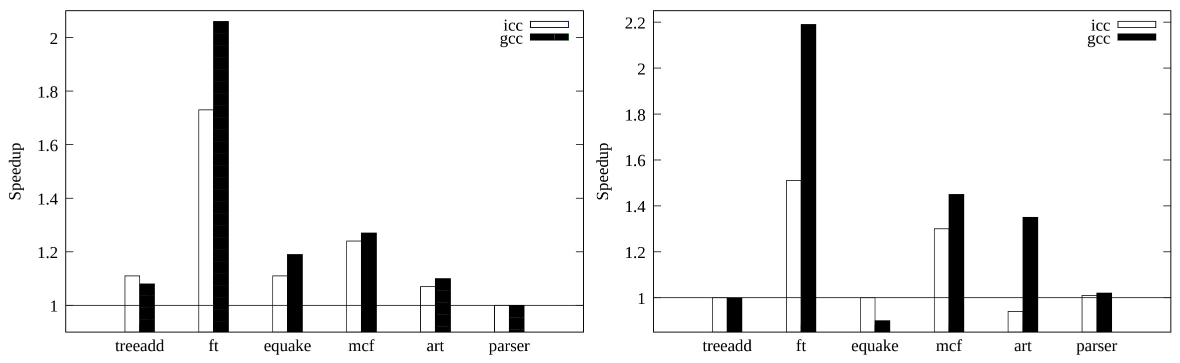
The *181.mcf* program is more complex than the previous program. However, there is one region of code containing memory accesses that is used extensively and optimizing those loads is sufficient to achieve a significant speed-up.

Listing 3.7 shows the instrumentation performed on the *181.mcf* program. Two separate data streams are optimized by the Esodyp models. Using two different structures, the streams are handled independently. This is useful when the programmer, compiler or dynamic framework knows that different data structures are being used. Though providing the full stream to the Markovian model is a possibility, it generally complicates the model and more miss predictions occur.

The instrumentation starts by calling the *Markov_SetAddress* function to set the current address correctly. Once that is done, the while loop is executed and, at each iteration the model $m1$ is called and the $m2$ model is called but at different points in the code. Though, in this case, because of the keyword `continue`, at each iteration the $m2$ model is only called once. This shows that, in some cases, it can be beneficial to call the same models at different areas of the code.

Listing 3.7: Call to Esodyp in the 181.mcf program

```
SMarkov *m1, *m2;

...

if(arcin)
        {
        Markov_SetAddress(m1, arcin->tail);
        Markov_SetAddress(m2, arcin->tail->mark);
        }

while( arcin )
        {
            tail = arcin->tail;

            m1->fct(m1, arcin->tail);

            if( tail->time + arcin->org_cost > latest )
            {

                m2->fct(m2, tail->mark);

                arcin = (arc_t *)tail->mark;
                continue;
            }

            ...

            m2->fct(m2, tail->mark);

            arcin = (arc_t *)tail->mark;
       }
...
```



(a) On the *Itanium* (b) On the *Athlon*

Figure 3.21: Speedup results of the benchmark programs

3.5 Performance and evaluation

The previous section presented two examples using the Esodyp framework to optimize two programs. This section presents the performance and evaluation of the system on different architectures, different compilers and different programs. Our first test machine is a 2-CPU 1.3Ghz Itanium-2 rx2600 workstation. However, notice that our system is entirely run onto one unique processor. The second processor is an AMD Athlon 64 X2 Dual Core Processor 4800+. We used both compilers `gcc` version 3.4 and `intel icc` version 8.1 with `O3` to compile the benchmark programs. Notice that the `icc` compiler generates prefetch instructions with `O3` while the `gcc` compiler does not. Finally, every result is the average of 5 distinct runs.

Figure 3.21(a) shows the acceleration of the programs depending on the compiler used. As we can see, using a different compiler has a different impact on the speedup obtained. In the case of `mcf`, it has been accelerated by a ratio of 1.24 using the *icc* compiler and 1.27 using the *gcc* compiler.

3.5.1 197.parser

In this subsection, we detail the behavior of the model when trying to optimize the `parser` program. Table 3.4 shows the impact that the depth of the model has on the speedup and overall performance of the model. We present this program here, since it shows that a simple depth 1 is not enough to correctly model certain memory behaviors.

Eventhough the model cannot achieve a significant speedup when working on the `parser` program, it is capable of compensating its overhead. As we can see, 3 is the best depth we can choose from. Depth 1 gives too many prediction mistakes for the model to be efficient. This can be noticed by the number of resets that have been made using a depth of 1. By using a depth 3, we reduce the number of resets from 1443 to 68 and the number of total nodes created (this count is based on the number of nodes created throughout the execution of the program, not the number of nodes created in one construction phase). If we push the depth to 4, these numbers are not reduced anymore (the execution time is even a little greater).

What is interesting to note, is that even if the model has almost the same prediction rate in the different depths presented, the number of resets is reduced significantly. After looking more in detail, it was shown that this had to do with the number of consecutive errors. With a higher depth, the model was able to reduce this number and keep it under the threshold that would

Information	Original Code	Depth 1	Depth 3	Depth 4
Correct predictions (%)	N/A	81.7	82.22	84.84
Execution time (in sec)	215.5	235.4	212.8	214.4
Number of nodes	N/A	8190	1808	2972
Number of resets	N/A	1443	68	68
Speedup	1	0.9	1.01	1.00

Table 3.4: Impact on the depth used for the Markovian model on the **197.parser** program (for the Athlon processor).

Program	File involved	Function involved	Input
treeadd	node.c	Treeadd	20 nodes and 1 processor
ft	graph.c	PickVertex	The reference input
mcf	implicit.c	price_out_impl	The reference input
equake	quake.c	smvp	The reference input
art	scanner.c	match	The reference input
parser	xalloc.c	xfree	The reference input

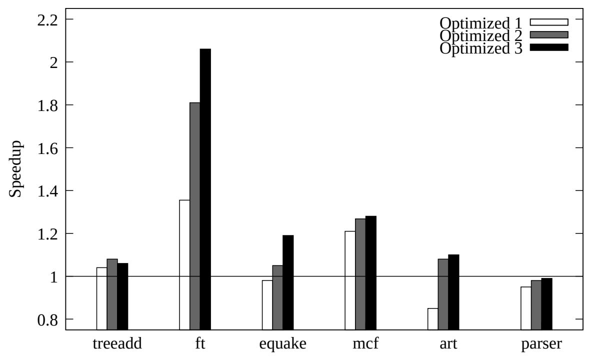
Table 3.5: The functions yielding a lot of cache misses

have provoked a flush.

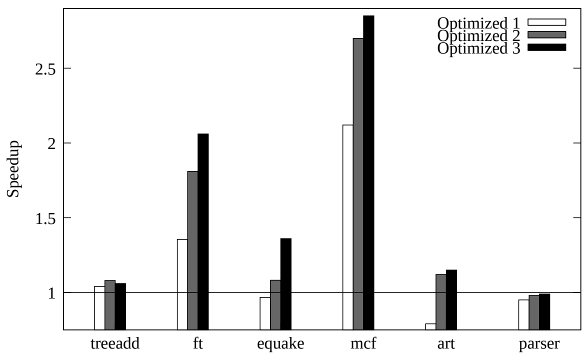
3.5.2 Other benchmark programs

We present four other benchmark programs: `treeadd` from the *Olden* benchmarks [89], `equake` and `art` from the *Spec2000* benchmarks [101] and `ft` from the *Pointer Intensive* benchmarks [90] for which one of the most costly functions of each program has been optimized (See table 3.5).

The speedups achieved on these different programs can be seen in figure 3.21. Notice that the given measures are resulting from the whole execution time of each program and not uniquely from the time of the optimized function. We were interested in optimizing the most time



(a) Program Optimization



(b) Function Optimization

Figure 3.22: Speedup results of the benchmark programs

Program	Distances
treeadd	4/6/9
ft	2/4/7
equake	3/5/10
mcf	2/4/9
art	1/4/9
parser	1/4/10

Table 3.6: Different prefetch distances used for the histogram in figure 3.22

consuming function and we looked for the delinquent load that heavily stalled the program. Only `mcf` was optimized in two different places since the loads were in the same while loop. We also present in this figure a comparison between the Intel compiler *icc* and the GNU compiler *gcc*. As we can see, the difference between both compilers is noticeable, especially for the *ft* program where we have a difference of more than 2 points.

Figure 3.22 and table 3.6 show the impact of different prefetch distance that the model can use. The direct impact is the resulting execution time. They were obtained with different parameters shown in the table. For example, if we look at the `treeadd` histograms, from left to right, they represent a prefetch distance of 4, 6 and 9.

A comparison between figures 3.22(a) and 3.22(b) can be made by stating that the programs *ft* and *treeadd* do not have a significant change between the program speedup and the function speedup since the functions that are optimized use 99% of the execution time. However, *mcf* has a major change since the function optimized only uses 31% of the original time. Therefore, when we look at the function speedup, we notice a significant boost (from 1.27 to 2.85). Program *equake* also gains a few points when we look at the function speedups.

3.6 Conclusion

In this chapter we present a Markovian model that was implemented as a totally dynamic and software solution. We presented the theory behind Markov chains, showing that Markov chains of order M offer an interesting accuracy when used to predict data streams. However, the higher the order, the bigger the graph and the overhead incurred. A compromise must be found if a speedup is to be achieved.

The implemented representation of the model is also an important element. Two solutions were presented and a graph representation was chosen since it offered the minimal overhead once the model was built. Using different optimization techniques, the calculation of the prediction can be made in a O(1) complexity. Only a few tests are needed in order to make the prediction.

Also, an optimizing framework called *Esodyp* was implemented. This gives users the possibility to easily use and integrate the system to their programs. Using a few functions, the user is not only able to set up, profile and optimize programs but also understand them by generating the graph representation in the form of a *dot* file that can be converted to most image formats. Two examples were presented in order to fully understand the system.

Finally, a performance and evaluation section presented the behavior and capabilities of the model across different benchmark programs. The performance section compared the *icc* compiler that inserts prefetches and the *gcc* compiler that does not. For the data streams we modelized and optimized, both compilers were unable to fully optimize and eliminate the delinquent loads. We showed that Esodyp can significantly reduce the delinquent loads and thus accelerate the executions. However, the programmer must insert the calls to Esodyp manually and check if the overall system reduces the execution time. Chapter 5 shows how it is possible to automatically integrate Esodyp into a dynamic optimization system.

The next chapter presents an extension to the Esodyp framework. The system was integrated into a *Distributed Shared Memory* system called *Jackal*. Instead of predicting local memory accesses, the system predicted distant memory accesses that cause costly messages. By predicting the next N elements that would need to be accessed, the system is able to limit the number of messages sent by the global system.

Chapter 4

Communication prefetching in Distributed Shared Memory Systems

The previous chapter presented a Markovian model called Esodyp. If the model is informed of every memory access of a program, the overhead would be too high compared to the acceleration provided on modern processors. However, in the case of Distributed Shared Memory (DSM) systems, this is not true since it is the number of messages sent between distant nodes that is costly. Therefore, the calculations needed to maintain the model can be considered as negligible when the model is accurate. This chapter shows that Esodyp+, an extension to Esodyp for the DSM systems, is a good candidate for such applications and, by modifying the DSM system called Jackal, Esodyp+ can handle each access check efficiently. Finally, the chapter presents a state machine like the one presented in chapter 2 to rewrite the binary code depending on the memory access behavior of each access check.

4.1 Introduction

The previous chapter presented *Esodyp*, an *Entirely SOftware and DYnamic data Prefetching system* based on a Markov model. We showed that the model is able to accurately predict future elements of the monitored sequence while continuously checking its accuracy. However, trying to optimize data cache misses on modern processors demands very low cost dynamic systems in order to obtain a speed-up.

Markov models can also be used to predict other sequences such as communications between distant machines on a network. The cost of sending messages being greater that data cache stalls, the models can be more complex while still yielding a noticeable acceleration. This chapter presents the challenges and results of porting the Esodyp framework to parallel programs. This work was done with the collaboration of Michael Klemm from the Computer Science Department 2 at the University of Erlangen-Nuremberg.

Today's high-performance computing landscape mainly consists of Symmetric Multi - Processors (SMPs) and clusters [1]. SMPs provide a hardware-managed global address space; clusters are assembled from nodes with private memory. Thus, programmers must explicitly communicate (e. g. with MPI's send/receive) to access remote data. A Software-based Distributed

Shared Memory system (S-DSM), like TreadMarks [58], Delphi [102], or Jackal [110], provides a shared-memory illusion on top of the distributed memory. In addition to registers, different cache levels, and the local memory of the node, the DSM system adds *the remote memory* as another level to the memory hierarchy of the nodes. Remote memory accesses are much more expensive than local accesses, as the high-latency interconnection network has to be crossed. Hence, it is desirable to direct the memory accesses of the application to the local memory as much as possible. For applications that do not offer such locality properties, performance drastically drops due to the high-latency remote accesses.

Esodyp+ extends Esodyp for use in an S-DSM system on a cluster. It is implemented in Jackal, an object-based DSM system, whose compiler prefixes each object access with an *access check* to test whether or not the object is cached on the local node. If not, the runtime system (RTS) requests the object from its home node. The home node is generally the node that created the object. However, if another node is extensively using that object, an object migration can occur. Esodyp+ extends the checks to monitor data access patterns in order to predict future accesses and prefetches the objects by bulk transfers.

Obviously, a dynamic prefetcher is superfluous for pleasingly parallel applications or if data access patterns can be analyzed and optimized statically. In contrast, network-bound applications with complex data access patterns benefit from this prefetching solution.

The related work can roughly be divided into a hardware axis and a size axis. With respect to hardware, prefetching occurs either (1) for single CPUs/SMPs or (2) in clusters, especially in S-DSMs. With respect to size, either (A) a single data item is prefetched or (B) a bulk transfer is used.

Most modern processors provide prefetch instructions for single data items (1A). There are also projects, e. g. Adore [73, 74] or the work of Chilimbi and Hirzel [22], that dynamically insert prefetches into programs. Modern hardware platforms with caches show a simple form of bulk prefetching (category 1B) since a whole cache line is prefetched upon memory accesses. On clusters, prefetching single data items (2A) is prohibitively expensive. As this chapter presents work contained in category 2B, we present this related work in more detail.

Adaptive++ [14] and JIAJIA [72] use a history of past accesses to predict. Delphi [102] predicts by means of a third order differential finite context method. In [54], an OpenMP DSM uses the inspector/executor pattern to determine future accesses. All these projects asynchronously request predicted pages one by one. For an object-based DSM system this is not viable, as prefetching small objects cannot hide high network latencies.

Stride predictors [42] often separate data streams by using the instruction address as a hash key. While this works well for stable access patterns with regular strides, it does not for the more complex access patterns of object-based DSM systems.

Whereas most prefetchers in S-DSMs request page by page, our approach bulk-transfers sets of objects. MPI programmers often use this technique to manually transfer large amounts of data by one MPI send/receive pair. Similarly, Java RMI [107] ships a deep copy of arguments to remote method invocations even if some of the shipped objects are not needed at the remote side. Jackal statically groups associated objects into larger ones [111] and, aggregating access checks, it requests a set of objects and ships it in bulk fashion [110].

Instead of object combining, TreadMarks dynamically builds page groups that can be considered as larger prefetching units [59]. A predictor continuously monitors page faults and decides which pages to group. The predictor is also capable of ungrouping pages if too much false sharing is caused. Our approach is similar in that it requests a set of objects during a prefetching request. However, the to-be-prefetched objects are not grouped and un-grouped, but the set is

```
int foo(SomeObject o) {


    return o.field;
}
```

```
int foo(SomeObject o) {
    if (!readable(o))
      fetch(o, readable);
    return o.field;
}
```

Figure 4.1: A function before (left) and after the insertion of an access check (right).

dynamically formed at each prediction step as necessary.

The rest of this chapter is organized as follows. Section 4.2 presents the integration of the Esodyp+ into the Jackal DSM. However, to use the Esodyp+ framework necessitates that the programmer inserts calls to the predictor as it was done using the API in chapter 3. This is a significant drawback and section 4.3 presents an extension to the system that automatically instruments the access checks. At the beginning of the execution, every access check is instrumented. During the execution, if the data cannot be modelized by the Esodyp framework, the access check is de-instrumented to lower the overhead.

The work presented in the first part of this chapter was published in [64] and the second part is currently submitted for publication [13].

4.2 Esodyp+: Integrating Esodyp in a Distributed Shared Memory System

4.2.1 Jackal

As an introduction, this subsection presents Jackal in detail. Jackal implements an object-based DSM system for Java on clusters. Instead of cache lines or pages, data transfer and memory consistency are implemented on the granularity of Java objects to reduce false sharing.

Jackal respects the Java Memory Model (JMM) [80] and provides a single system image to programmers. The Java Memory model distinguishes a global *main memory* which is used for communication between the different threads and the separate and private *working memory* which is considered as a thread-private cache. The data modified by each thread is only flushed to main memory when a synchronization point is encountered. The Jackal compiler and the runtime system limit the amount of synchronizations by performing code-based optimizations.

Jackal compiles to a native executable, e. g. for IA-32 or Itanium, and inserts access checks into the Java code to prepare it for the usage in the DSM environment. In order to handle memory coherency, each thread contains two bitmaps to handle the different objects. One is used for readable objects and the second is for writable objects. Before a set or a read to an object, a check is performed on the associated bitmap. If the right bit is set, the object is directly read or written via the local memory. However, if the bit is not set, the program is halted until the runtime system requests and receives the data.

From the programmer's perspective, a Jackal program is a simple multi-threaded program that can be executed on a single computer. However, since Jackal can send the threads to distant nodes, different access checks are used to request the required distant data. Figure 4.1 shows a sample of original Java code (left) and the added check as Java-like pseudo-code (right). If the accessed object is not locally available, a message is sent to the *home-node* that stores the master copy of the object. This request is answered with a message that contains the requested data. Thus, the caused delay is roughly two times the network latency plus the cost of object serialization and deserialization. A prefetcher suffers from about the same latency for its request,

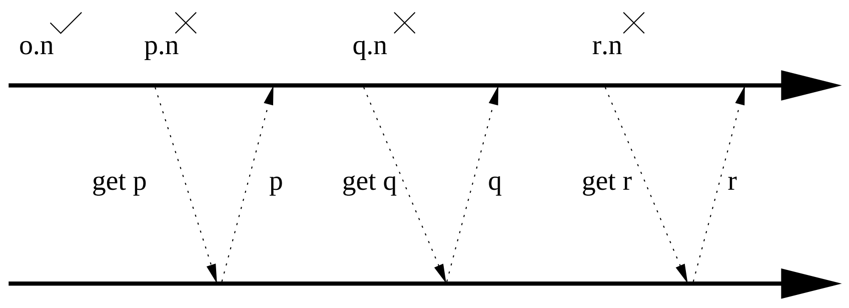


Figure 4.2: The execution of the Jackal request/send system

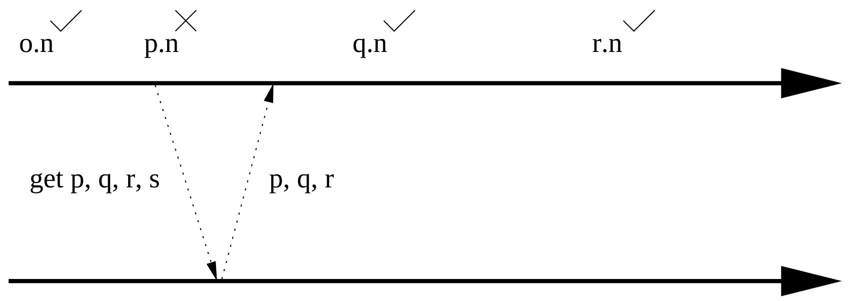


Figure 4.3: The execution of the Jackal request/send system using a prefetching mechanism

but reduces the number of blocking waits by requesting multiple objects at the same time. This bulk transfer request greatly reduces the number of messages.

Figure 4.2 shows the execution of the Jackal request/send system. When the node on the top requires an access to p, it notices that the object is not in the local cache. A message is thus sent to the home node which sends the required object. Later, when q or r are needed, the same mechanism is used. However, figure 4.3 shows the same access sequence using a predictor. When the access to p fails, let us assume that the predictor requests p but also q, r and s. In this example, the object s does not exist. Since a predictor is not always accurate, it can request data that does not exist. In this case, the home node simply ignores such requests and only sends the objects that do exist. Therefore, upon receiving the objects p, q and r, the first node accesses the objects without stalling.

To uniquely identify each object that is allocated in the DSM system, a Global Object Reference (GOR) is created. A GOR is a tuple *(alloc-node, alloc-address)*, where *alloc-node* is the logical rank of the node that created the object and *alloc-address* is the address of an object on that node. The GOR is fixed during the life time of an object; only if the object is reclaimed by the garbage collector, the GOR is released and may be recycled. When a non-local object is received, it is stored in a local caching heap and it is assigned a local address in the local address space for efficient object access. The DSM system maps these local addresses to their corresponding GORs and hence to the home-nodes (for status or data updates).

Whereas Jackal mostly transfers individual objects, arrays are handled differently for performance reasons. Since transferring an array as a whole is not an option since its size can be arbitrarily large, the RTS partitions it into *regions* of 256 bytes. Thus, a failing access check requests only one region from the concerned home-node.

Listing 4.1: A Java class that extends the Thread class

```
class SumWorker extends Thread {
        static int sum = 0;

        int start, end;

        StencilWorker(int start, int end) {
                this.start = start;
                this.end = end;
        }

        private int f(int i) {
                return 2*i +3;
        }

        public void run() {
                int local_sum = 0;

                for(int i=start; i<end; i++) {
                        local_sum += f(i);
                }

                sum += local_sum;
        }
}
```

4.2.2 A Jackal Program

A Jackal program is by definition a multi-threaded Java program. This subsection presents a simple Java example in order to explain how Esodyp is integrated into the Jackal DSM system. In Java, a multi-threaded program starts by writing a class that either extends the *Thread* class or implements the interface *Runnable*.

When extending the Thread class, different methods can be redefined but the most important is the `run` method. When a program creates a thread, it must first call the `start` function that will handle internal initializations and then calls the `run` method. If not redefined, the method simply returns and the task of the thread is finished.

Listing 4.1 shows a simple example of a class extending the Thread class. As we can see, the class contains a static field called `sum` and two private fields named `start` and `end`. The `run`

Listing 4.2: The correct implementation of the `run` method

```
        public void run() {
                int local_sum = 0;

                for(int i=start; i<end; i++) {
                        local_sum += f(i);
                }

                synchronized(sum) {
                        sum += local_sum;
                }
        }
```

Listing 4.3: A Java program that creates 5 threads

```
public class Sum {
        public static void main(String[] args) {
                int threads = 5;

                for(int th = 0; th < threads; th++) {
                        SumWorker sw = new SumWorker(th * 100, (th + 1) * 100);
                        sw.start();
                }
        }
```

method simply calculates the sum of values returned by the `f` method using the different values ranging from `start` to `end`.

The field `sum` is shared among the different instances of the `SumWorker` class and, at each update, a verification must be made to keep the program semantically correct. In order to optimize the program, a local variable called `local_sum` is used. Since the local variable can be updated without having to take special precautions, the calculation can be achieved without any special care. However, once it is finished, a single update to `sum` is required but this update must be handled correctly.

In a multi-threaded program, the update to `sum` requires a mutex to ensure that the semantics of the program is respected. Java provides mutex handling using the key word `synchronized`. Listing 4.2 shows the `run` function that is semantically correct. Since the Java Memory Model does not require the thread to write back the update to the static variable, the thread can simply keep the modified value until a synchronization point is reached. Therefore, in a Jackal program, though access checks are inserted by the compiler, the `synchronized` keyword is still necessary since the keyword forces Jackal to lock the value before each update.

Finally, to use the `SumWorker` class, a number of threads are created as it is shown in the listing 4.3. The main method starts by creating five instances of the SumWorker class and calls the method `start` for each object. The `start` method is the entry point to the thread and calls the `run` method internally.

4.2.3 Integration of a communication predictor into Jackal

Esodyp+ extends Esodyp, the model presented in chapter 3. This subsection briefly presents the model, how the addresses passed by the DSM framework are processed and how it helps to prefetch objects.

One extension to the Esodyp framework is that Esodyp+ no longer handles only simple addresses but can store any type of information in its prediction model. Hence, we can directly map the *Global Object References* (GORs) of the Jackal system to the model and get exact predictions. However in such a configuration, for each monitored GOR, at least one node in the graph is created. Since this would make the model unmanageable for large working sets, it has to be kept compact without losing prediction accuracy. Our solution to this problem is similar to the stride system used in the Esodyp model, as we compute the differences between two subsequent GORs and store only the difference in the model.

Originally, Esodyp is implemented using a *construction phase* to create the model that later is used in the *prediction phase*. Esodyp+ merges both phases and emits predictions even while it is constructing. This helps to reduce the overhead of the model by starting to predict earlier. This means that prediction strides must often be recalculated when changes are made in the

Listing 4.4: An example of false sharing

```
class IndependentSumWorker extends Thread {
        static double sumLocal[N_THREADS];

        int idthread;
        double[] tab;

        IndependentSumWorker(int id, double[] t) {
                idthread = id;
                tab = t;
        }

        void run() {
                sumLocal[idthread] = 0.0;

                for (i=0; i<tab.length; i++) {
                        sumLocal[idthread] += tab[i];
                }
        }
```

graph. Every time a change to the most probable child of a node is more recent than the last change of the current node, the prediction is recalculated. This was presented in subsection 3.3.3 page 110.

Finally, a function called `Markov_PredictN` was added to the API of the framework. This function enables bulk prefetching and receives three parameters: a pointer to the Esodyp model m, an array *tab* and an integer N. The function fills the array *tab* with the next N predictions using the graph contained in m.

Integration

A predictor for an object-based DSM system is different in several aspects compared to single-core processor predictors. Since predictors cannot guarantee accuracy, the DSM runtime must still check the validity of predicted GORs. Except for the creator of an object, nodes cannot guarantee the validity of GORs without previously having accessed the object. Hence, only the home-node of an object can check a GOR for correctness. Thus, the predicted GORs are sent to the home-nodes and an object is only sent back for valid GORs; otherwise the request is safely ignored.

Predictors often only predict the next probable address. For each address, a prefetch instruction is emitted into the instruction stream [22, 73]. Because of the high network latencies in the DSM system, prefetching of single objects does not give enough overlap of communications and computations. Hence, Esodyp was extended to emit predictions to the next N objects by calling the `Markov_PredictN` function. If N objects are bulk-transferred, the program can continue without delay until the $(N+1)^{th}$ object is needed. In addition, the message count is reduced from $2N$ to 2 by serializing together N objects into a single message. While predicting the next object can be fairly accurate, the farther away a predicted access is, the smaller is the accuracy. This causes predictions of unused objects. In addition, a growing N increases the chance of *false sharing*, which in turn increases DSM protocol activity. False-sharing occurs when two threads are accessing different elements but the DSM protocol computes useless checks.

For example, listing 4.4 shows a classic case of false sharing. Each thread uses an independent

Listing 4.5: The wrapper definition

```
class MP {
    /* Read */
    public static jackal_native void shm_markov_add_array(Object o, int elem);
    public static jackal_native void shm_markov_add_obj(Object o);

    /* Read/Write */
    public static jackal_native void shm_markov_add_array_rw(Object o, int elem);
    public static jackal_native void shm_markov_add_obj_rw(Object o);

    /* Reset addresses */
    public static jackal_native void shm_markov_resetAdd_obj(Object o,
                                                           int writefault);
    public static jackal_native void shm_markov_resetAdd_array(Object o,
                                                             int elem,
                                                             int writefault);
}
```

element of the array *sumLocal* which is used to calculate the sum of an array. Consider the case where the threads are all modifying the array *sumLocal* and that the elements are located in the same region. In this case, every time the DSM protocol requires a synchronization of one element, the other one must also be synchronized though both elements are semantically independent. However, this is only a simple example since the Jackal compiler reduces such cases of false-sharing.

To reduce the negative effects of false-sharing, Jackal detects multiple writers to the same region. In such cases, Jackal creates a shadow copy of the region before any modification takes place. When the system flushes the modifications, thus sending the information to the home node, Jackal compares the unmodified shadow copy and the modified region and only sends back the change to the home-node. Hence, the false-sharing no longer causes a message upon each access but instead defers the update message until the flush operation eventually is executed. For the different benchmark programs we have tested, our measurements have shown that, on average, $N = 10$ is a good trade-off between false sharing and message reduction for the Jackal DSM.

Since prefetching alters the sequence of memory accesses (accesses that would regularly happen later in the execution are now performed earlier when a node prefetches data), it interferes with memory consistency. As a data item might be updated while a prefetch is outstanding, depending on the memory model, a prefetch either can continue or has to be canceled. Jackal implements the JMM [80] even in the presence of a prefetcher. Simplified, each thread owns a private memory to cache its working set of data items that must be flushed when a synchronization point is reached. Hence, an update is usually not visible to other threads until all of them have reached a synchronization point as well and have flushed their caches. Prefetching blends well with such a weak memory consistency model. An active prefetch is not affected by concurrent updates that happen in the private cache of the updater. If the update hits a synchronization point, it still does not affect the prefetch, since the JMM requires a synchronization for a cache refresh as well. As the requestor waits for the prefetch request to complete, it cannot reach a synchronization point in the meantime.

Listing 4.6: The SGlobalAdd structure

```
typedef struct sGlobalAdd {
        u_int64_t low;
        u_int16_t high;
        u_int16_t region;
        void *obj;

        u_int8_t write_fault;
} SGlobalAdd;
```

Calling the predictor

As it was shown in chapter 3, the program must provide the access sequence to the Esodyp model. In the Jackal DSM, this is achieved by creating a Java wrapper around the Esodyp+ C code. Listing 4.5 shows the definition of the class. Each function with a name starting with `shm_markov_add` handles a distinct access check type.

Jackal first distinguishes read and write access checks. The DSM system then handles object access checks and array access checks differently. In the case of an array, the index is also provided in order to calculate the region of the element that is accessed. The `jackal_native` identifies the function prototype as being implemented as a optimized ABI (Application Binary interface) for the Jackal compiler. This keyword is only used within the implementation of Java wrappers to call C-functions in the Jackal RTS system. In each of these codes, the program simply transforms the reference of the accessed object into a special internal structure that is then used by the Esodyp framework.

When writing a wrapper, a structure is generally needed to go from one part of the code to the other. Listing 4.6 shows the `SGlobalAdd` structure definition that was used to handle this migration. An object in the Jackal DSM system is defined by four elements. Like many parallel paradigms, each thread of the program is assigned a unique index for identification purposes. To reach optimal results, each thread is sent to a unique distant node in order to use each processor at its maximum capacity. Therefore, for simplification reasons, we can suppose that the index also represents the node that contains the thread though this is not always the case. The `high` field represents this index and that node contains the original copy of the object. The `low` field represents the address of the object at its home node. The `region` field is only used for array objects and represents the chunk of 256 bytes that contains the first byte of the object. The `obj` field contains the address of the object in the local memory. Finally, the `write_fault` field is set to 1 if the access check is a read/write access check and is set to 0 if it is a read-only access check.

Listing 4.7 shows a simplified version of the function handling an access to an object. As we can see, the `low`, `high` and `obj` fields are simply updated using respectively local fields of the Java object. The `region` is set to 0 since this access is not an array access and the `write_fault` field is set to 0 since this is a read-only access check. The last instruction of the function calls the Esodyp+ pointer function `fct` as we have shown in chapter 3.

Finally, listing 4.8 shows a code segment from benchmark program `Blur2D`. It contains a call to the predictor. Since the object that is read and written is $M[i][j]$, it is that object that we pass to the predictor. This is done by calling the function *shm_markov_add_array_rw* and providing the array $M[i]$ and the index j.

Listing 4.7: A wrapper function

```
void shm_markov_add_obj(javaObject *o)
{
    SGlobalAdd a;

    a.low=o->alloc_home_address;
    a.high=o->alloc_home_node;
    a.region=0;
    a.obj=o;
    a.write_fault=0;

    rm->fct(rm,&a);
}
```

Listing 4.8: Code fragment calling Esodyp+ from the Blur2D benchmark program

```
            for (int i = 1; i < size - 1; i++) {
                for (int j = lb; j < ub - 1; j++) {

                    MP.shm_markov_add_array_rw(M[i],j);

                    M[i][j] = (
                            M[i-1][j-1] + M[i-1][j] + M[i-1][j+1] +
                            M[i  ][j-1] +             M[i  ][j+1] +
                            M[i+1][j-1] + M[i+1][j] + M[i+1][j+1]
                            )/8;
                }
            }
```

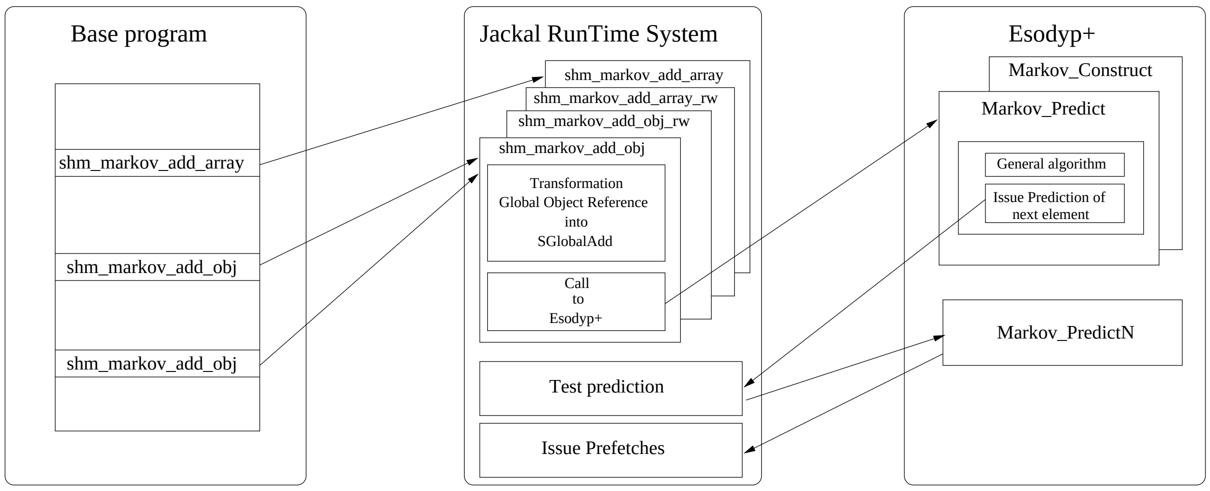


Figure 4.4: The call sequence between the base program, the Jackal RunTime System and the Esodyp+ predictor

Handling the access sequence

Once the predictor is informed of the access sequence, the model can be created and predictions can be issued in order to prefetch the data. Figure 4.4 shows the call sequence between the base program, the Jackal RunTime System (RTS) and the Esodyp+ predictor.

The main program contains calls to the wrapper of the model. The wrapper functions directly call the Jackal RunTime System depending on the type of the access check.

The Wrapper functions start by transforming the GOR representation into the `SGlobalAdd` representation as it was shown in listing 4.7. Once the `SGlobalAdd` structure is completed, the code calls the Esodyp+ predictor.

Esodyp+ uses, like Esodyp, a function pointer to determine efficiently the function code handling the current state of the model. For simplicity reasons, only two states are represented in the figure: the construction phase and the prediction phase. Let us assume the model is currently in the prediction phase. This means the $Markov\_Predict$ function is called and the general algorithm 4 presented page 109 is used.

However, there is a difference between both models. When the prediction code requests a prefetch in the basic Esodyp model, Esodyp+ calls a Jackal RTS function to test the prediction.

The prediction test checks if the predicted data is available locally. If it is, a prefetch is not useful since the data is already present. On the other hand, if it is not present locally, the Jackal RTS calls the $Markov\_PredictN$ function.

The $Markov\_PredictN$ function fills an array with the N next predictions. This is done by moving through the graph following the $probableNext$ links. However, while filling the array with predictions, if the last GOR was A and the system directly predicts A again, the $last2$ or $last$ link is followed instead in order to predict another GOR. This happens for example when the model has a node whose label is equal to zero. In such a case, the predictions would no longer be different. Instead, the system, follows a less likely link to fill the array.

The prefetches are issued but the array is first checked and the predictions are grouped by node index. Each group is then sent to the corresponding home node that checks the validity of the different requests and sends the objects that are valid.

4.2.4 Performance

To evaluate the performance of Esodyp+ in Jackal, we compare it to both a stride predictor and the Delphi predictor presented below on a Gigabit Ethernet cluster of Xeon 3.20 GHz nodes with 2 GB of memory and Linux (kernel 2.6.14.2). Since all prefetchers use the same RTS interfaces and the same maximum of $N = 10$ objects for each prefetching request, the measured differences are only caused by the overhead incurred and by the prediction quality. In all tests, each thread uses its own predictor model (since a global predictor does not make sense). Hence, the memory consumptions below are per thread and are not global.

We present four benchmark programs and discuss the effects of the prefetchers. Since a predictor is useless in an embarrassingly parallel program, the programs represent classes of applications that communicate with different access patterns. We feel that the selection is representative for applications that make use of general purpose DSM systems. The results are the averages over five runs of each program.

Predictors

Our multi-stream **stride predictor** implementation [42] with a table of 128 bytes calculates a new stride as the difference between the current GOR and the last GOR. Using a *confidence counter*, predictions are only made if the same stride occurs a certain number of times in sequence. To give the stride prefetcher a better chance to keep recurring strides, we use a *dirty counter* that enables the predictor to ignore a different stride as long as the recurring stride reappears quickly enough in the sequence of strides. The next data accesses are predicted by adding the active stride to the current address. Stride predictors can only predict very regular accesses. As a GOR not only consists of a memory address but also contains a node rank, the Stride often mispredicts in non-array programs or for complex data distributions.

Our **Delphi predictor** implementation [102] continuously updates its information and uses a constant memory cache. As it cannot detect the frequency of sequences, rare sequences cause it to forget earlier sequences and to emit mispredictions. Delphi uses a hash function to map sequences to its table. In the best case, each sequence receives a unique index. However, the number of conflicts depends on the memory access pattern. Our implementation employs a 4096-entry hash table that uses the last three accesses as the hash key. Each entry contains a pointer to a structure containing a GOR and access check information. Hence, the total size of the table is 96 KB per thread. In a certain sense, Delphi is closer to a Markov model, as it uses a fixed depth of 3. But it cannot handle depths of 1 or 2 simultaneously. For n sequences $(A, B, *)$, Delphi needs n entries, each of which stores A, B, and the last element. Esodyp+, however, handles all the depths 1, 2, and 3, and stores the subsequence (A, B) only once. A prioritized linked list then covers the n possibilities.

Table 4.1: Runtimes and message counts for the benchmarks (best in bold).

		Runtime (in seconds)				Messages (in thousands)			
	Nodes	w/o	Stride	Delphi	Esodyp+	w/o	Stride	Delphi	Esodyp+
SOR	2	24.0	**23.7**	23.7	23.9	27.6	7.1	**5.4**	6.0
	4	13.2	12.3	**12.3**	12.4	83.1	22.7	17.9	**17.6**
	6	9.4	**8.6**	8.6	8.7	139.2	36.2	**29.9**	30.0
	8	8.0	**7.2**	7.3	7.3	196.0	53.9	**40.6**	42.3
Water	2	113.8	113.7	114.5	**102.3**	1593.0	1593.0	1593.0	**962.4**
	4	78.0	78.2	78.5	**62.1**	2651.6	2651.6	2651.5	**1593.9**
	6	64.4	64.4	64.5	**52.6**	3260.4	3260.5	3260.6	**1967.9**
	8	58.7	58.7	59.1	**49.6**	3756.5	3756.7	3758.4	**2248.9**
Blur2D	2	10.6	5.9	9.3	**3.6**	223.9	223.9	134.6	**33.4**
	4	6.9	6.6	5.9	**4.0**	385.4	385.4	190.0	**99.8**
	6	7.9	10.0	6.1	**5.0**	483.4	483.4	230.6	**166.2**
	8	8.8	13.5	7.1	**6.9**	581.5	581.5	277.2	**232.3**
Ray	2	**69.2**	71.7	71.8	69.9	9.2	5.9	8.7	**5.9**
	4	35.4	36.1	36.5	**35.2**	27.3	17.4	25.8	**17.1**
	6	24.1	24.2	24.6	**23.8**	45.4	29.0	42.1	**28.4**
	8	18.4	18.5	18.9	**18.3**	57.3	42.7	59.9	**41.1**

Table 4.2: Accuracy of different prefetchers and their unused but prefetched objects.

	Accuracy (in %)			Unused objects (in %)		
	Stride	Delphi	Esodyp+	Stride	Delphi	Esodyp+
SOR	81,09 %	**97,13 %**	95,39 %	2,48 %	**0,75 %**	6,61 %
Water	8,91 %	31,81 %	**67,81 %**	79,42 %	**7,88 %**	12,83 %
Blur2D	0,00 %	49,94 %	**76,81 %**	**0,00%**	1,96 %	10,36 %
Ray	79,99 %	0,00 %	**83,12 %**	6,31 %	100,00 %	**1,93 %**
Average	42,50 %	44,72 %	**80,78 %**	22,05 %	27,65 %	**7,93 %**

Benchmark programs

Table 4.1 shows the runtime and the number of messages obtained on the various benchmark programs. We tested the programs using a 2, 4, 6 or 8 node configuration on the cluster. The column *w/o* represents the benchmark program execution without any instrumentation.

Table 4.2 presents the accuracy percentage of the different predictors and the percentage of unused prefetch objects. Accuracy is important since a non-accurate model is not able to reduce the number of messages of the system. However, the percentage of unused prefetched objects is equally important since it directly affects the number of messages that are added by the mispredictions. Both elements are important when considering a predictor since, in an extreme case, a 100% accurate model is possible if the predictor issues prefetches for every object in the program. However, the percentage of unused objects would be also approximately at 100% and the total number of messages would be very high. The ultimate predictor has an accuracy rate of 100% and never predicts an unused object.

The rest of this subsection presents the different benchmark programs and analyzes the obtained results.

SOR iteratively solves discrete Laplace equations on a 2D grid. The size of the grid is 4,100×4,100 and the number of iterations of the program is set to 50. It computes new values of a grid point as the average value of four neighboring points. Each thread of SOR receives a contiguous set of rows of the grid. SOR only communicates at the boundaries of the partitions, at which the threads read the values of the grid points of other threads.

Although restructuring compilers for array-based languages may handle such highly regular programs better, it is instructive how a prefetcher affects SOR. With a model of 2.2 KB, Esodyp+ reduces SOR's runtime by about 9 % and the message count by about 78 %. As can be seen in Table 4.1, all predictors roughly achieve the same gain; Stride is slightly faster due to its lower internal overhead. Table 4.2 shows that Delphi is most accurate closely followed by Esodyp+. Stride cannot outperform the others as the access sequence is not simple enough.

Water [115] simulates moving water molecules by means of an (N-square) N-body simulation. Work is divided by assigning molecules to different threads. After a thread has finished computing new directions and accelerations for its molecules in the current time step, it publishes these results for the other threads by means of a special class that implements both synchronization and simultaneous exchange of updates with other threads.

Water has a large working set (1,792 molecules) that is communicated after each time step, making it highly network-bound. Stride cannot correctly predict, as the objects are scattered over the DSM system. Delphi suffers from a number of conflicts in its database and from the high number of objects being accessed. In contrast, Esodyp+ builds an almost perfect model of 14.8 KB and reduces runtime by up to 20 % (for 4 nodes). The message count is decreased by about 40 %. No static analysis can achieve similar results for such irregular applications. Esodyp+ achieves 68 % accuracy (Table 4.2), which is twice as good as Delphi and about 8 times better than Stride. Stride predicts many unused objects since, once a decision is made, the program behavior has already changed.

Blur2D implements a 2D convolution filter that softens a picture of 400×400 gray-scale points (20 iterations). It uses a 2D array of double precision values that describe the pixels' gray values. The value of a pixel is computed as the average of itself and its eight neighbors. For such a stencil computation (like SOR) accesses are difficult to predict because the parallelization does not fit to the data distribution of the DSM system. While Jackal favors a *row-wise* distribution scheme, Blur2D uses a *column-wise* work distribution. Hence, false sharing and irregular access patterns make Blur2D highly network-bound.

In contrast to Stride and Delphi, Esodyp+ saves roughly 20 % runtime on 8 nodes and the number of messages drops by 60 %. The size of the model is 5.4 KB. Esodyp+ predicts more accurately (almost 77 %) due to the additional information in its model. In contrast to Delphi and Stride, when constructing the request message, Esodyp+ first tries the most probable next access. If this leads to an object that has already been requested, it uses another edge of its graph to predict a less probable but still possible access. Hence, Esodyp+ prefetches not only the most likely sequence but also less frequently ocurring data access sequences. For increasing node counts, it cannot compensate the false sharing. The simplistic model created by the Stride predictor is the reason for its poor behavior. Due to a high number of consecutive accesses to the same DSM region, Stride does not emit any good predictions. As of the confidence counter, Stride does not prefetch when a pattern change occurs. Disabling this counter causes a 50 % slow-down due to mispredictions. Delphi's hash table is not able to provide enough slots to store the complex data access sequences of Blur2D.

Ray renders a 3D scene constructed from $2,000$ randomly placed spheres. The image is stored as a 2D array (500×500) of RGB values. For parallelization, the image is partitioned into independent sub-images that are assigned to the threads. Raytracing is inherently parallel: it has a read-only working set (the spheres) and a thread-local working set that is written (the sub-images).

On average, the prefetchers do not gain any performance, although Stride and Esodyp+ (1.5 KB model) save roughly 35 % of the messages. Delphi is unable to make correct predictions due to conflicts in the hash table. This leads to an accuracy of 0 % and an unused prediction percentage of 100 %. Stride is able to keep a score of 79 % compared to the 83 % of Esodyp+. However, Stride makes more unused predictions (6,31 % vs. 1.93%). Transferring read-only data at initialization, the threads work on local data that is only transferred at the end of the computation. Obviously, for embarrassingly parallel applications prefetching does not improve performance. Variations in the cluster load cause the fluctuations in Table 4.1.

To **summarize**, let us compare the overheads of the predictors. When learning, Stride has the lowest overhead as it only considers strides relative to the last access. Delphi causes a higher overhead by computing the hash index for each access. Esodyp+ updates a few nodes and counters in the model, which also results in a higher overhead. When predicting, Delphi still calculates the hash index of the current access sequence, whereas Esodyp+ predicts by just following pointers to the most probable prefetch candidates. Hence, for stabilized models, Esodyp+ reaches the low overhead of a stride predictor. This is a major advandage of the Esodyp+ model, as it is able to modelize complex memory access behavior with a low prediction overhead once the model has been created.

4.2.5 Conclusions

In this section, we have shown that it is useful to integrate a predictor into an object-based DSM system since predictors can generally compensate their overheads. Esodyp+, a novel Markov-based prefetcher performs better than two existing prefetchers. It is more precise and more efficient in predicting and emitting prefetches and reduces the message count by about 60 %. It reduces runtime by 15 % on an 8-node computation. On average, it achieves an accuracy of 80 % compared to 45 % of the other predictors. Hence, our prefetcher is well-suited for network-bound applications with complex data access patterns.

However, the fact that the programmer must, as for the Esodyp model shown in chapter 3, manually insert calls to the model greatly reduces the possibilities of the framework. Though this section has shown that correctly inserted, the framework is able to reduce the execution time and the number of messages, an automatic solution such as the one presented in chapter 2 would be beneficial.

The rest of this chapter presents such a solution by automatically instrumenting the code and inserting calls to the Markovian model when it is considered beneficial.

4.3 Automatic Instrumentation in the Jackal DSM

4.3.1 Introduction

Automatic instrumentation is a technique that has been presented in chapter 2. The code is instrumented with monitoring code at runtime without any prior knowledge of the behavior of the program. Once instrumented, decisions must be taken in order to reduce the overhead incurred. In chapter 2, this is done by using a *Dormant* state where load instructions are de-instrumented. This solution enables the code to execute itself natively without any instrumentation.

This section presents a similar method but in the Jackal DSM system. It also shows that the same level of performance of manually placed prefetching hints can be reached by dynamically removing superfluous or unprofitable calls to Esodyp+. This is achieved by rewriting the code dynamically. However, instead of being totally dynamic, the solution adopted modifies the Jackal compiler to use modified versions of the access checks.

If the modified access check decides it is unprofitable to prefetch, the call is replaced with an almost original access check. This special *Waiting* access check adds a simple test to the original

```
 1          leaq -1308622848(%r8), %rdi
 2          shrq $5, %rdi
 3          movq %rdi,%rcx
 4          movq %fs:(0), %rdx
 5          movq thread_local_dsm_read_bitmap@TPOFF(%rdx), %rdx
 6          bt %rcx, (%rdx)
 7          jc .L28961
 8          movq %r8,%rdi
 9          call shm_start_read_object@PLT
10          movq %rax,%r8
11  .L28961:
```

Figure 4.5: Example of an access check in assembly code.

check code. When an access check is sent to the Waiting state, a *wake-up* time is set. The test compares the current time and the wake-up time and, if the first is superior to the latter, the access check is re-instrumented. However, if the prefetches seem profitable, the monitoring calls are replaced with calls to the Esodyp+ framework to enable prefetching. Finally, this new system adapts its bulk prefetch distance in order to obtain the best performance possible.

A final but important distinction between both methods is the position of call to the Esodyp+ model. In section 4.2, the Esodyp+ model was informed of every access check since the programmer inserted a call to the wrapper directly in the code. The model created represents the general memory behavior whether the accesses provoke cache misses or not. In this new version, the model is directly integrated into the access check which means that it is only called when the check fails. This section presents in greater detail the reasons and consequences of this choice.

Access Checks

It is important to know how an access check is implemented on the system level to understand how it is rewritten by our binary rewriter.

Jackal assigns two bits in the local heap for every object. The first bit is used to indicate that a particular object is readable by a thread; the second one is set for writable objects. Hence, the individual objects in the heap have to be aligned to efficiently compute the bit position in the per-thread bit list during an access check. Jackal uses a 32 byte alignment since most Java objects fit into a 32 byte boundary and fast integer shift operations can be applied to compute the final bit position.

Figure 4.5 shows EM64T assembler code that is emitted by the compiler to implement the functionality of the access check depicted in figure 4.1 page 123. It is assumed that the address of the object is already stored in register `%r8`. Line 1 computes the relative offset of the object in the local heap. The shift operation in line 2 then divides the heap offset in `%rdi` by 32 to obtain the position of the bit of the object in the read bitmap. Lines 4–5 load the address of the read bitmap from the Thread Local Storage area of the current thread. The bit test in line 6 determines whether the bit is set and sets the condition register for the following jump. If the bit is set (the object is already cached), the object reference can be used after the jump to the label in line 11. If the bit is not set, the object address is loaded into `%rdi` and passed to `shm_start_read_object` (passing the reference through `%rdi` is required by the application binary interface of the Linux systems on EM64T).

The function `shm_start_read_object` is part of Jackal's RTS and is responsible to request

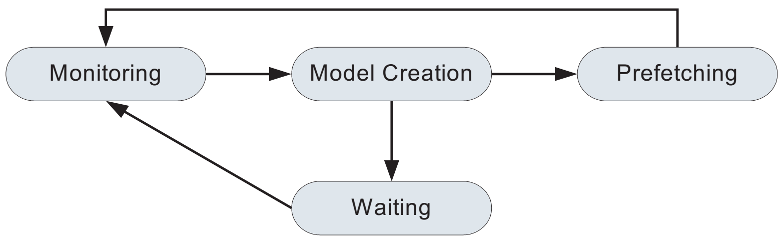


Figure 4.6: State evolution of the Esodyp+ access checks

```
        leaq -1308622848(%r8), %rdi
        shrq $5, %rdi
        movq %rdi,%rcx
        movq %fs:(0), %rdx
        movq thread_local_dsm_read_bitmap@TPOFF(%rdx), %rdx
        bt %rcx, (%rdx)
        jc .L28961
        movq %r8,%rdi
        call shm_start_read_object_monitored@PLT
        movq %rax,%r8
.L28961:
```

Figure 4.7: Example of an access check in assembly code after rewriting to monitoring state.

an object from its home node and to allocate a region for it in the cache of the local node. To request the object from the home node, the RTS uses the hash table to map the object pointer to the GOR. A message is created that contains the GOR and information about the access check. The message is then sent to the home node of the object which unpacks the request and locates the object. It is the task of the home node to maintain a list of nodes that read the object (for example, to direct the distributed garbage collector). After updating the object status, the home node serializes the object into a data message and delivers the message to the requester. Back at the requester, the data of the object is deserialized and mapped to the caching heap. Finally, the bit that corresponds to the object is set to reflect the new local state of the cache.

A write fault is processed in a similar manner. The only difference is that, upon object receipt, the requester not only maps the object into its caching heap, but also creates a shadow copy, the *twin*, of the object. When the cache is flushed, the twin is used to determine the modifications that have been made by the local node.

4.3.2 Dynamic Monitoring

While the application is executing, access checks are performed and if the data is not locally available, it is requested from its home node. In order to prefetch the data, a model is created and used to predict future accesses. However, when every access check is instrumented to provide information to the model, the predictor might receive too much information and this can hurt performance. For example, if an access check cannot be accurately modeled by our system, it is best to de-instrument that access check. To achieve this, we apply a state machine to determine which access checks the model handles.

State Machine

Figure 4.6 shows this state machine. First, the access checks are simply monitored by replacing the call to `shm_start_read_object` with a call to `shm_start_read_object_monitored` as it is shown in figure 4.7. This replacement is performed by the Jackal compiler during compilation of the source code. Using a hash table to monitor the number of times each access check is executed, frequently executed access checks are sent to the model creation state.

In this state, access checks collect the accessed GORs in a buffer. Once the buffer overflows, the system selects the most frequently executed access check, as it is likely to benefit most from prefetching the data it is requesting. The GORs of that access check are then used to construct a model for which the accuracy is calculated. If the accuracy is over a fixed threshold, the access check enters the prefetching state and prefetching starts for it. However, if the accuracy is too low, the access check enters the waiting state.

The access check remains in the waiting state until it is time to re-instrument itself. Using a *waiting level* resembling to the *dormant level* presented in chapter 2, the access check is, at first, quickly re-monitored. Then, if the model is still unable to create an accurate model, the level is incremented and the access check must wait a longer period of time before the next check. As for the *dormant level* in the system from chapter 2, the associated waiting level is incremented every time an access check enters the waiting state. This solution enables the overall system to handle each access check independently.

Finally, in the prefetching state, when access checks fail, predictions are made and prefetch messages are sent. Once a new access check is integrated into the prediction state, the Esodyp+ model automatically adapts or even reconstructs itself depending on the new memory behavior.

A state change of the access check is directly reflected in the executable code. Each state in the state machine consists in a particular entry point that is directly used by the access checks. Figure 4.7 shows the assembly code that calls the monitored entry point. When the access check is selected for prefetching, the call to `shm_start_read_object_monitored` is replaced by the call to the entry point of the prefetching code. These calls to the different versions of the access checks are directly replaced for performance reasons. Another solution is to call a common function that uses a series of `if` statements but this causes the execution of additional instructions, increases load on the memory bus, and puts a higher pressure on the branch prediction unit of the CPU. For these reasons the call instruction is directly replaced.

When compared to the state machine in chapter 2, there are certain differences. First, this system seems simpler in design. In certain aspects it is. For example, the distance learning state is removed and a general distance learning system is implemented and presented in subsection 4.3.3. Second, monitored access checks no longer go into the Waiting state since this system is called upon directly in the access checks. This means that, when the model is called, the access check did fail. Hence, the system knows that every access check in the model is one that fails.

Finally, since the predictor is only triggered during a failing access check, the created model only represents the behavior of the application for accesses into the remote memory instead of modeling the memory references of every accessed object. This means that at each time the model is called, it can predict the next remote accesses and prefetch them.

Modeling Remote Faults

In the previous version explained in section 4.2, Esodyp+ was informed of every (local and remote) access made by the program, because the model creation functions were prefixing the access checks. Although this solution creates models that correctly represent the general memory behavior of a program, the model cannot know which accesses actually need prefetching and which do not. This happens, as remote faults and accesses to locally allocated objects are interleaved. As another disadvantage, this kind of instrumentation causes additional runtime overheads as the instrumentation code is always executed, even if only local objects are accessed. Furthermore, if the model contained local references as well, it would also mix in predictions that target the local memory, which have to be filtered out and, thus, cause additional overhead when prefetching.

In the current system, the model creation code replaces the access check code of the DSM runtime. Hence, the predictor model only receives memory references that stem from faulting access checks. The system only incurs an overhead if there are failing access checks. Therefore, in the case of an application that only accesses its local memory, there will be no overhead since the prefetcher is never called. As the assembly code contains the bit tests, the monitor code is skipped if the object is already available on the local node. Hence, in the local case, the overhead is null. A second advantage is that access checks that cannot be modeled are directly de-instrumented and replaced by their regular DSM runtime counterparts. Therefore, these access checks are executed without any instrumentation and, thus, without any overhead.

However, there is a major disadvantage of such a system. Once the model represents the current memory behavior of failing access checks, the next failed access check might not be the one predicted. This happens when the model is able to predict the next access check and emits a prefetch for it. Hence, this particular access check does not fail as expected, as the object was already requested by the prefetch, and the model cannot update its information. The next time it is called, it can only notice that it apparently performed a miss prediction. This happens since the prediction it made transformed the next failing access check into a successful one. However, this is not a problem with our predictor, as at the next failing access check, the model will notice that there is a difference to the modeled behavior. In this case, the model will not emit prefetches until the memory references match the model again or the model notices that too many faults happen, which indicates a change in the behavior of the application. If the memory access behavior remains constant, our predictor is able to quickly predict correctly, minimizing this effect. Finally, if the prefetch distance is chosen high enough such that it fit the application's access behavior best, this effect becomes negligible.

4.3.3 Dynamic Adaption of the Prefetch Distance

In the previous version of Esodyp+, the prefetch distance was kept fixed to $N = 10$, as a set of experiments has shown that this value was a good trade-off between message reduction and DSM protocol overhead. While an increasing prefetch distance may reduce the overall number of messages, prediction accuracy generally drops and, thus, DSM protocol activity increases due to unused objects or false sharing. However, a static N does not give the best prefetch distance for all applications.

Hence, our new system also adjusts the prefetch distance to reach its optimal value. Whenever the local node sends out prefetches to remote nodes, it maintains a counter of how much objects have been sent back as an answer. This number is then compared against the current prefetch distance N to determine whether to increase or to decrease N. It is doubled if the number of received objects is higher than 75% of the requests. The distance is decreased by two thirds if less than 25% of the objects arrived. The current distance is kept for any other number in between. Thus the prefetch distance stabilizes if the application's memory access behavior is stable.

	Access Checks	Model Creation	Prefetching
SOR	9	3	1
Water	68	24	16
Blur2D	9	3	3
Ray	6	2	1

Table 4.3: Number of access checks that are passed to the model creation state and the number that are for prefetching using two nodes.

4.3.4 Performance

To evaluate the performance of the dynamic prefetcher with binary rewriting, we measured the performance of the four different benchmark programs prsented in 4.2.4 page 4.2.4. These benchmarks represent classes of applications that communicate with different access patterns. An embarrassingly parallel application in the benchmark suite is used to show that the prefetcher does not cause any performance penalty in such cases. We feel that the selection is representative for applications that make use of general purpose DSMs on clusters.

We evaluate the programs on a cluster of dual Xeon 3.2 GHz nodes with 2 GB of main memory running Linux (kernel 2.6.20.14). The nodes of the cluster are interconnected with Gigabit Ethernet. On each node, we only use one CPU to show network effects of our prefetcher. Table 4.3 shows the number of prefetching states compared to the overall number of access checks. Table 4.4 lists the runtimes and the messages counts for each program. It shows the unmodified programs (*w/o*), the programs with manually added prefetches (*Esodyp+*), the automatic prefetching placement (*DyCo*), and the automatic prefetching placement with automatic prefetch distance adjustment (*Dyn. N*). The shown results are the averages over 5 runs of each program.

If we compare the table 4.4 with the results obtained in table 4.1 on page 133, we notice that the values are not always identical. This is a consequence of multiple factors. First, the benchmark executions of both sections of this chapter were not done at the same time and the Jackal compiler has evolved in between. The generated code is thus different between both versions and this alone modifies the execution of each benchmark program. Second, distributed shared memory systems are rarely entirely deterministic since the current state of underlying network directly affects the number of messages that are sent and received. Finally, the interleaving of executions of each individual process makes the system non-deterministic since, for example, from one execution to another, the order of received messages can change. Results from executions where the cluster was not being extensively loaded and executions where the network is saturated differ, as the network load has an impact on the round-trip time of messages and, thus, on the interleaving of the processes by delaying messages. For example, if a distant node is sending the required data to a node but the message does not arrive on time, this can result in a failed check and a request is sent. However, if the data had arrived on time, the program could have directly continued the execution. Though this is true between two sets of results, we can suppose that during one set of executions the state does not greatly differ and the results we obtained showed this since the variance between two executions of a same set did not greatly vary. It is for all of these reason that the results for the version without instrumentation and the version with the manually added prefetches (*Esodyp+*) were re-evaluated for this table.

SOR only has a small working set that is accessed in a very regular fashion by the threads. As Table 4.3 shows, the number of access checks is very low and only one qualifies for prefetching. While the manually placed prefetching hints are able to reduce the number of messages sent by 57%, our automatic system roughly achieves 60% reduction on a 8 node execution. As SOR

		Runtime (in seconds)				Messages (in thousands)			
	Nodes	w/o	Esodyp+	DyCo	Dyn. N	w/o	Esodyp+	DyCo	Dyn. N
SOR	2	34.8	34.7	35.0	**34.5**	28.2	**11.5**	14.6	11.9
	4	18.6	**17.5**	19.2	18.0	84.1	**34.0**	44.1	34.7
	6	13.2	**12.2**	13.5	12.8	141.8	62.6	74.3	**56.7**
	8	11.0	**10.0**	11.4	10.4	199.2	85.2	104.6	**80.4**
Water	2	122.7	110.3	106.0	**93.7**	1713.5	1088.9	1114.6	**871.6**
	4	73.3	58.4	65.8	**56.3**	2941.5	1760.4	1843.3	**1401.8**
	6	68.0	55.6	58.0	**49.0**	3680.7	2183.6	2180.0	**1692.6**
	8	64.5	54.5	56.4	**45.7**	4299.6	2497.7	2612.9	**1984.3**
Blur2D	2	10.3	**3.5**	7.4	7.8	227.1	**68.3**	143.2	138.6
	4	6.9	4.0	4.3	**3.2**	394.5	182.0	159.4	**87.6**
	6	8.2	5.2	4.3	**3.8**	494.8	287.9	181.6	**142.0**
	8	10.0	7.9	5.2	**4.2**	589.2	403.6	228.7	**170.2**
Ray	2	41.9	**41.4**	41.8	42.9	9.2	**6.6**	8.5	8.4
	4	21.4	**20.9**	21.6	21.2	27.6	**19.5**	25.7	25.0
	6	14.9	15.1	15.1	**14.8**	46.2	**32.7**	43.4	42.0
	8	12.5	**12.0**	12.5	**12.0**	64.8	**46.0**	60.3	58.9

Table 4.4: Runtimes and message counts for the benchmarks (best in bold).

only has a small working-set, it does not give much rise to a prefetcher to gain performance. Although all three systems reduce the number of messages by at least 48%, no runtime reduction can be observed. Because of the high runtime compared to the average message latency, saving messages does not pay off in a runtime reduction.

Water only has 24 out of 68 access checks that are actually assessed for their access behavior and 16 are selected for prefetching. As Table 4.4 shows, the automatic system loses in message count reduction due to the initial monitoring phase. However, as Water contains a very large working set (the molecules) that is shared among the worker threads, the impact of the warm-up phase is almost completely hidden by the runtime. In addition, the new system is able to increase the prefetch distance which causes a reduction of the messages in transit by roughly 55% on 4 nodes, which corresponds to a gain of 12% over the old system. This is reflected by a runtime gain of 29%.

Blur2D uses a *column-wise* work distribution. Hence, false sharing and irregular access patterns make Blur2D highly network-bound. This is directly reflected by the poor execution time of the unmodified Blur2D program. As Table 4.4 shows, the runtime increases if the node count exceeds four nodes. Similar to SOR, Blur2D has a very small working set which makes it difficult to monitor the access checks correctly. Hence, the manual system wins for small node counts (70% compared to 39% for the automatic system). Yet, for larger node counts, our system catches up and reduces the message count by a total of 71% compared to 32%. As the data access behavior of Blur2D causes more and more false sharing, the automatic system gathers more data about failing access checks and, thus, is able to correctly predict the beneficial access checks. This results in a performance gain of rougly 58% on 8 nodes.

Ray shows that no prefetching technique can help to reduce significantly the runtime of this benchmark program. Although the working set (the spheres) is large enough to trigger state transitions (one access check is selected for prefetching) and to reduce the message counts. Compared to the manually placed prefetches, the performance of our system is worse due to the initial monitoring phase. In total, the reduction of the message count by the automatic system is not sufficient to reduce the overall runtime of the program, as saving a few thousands of messages is completely hidden by the runtime of the application.

4.3.5 Conclusions

In this chapter, we have shown that it is worthwhile to integrate a predictor into an object-based DSM system, since predictors can easily compensate their overheads. First, we showed how Esodyp could be extended into Esodyp+ and integrated into the Jackal infrastructure. The first part of this chapter showed how, using a wrapper class, the Esodyp+ API could be directly called upon by the programmer as it was done in chapter 3. Not only did the results show that speed-ups and message reduction are possible, a comparison with two other predictors showed that Esodyp+ handled the memory behavior more accurately.

The second part of the chapter showed that, in addition, it becomes possible to remove superfluous or unprofitable prefetching calls by using binary rewriting techniques. Also, the overhead of monitoring is eliminated if the monitor calls can be removed. Our measurements have shown that, on average, a performance improvement of 13% can be observed when binary rewriting is enabled to automatically place the prefetching access checks; the message count drops by 37%. The dynamic adjustment of the prefetch distance saves 19% of the runtime and decreases the number of messages by 46%. Finally, the maximum speed-up by our system for the benchmark programs is 58%. It has also been shown that the high prediction accuracy of Esodyp+ can be kept, even in presence of the dynamic rewriter and the new placement strategy for the prefetching calls.

Because of its Markov model, Esodyp+ automatically adapts to various access patterns and is only limited when a pattern is completely irregular. To cope with such irregular applications, we work on passing to Esodyp+ structural information about object associations, i. e., relations between individual objects. Hence, the predictor will no longer emit GORs that are prefetched, but instead it will predict which field of an object is used to access the next object. The DSM runtime has then to evaluate this structural information to actually determine which objects to send to the requester.

The next chapter presents a dynamic and automatic solution for the Itanium processor. As the state machines from chapters 2 and 4, the state machine presented in the next chapter is used to de-instrument and re-instrument loads efficiently, while at the same time selecting delinquent loads for the optimization scheme. Furthermore, as this chapter has shown, the Esodyp framework can be directly used to detect memory behaviors that can be optimized. The next system also includes the Esodyp framework into its selection strategy. However, a dynamic optimization system on a modern processor requires the lowest overhead possible and the next chapter shows how this is handled on the Itanium processor.

Chapter 5

Code Abstraction and Markovian memory behavior modelization for Dynamic Data Cache Prefetching

The extension Esodyp+ showed that the Markovian model could be used as a predictor but also as a filter to decide what memory access checks should be optimized and which should be de-instrumented. By using a state machine similar as the one in chapter 2, this chapter presents a solution using the Esodyp model as a filter for each load instruction of a program. In order to integrate Esodyp into the dynamic optimizer, it was necessary to modify the framework and a code abstraction level was added. This abstraction level helps to limit the memory used by the system and to insert complex algorithms into the instrumentation code. Using this technique, the general overhead of the system is reduced while actually creating a more precise system.

5.1 Introduction

In chapter 2, we presented a dynamic optimization system that dynamically modifies the executing binary in order to insert prefetch instructions into the code flow. The system uses a dynamic profiling mechanism that selects delinquent loads by calculating the current average latency of each load. Once loads are chosen, a stride and a prefetch distance are then calculated. By adding a prefetch instruction before the load, significant speed-ups are achieved on several benchmark programs.

However, the system in chapter 2 contains a few drawbacks that makes extensions complicated. First, it relies exclusively on modifying the bundle containing the load with an illegal instruction for each state change. Second, since the state changes only occur when the second thread checks the different counters, the system remains in the profiling state longer than necessary. This increases the overhead incurred by the system.

This chapter extends the previous work by allowing a state change without the need of halting the execution of the program. Furthermore, most of the state changes are handled internally, keeping to a minimum the task of the second thread. Second, since state changes are handled internally, loads are strictly instrumented as long as necessary. Third, the memory overhead of the system has been evaluated and a new internal structure has been created in order to minimize this overhead. Finally, instead of calculating the stride by storing the last stride and expecting that it will be the best one, the load memory accesses are stored and later processed by the Esodyp framework in order to calculate a Markovian model. If this model shows a strong stride-based load, a stride prefetching system is used. However, if a strong cycle is found in the model, the sum of strides in the cycle is used.

The current chapter can thus be seen as a combination of all previous chapters. We propose a dynamic optimization scheme that is totally software. Like the other systems in chapters 2 and 4, it does not use any prior knowledge of the program and calculates during the execution the best distance that should be used by the prefetcher. The framework selects loads by evaluating their latencies as it was done in chapter 2 and then modelizes their memory behavior using a Markovian model [55, 62] as Esodyp+ was used in chapter 4. Once the model is created, the system calculates the best prefetch distance and inserts a prefetch instruction into the code flow.

The overall system is handled with a state machine that is inspired by the one found in chapter 2. However notable differences will be presented in subsection 5.2.1. Since the framework is totally dynamic, the overhead of the monitoring system must be at a minimal. To achieve this, the loads are de-instrumented if they are considered non-delinquent. However, since a load can later become delinquent, the system periodically re-instruments the loads and calculates their current latency. Each load is considered independently and is instrumented/de-instrumented separately without modifying the code of the surrounding loads. This is a major difference with other trace-based systems that would need to flush the basic block containing the load in order to modify the instrumentation of the code.

Finally, the system uses an *instrumentation pool* to contain the dynamically generated code. However, the content of the instrumentation pool is different from the one presented in chapter 2. In order to implement complex algorithms, an abstraction level is used to exploit the same generic instrumentation code for each load. The generic code is written once in memory and the abstraction code spills and fills registers between the original code to the generic code. The system is then able to use complex algorithms to calculate and monitor loads without using an excessive amount of memory. This technique is similar to the framework called Dyninst [18], since it also uses a code abstraction level. However, Dyninst is not intended for dynamic optimization. For example, it processes a user code as the instrumentation code and it is possible to change the instrumentation code when the user requests it but this would require modifying the code at each state change. Our abstraction level uses a state handling system driven by a simple *state integer* that acts as an index into the instrumentation array. This directly decides what instrumentation is needed. It is an important element in our system since it allows to change the performed instrumentation without modifying the code and without having to directly interfere with the program.

The presented framework is implemented on the Itanium-2 processor and it automatically modifies and instruments binary code with an overhead of 3% on average. On a large set of benchmarks compiled at the optimization level O3 for both `gcc` and `icc` compilers, it is able to achieve speed-ups ranging from 2% to 163%.

The chapter is organized as follows. The next section presents a general overview of the system. A finite state machine is associated to each load once they are instrumented. Once loads are selected as being delinquent, they are modelized by a Markovian model in order to decide whether or not to optimize them. Section 5.3 presents the Markovian model, while section 5.4 presents the technical issues of the implementation. Section 5.5 shows the experimental results and compares performance results between the compilers `gcc` and `icc`.

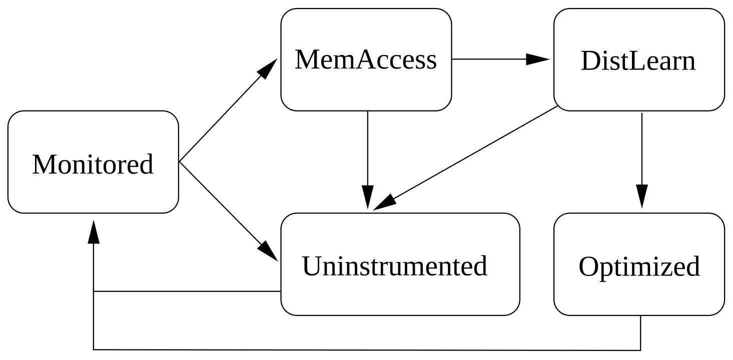


Figure 5.1: The state machine giving the possible transitions between the different states for each load

The work presented in this chapter has been submitted for publication [10].

5.2 The general framework

5.2.1 State machine

At the program startup, the system parses the code and instruments each load. The same technique was used in chapter 2 and presented extensively in section 2.4 page 66. Furthermore, each load is associated a state that determines the action to be taken when the load is to be executed. Figure 5.1 shows the finite state machine representing the transitions between the different states. First every load is instrumented and placed in the *monitored state* to determine whether it is delinquent or not.

If a load is determined as being non delinquent, a specialized optimization would not be beneficial. Therefore, instead of continuing to instrument the load, the system restores the original instruction: this lowers the overhead of the system. We define this state as being the *uninstrumented state.*

If a load is delinquent, the system tries to optimize it. The adopted optimization solution inserts a stride-based prefetching system. The first thing the framework does is study the memory behavior of that load in the *MemAccess* state. This is performed by storing a sequence of consecutive memory addresses that the load accesses. Once a sufficient number of accesses are stored, the system uses a Markovian model to analyze the behavior of that particular load. If the system can modelize the load behavior, different prefetch distances are evaluated in the *DistLearn* state. Then the best prefetch distance is used, a stride-based prefetch instruction is included into the control flow and the load is sent to the *Optimized state.*

Compared to the previous systems where code rewriting was necessary in order to handle each state transition, this new system does not. Instead, a code abstraction layer is added and each load is assigned an integer *state index.* The index gives the abstraction layer the information needed to determine what state code should be executed. Section 5.2.3 explains the code abstraction infrastructure.

The rest of this subsection presents this framework in greater detail, explaining how each state is handled and how it is possible to achieve this with the lowest overhead possible. At the same time, this subsection presents the differences between this third state machine and the two previous ones.

Monitored state The monitored state is the initial state for each load. It is the entry point of the system and it determines whether or not the system will try to optimize the load. This state simply counts the number of executions for each load and accumulates the number of cycles needed to perform the load. Each load is associated to a structure containing the number of times it is executed and the accumulator containing the sum of the number of cycles.

The instrumentation of a load starts by loading the counter, increments it and accesses the clock counter. The load is then performed and the clock counter is accessed a second time. The number of cycles needed to execute the load is obtained. By adding this value to the accumulator, the sum of all the executions is stored. Once the number of executions of the load exceeds a threshold, the accumulator is checked and if the load is considered non-delinquent, the load instruction is restored and sent to the uninstrumented state. If the load is delinquent, then the memory addresses are monitored in the memory access state.

As for the state machine integrated into the Jackal DSM system in chapter 4, this system does not require an external help to modify the state of these loads. This means that the framework does not need a second thread to check the execution counter in order to change the state of a load. In this system, once the execution counter of a load reaches the threshold, the state of the load is directly modified. Subsection 5.2.3 explains how this is performed in detail.

Uninstrumented In the uninstrumented state, loads are no longer instrumented. When the load enters this state, the system puts the bundle containing the load back in its initial state, lowering the overhead since the load is executed without any instrumentation. The system will re-instrument the load at an ulterior time in order to check whether the load becomes delinquent. This is the same as the *Dormant* state of chapter 2.

As in chapter 2, table 5.1 shows the number of latency variations of the loads for each benchmark program compiled with both `gcc` and `icc` compilers. The values in the table are the average of three runs and represent the number of noticed group changes. The four groups are defined by the average latency needed to complete the load: under 8 cycles, between 8 and 15 cycles, between 15 and 40 cycles and over 40 cycles. The second and third columns represent the total number of latency changes for each program using the `gcc` and the `icc` compilers. The fourth and fifth columns show the number of times a load changed latency groups. For example, for `parser`, there are on average 1951 group changes but only 634.33 loads changed latency groups at least once. Program `vpr`, for example, has 317.33 group latency changes for only 107.67 loads. Thus, on average, there are three latency groups changes for `vpr` per load. The last two columns represent the number of loads executed at least once before the last poll of the monitoring thread.

If we compare the table with table 2.2 on page 58, we notice that there are certain differences. This can be explained by the fact that the current system does not wait for the *monitoring thread* to wake-up in order to calculate the average latency. Since this system directly modifies the state of the load, the values obtained by both systems differ and this affects the results obtained. Of course, other internal differences between both systems lead to a change in the decision making of the overall systems. These differences directly modify the data obtained by both systems and thus the results of such instrumentations.

Memory accesses In order to analyze the memory behavior, the system stores data information in a buffer. Since multiple loads might need to be studied simultaneously, the study of the memory behavior must be done concurrently. In order to limit the memory overhead, instead of

Benchmark	Latency changes		Loads changing latencies		Executed loads	
	gcc	icc	gcc	icc	gcc	icc
ft	30	28.67	20.33	17.67	105	89.33
ks	11	6.67	8.33	4.67	189	176
anagram	10	4.33	6.33	2.67	303	119
bc	21	19	13.33	11.33	883.33	847.33
mcf	254.33	200	84.67	61.67	402	374.33
equake	35.67	23.33	13	10.67	392	753.33
art	28.33	30	15.33	18.33	318	522.67
ammp	487	385	193.67	145	1592	1124.67
bzip2	106	92.67	37.33	36.33	797	861.33
parser	1951	1282	634.33	390.33	5193	2540
gzip	53.67	56.67	24.67	28.33	768	831.33
swim	74.33	37.33	29.33	18.33	205	673.33
mesa	90.67	90.67	40	36.67	1664	1808.67
applu	229	29.67	88.33	13	1105	744
twolf	395	288.33	167	132.67	4534.67	3578
lucas	56	2.67	31	2.33	478.67	483
crafty	939	561.33	450.67	257	4446	3633.33
fma3d	474	117	156.67	72.67	3815	2853.67
facerec	9.67	29.33	5.67	16.67	79	1286.67
vpr	317.33	258.33	107.67	83	2898	2806
treeadd	73.67	1.67	23.33	1	74	17

Table 5.1: Latency variations of the load instructions for the benchmark programs

using a dedicated buffer for each load, a single buffer is used for all loads. The instrumentation stores the unique index associated to the load with the address accessed by the load. Once the buffer overflows, multiple loads can be present in the buffer. Instead of creating a model for each load, the system selects the load that was executed the most since the buffer was cleared. It then creates the Markovian model associated to that load and decides whether or not that load should be considered for optimization. If the model decides to optimize it, the load is passed to the distance learning instrumentation state. On the other hand, if the model decides not to optimize it, the instruction containing the load is restored and the load is sent to the uninstrumented state.

In the system presented in chapter 2, each delinquent load was sent to the distance learning state. It was then directly studied independently by testing various prefetch distances without knowing if the memory access behavior of the load was stride-based or not. In this system, only the delinquent loads that the Esodyp model decides to optimize are sent to the distance learning state. The decision to optimize or not depends of the accuracy of the model when considering the memory access sequence of the load. This screening process reduces the overhead of the overall system since a minimum number of loads are extensively tested.

Distance Learning Once the model has decided to optimize a load, it calculates the stride that will be used for prefetching the data. However, the final information needed to prefetch correctly is the prefetch distance. The prefetch distance is the number of elements the system

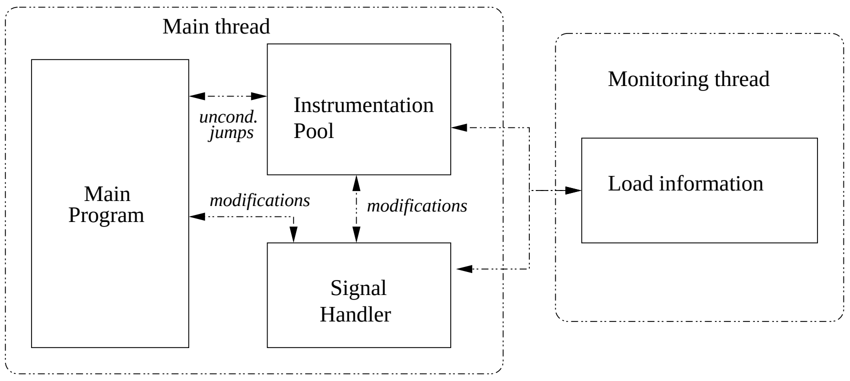


Figure 5.2: The general framework using two separate threads: on the left the original thread executing the modified program, on the right the monitoring thread

prefetches in advance. If the system uses a too small distance, there is a risk the data will not be in cache in time. On the other hand, a too high distance risks a cache eviction before the prefetched data is needed. The calculation of the prefetch distance is as follows: the memory address module passes the stride that should be used for prefetching. The system then tests a set of distances by inserting a prefetch instruction into the instrumented code and calculates the number of cycles the load takes to complete. It finally keeps the distance that yields the minimum number of cycles which is the best distance for the load at that time. In our experiments, we have found that testing every distance from 1 to 64 yields the best results while maintaining a low overhead. Finally, the load is sent to the optimized state if at least one prefetch distance is found to reduce the average latency. In the case where no prefetch distance was helpful, the load is sent to the uninstrumented state.

Whereas the system in chapter 2 only tests each power of two between 2 and 64, this system tests each distance while keeping a low overhead. Since the previous system tests each delinquent load instead of just loads that the Esodyp model can predict accurately, it has a higher overhead. This is the main reason why only the powers of two are tested. In the current system, since less loads are tested, the framework can test every distance between 1 and 64 without raising the overhead too much.

Optimized If a load enters the optimized state, the load is considered delinquent and the Markovian model can predict correctly the memory behavior. Furthermore, a stride extracted from the model for the stride-based prefetching is used and the system found a prefetch distance yielding a speed-up. The load remains in this state until the system re-evaluates the load. The decision to re-instrument the load is explained in greater detail in section 5.2.2. When this happens, the load returns to the monitoring state and the cycle starts over.

Like the system presented in chapter 4, this state machine does not contain a *Prefetched and Monitored* state that tests if the chosen stride and prefetch distances are still correct. Since the state of each load is handled internally, the system can directly recalculate the current best prefetch distance for a load without adding too much overhead. This gives the system a possibility to maintain the best distance throughout the execution.

5.2.2 The monitoring thread

In order to re-evaluate an optimization or re-instrument a uninstrumented load, a second thread periodically checks the current state of each load. Figure 5.2 shows the global framework. The main program is instrumented and jumps from the original code to the instrumented code using unconditional jumps. In the *instrumentation pool*, the code corresponding to the state of the load is executed. At the same time, the code updates the load information structure associated to that load. This lets the monitoring thread periodically check the state of each load.

However, once the original code containing a load is restored or once the code is optimized, it is unable to automatically re-instrument itself. Therefore, when the load is de-instrumented or optimized, a *wake up* field is updated in the load information structure. This wake up field is periodically checked by the monitoring thread. Once the wake up time is elapsed, the load is re-instrumented. The re-instrumentation is done by using a signal handler to halt the execution of the main thread while the code is being updated. The use of a signal handler presents two advantages: first, the code is halted and code modification can be done without any difficulties; second, function calls can easily be used to modify the code since the signal handler performs an implicit context switch.

As for the system in chapters 2 and 4, to further reduce the overhead of the system, confidence levels are added to each load. These confidence levels determine how long the loads remain without instrumentation. For example, if a load remains non-delinquent after multiple instrumentations, the system considers it will probably remain non-delinquent and wait a longer period of time before re-instrumenting it. The same is done for delinquent loads that the system is able to optimize. If the system is able to optimize a load, then it will at first remain optimized for a short period of time. Then, if the instrumentation determines that the load is still delinquent and is again able to optimize it, the load will remain optimized longer without any instrumentation. This lowers the overhead since loads that are always non-delinquent will almost never be instrumented at the end of the execution and loads that are optimized will remain so for long periods of time. Therefore, each load is associated to two confidence levels, one that is incremented when the load is de-instrumented and one that is incremented when the load is optimized.

Finally, as we can see, the monitoring thread of this system has less work to do. Instead of handling each state transition, the purpose of this monitoring thread is only to re-instrument dormant loads or to de-optimize prefetched loads.

5.2.3 Code abstraction for low memory overhead

As we have seen in the previous chapters, dynamic systems must handle at least three issues before performing dynamic code modifications. Let us summarize each issue:

- The use of a second thread in order to handle modifications of the target program code. For example, when a load has been sufficiently monitored, the system can decide to calculate an average and restore the original code to lower the overhead. The system waits until the second thread wakes up, polls the different counters and then decides to modify the instrumentation.

- To instrument and study the behavior of each load, the system creates an *instrumentation pool*. This code region contains the instrumentation code needed and the system inserts unconditional jumps between the original program and the instrumented code. However, each load must go through a certain number of states before being correctly optimized.

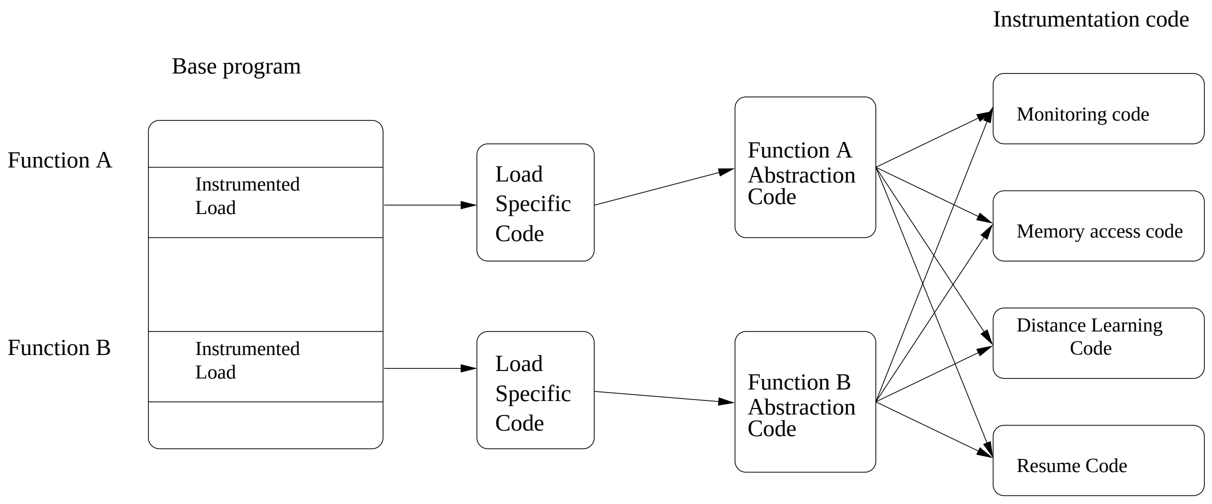


Figure 5.3: Code abstraction. From left to right, the original code enters a load specific code, jumps to the associated function code then jumps to one of the instrumentation code before returning

This means that the code executed for each load must be changed accordingly and the code region associated to the load must be at least as big as the largest instrumentation code. This is not possible when the number of loads is too important.

- The final issue is the register allocation scheme that needs to be handled correctly to allow the instrumentation code to use free registers. A practical solution is to use a trampoline between the original code and the instrumentation code. This trampoline saves internal registers, spills scratch registers and, when the instrumentation is over, restores those registers. However, a trampoline for each load uses a lot of memory. Instead, another layer of abstraction can be used to factorize the spill/fill code.

Figure 5.3 shows the general layout of the system. In order to reduce the overhead of the global system, the second thread does not handle modifications code since everything is directly handled using an associated *state integer*.

As we can see, the code is divided vertically into four parts. On the left is the original code. Each load is replaced by an unconditional jump to its specific load trampoline code. This small piece of code sets a register to the address of the associated structure and a second register to the address that will be used by the load instruction.

The code then jumps to the associated function abstraction code. Although it is possible that the system uses the same function abstraction code for multiple functions, generally each function needs a specific code to correctly spill and fill the registers needed by the final level. The function code starts by spilling and filling a few registers for the instrumentation code. The function code then also loads the current state of the load which is represented as an index to a code address table. From the table, the instruction flow is redirected to the associated state handling code.

The specific state code is the column on the right in figure 5.3. This instrumentation level contains the different instrumentation modules needed for the framework. Currently there are 4 modules: the monitoring code that detects delinquent loads, the memory access code that stores the loaded addresses, the distance learning code that tests different distances and the resume

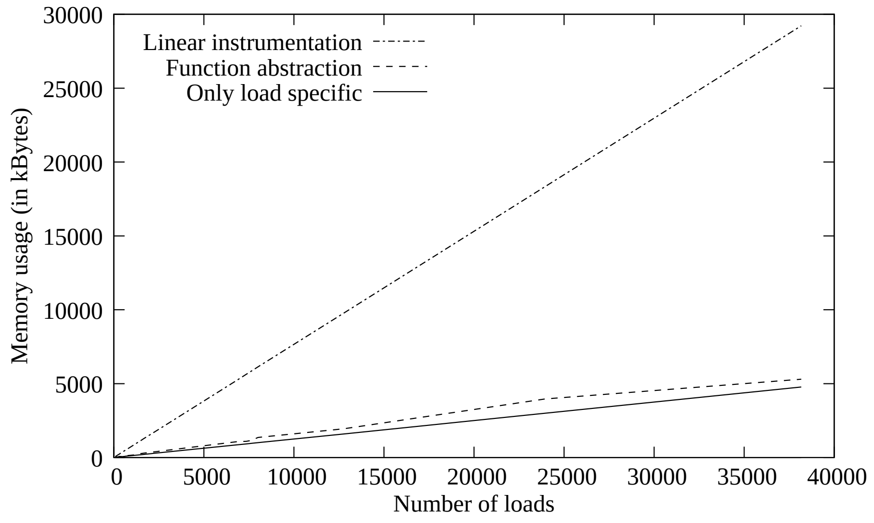


Figure 5.4: Comparing the memory overhead between using the same size of code instrumentation per load and the abstraction level system. The lowest curve is the memory used by the load specific code in the abstraction level system.

Program	Nbr of Loads	Program	Nbr of Loads	Program	Nbr of Loads
treeadd	77	art	622	vpr	6860
swim	219	lucas	666	ammp	7349
ft	311	equake	745	crafty	7633
facerec	312	applu	1335	parser	8011
ks	366	bzip2	1493	twolf	12759
anagram	451	gzip	1664	mesa	23940
mcf	480	bc	2311	fma3d	38166

Table 5.2: Number of loads instrumented per benchmark program

code that restores the original code and sends the load into the uninstrumented state. Since the function abstraction level spills and fills the same registers, this last level of code is only present once in memory. Also, when a state needs to be updated, instead of rewriting portions of code, a simple update to the state integer in the associated load structure is required.

This new infrastructure is of course very different from the version presented in chapter 2. In the previous version, the second monitoring thread had to handle each state transition whereas now the state integer is simply modified and the code abstraction layer will directly take the change into account.

This is not the only advantage. If there was no code abstraction, such a complex system would not have been possible since the memory consumption would be too great. Though the solution adds three levels of indirection, it uses the same code for multiple loads. Table 5.2 gives the number of loads instrumented by our system for each benchmark program. Figure 5.4 shows a comparison of the memory overhead used depending on the number of loads in a program and depending on three allocation schemes. The X-axis of the figure represents the number of loads in a program by sorting the benchmarks that were studied.

The most naïve technique to dynamically instrument the code would be to calculate the size of the largest instrumentation module and use that size for each code region associated to a load. This is not an interesting technique since certain instrumentation modules are relatively large and the modification of the instrumentation pool at each state change would be prohibitively expensive. Therefore, this solution is not possible for a dynamic optimizing framework. A second possibility is to merge the load specific code with the function abstraction code in order to make a single abstraction code. This would use more memory than the proposed solution and increase the risk of instruction cache pollution but would reduce the number of jumps during the instrumentation. Of course, the memory overhead would be an affine function and is represented in figure 5.4 as the top line of the figure. This affine function is the memory usage of the system presented in chapter 2.

The middle curve represents the current memory usage of our system depending on the number of loads. Since the function abstraction code is being reused for multiple functions, the curve resembles a logarithmic function. Finally, to show the amount of memory used by the function abstraction code and the instrumentation levels, the *load specific* curve represents the amount of memory needed for the load specific code. The difference between the two lowest curves gives the memory overhead used for the function abstraction code and the code for the different modules. As we can see, it is relatively low and when the number of loads grows, it can be considered negligible.

5.3 Integrating Esodyp

Once a load has been executed enough to extract an average latency and, if the system considers the load as being delinquent, the state integer is updated and the load enters the memory access state. In this state, the memory addresses that are used by the load are stored in a buffer with the index of that load. When the buffer overflows, the program is halted and the system checks the buffer. It first finds the load that is the most present in the buffer and creates a model using the Esodyp framework presented in chapter 3 of the current access behavior of that load.

Once a load has been selected, its associated memory accesses are passed to Esodyp. The model is created and, during the construction, the model remembers what nodes were visited last. When the construction is finished, the model starts predicting from the latest node by following the edge with the highest label. If the next address corresponds to the prediction then an internal counter is incremented. On the other hand, a misprediction increments another counter. Comparing both counters gives an indication on the accuracy of the model for the studied load behavior.

Once the whole buffer is handled, the system must decide whether to optimize the load or not. Since the optimization scheme uses a stride-based predictor, it is important to check if the memory behavior of the load is totally stride based or if it contains a cycle. If the load behavior is stride based then the optimizer will use that stride to prefetch the data. However, if there is a cycle in the behavior, the cycle is easily noticed in the graph. Then a correct stride must be calculated in order for each access to be correctly prefetched.

5.4 Technical issues

As for the system presented in chapter 2, certain technical aspects of this system are the same since both were implemented on the Itanium-2 processor. This section will briefly summarizes each important aspect, certain of which section 2.4 page 66 explains in greater detail.

The system was implemented as a shared library on Linux for IA64 that is automatically linked to the target application at startup. The *__libc_start_main* function is redefined to achieve this. It makes the operating system call our code before starting the target program, giving our system the possibility to handle initialization issues. First, the system parses the code by reading the ELF header and extracting the function information. The framework was implemented on the Itanium-2 processor and certain aspects of this section only apply to this architecture.

5.4.1 Register allocation

The allocation scheme for each function is studied at start-up. The need for free registers in order to implement correctly a dynamic optimization systems needs to be handled correctly as it is shown in [31]. On the Itanium-2 processor, register allocation is performed using the *alloc* instruction. There are also 32 global registers, most of which are scratch registers which can be used by any function. If a function does not have any *alloc* instruction, the system modifies the first bundle of the function by creating a patch code that adds such an *alloc* instruction. Of course, another instruction has to be added to set the state of the register file correctly at the end of the function.

However, if an *alloc* instruction exists, the system modifies the instruction to add 4 registers that will act as scratch registers for the instrumentation. Since no call instruction exists in the instrumented code, there is no risk of a register being used or read inappropriately. In the case of multiple allocation instructions in the same function, the system ignores the function. Finally, if the single allocation instruction requests every register, then the system also ignores the function since no extra register can be used.

5.4.2 Calculating the latency

The latency of a load can be calculated on most modern processors by accessing the internal clock register before and after the load. Though the Itanium processor is an in-order processor, using synchronization schemes would give the framework the possibility to monitor loads on out-of-order architectures. Since the number of executions that are monitored are negligible compared to the total number of loads executed, it would not have a high impact on performance.

5.4.3 Speculative loads and predicates

The Itanium-2 processor contains two special components that need to be addressed correctly. Speculative loads lets the program load data from an address that can possibly lead to an illegal memory location. However, these loads are handled exactly in the same way as other loads except that a speculative load is used instead of a normal load to calculate the latency. The rest of the algorithms are exactly the same.

The second particularity of the Itanium instruction set is the possibility to add a predicate in front of almost any instruction. If the predicate is set at true, the instruction is executed normally. Otherwise, it is executed but the result does not change the state of the processor. For semantics reasons, if the predicate is false, the system cannot let the load be executed. Therefore, the system adds the same predicate to the jump instruction in the load specific code. When the predicate is false, the jump from the specific load code to the function abstraction code is not performed.

5.4.4 Returning to the code

The final technical issue that needs to be resolved is how to return to the original code once the instrumentation code has been executed. Figure 5.3 shows the different levels of code the program must go through to finally execute the instrumentation code. The Itanium processor contains branch registers that contain instruction addresses that can be used as destinations for jumps. For example, these registers are used for performing function returns.

The Itanium processor contains branch registers that are used for indirect jumps. These are often used for returning from the callee function code to the caller function code. The same registers are used by our system to perform the return jumps. In the function abstraction code, before performing the jump, one of the branch registers is spilled and is used to store the return address to the function abstraction code. At the end of the instrumentation code, that register is used to return to the function abstraction code.

Once the return is performed, the function abstraction code must return to the load specific code. This is performed by adding the return address into the load information structure. Once the other registers have been restored, the code retrieves the return address, sets it to the branch register and jumps back to the load specific code. However, the branch register must still be restored before returning to the original code and this is performed once back in the load specific code. Finally, the return to the original code is done by a fixed unconditional jump since this portion of code is specific to each load.

Though the same spilling/filling mechanism could be used already at the load specific code instead of just at the function abstraction level, the choice of setting the information into the load information structure was made to reduce at a minimum the size of the load specific code.

5.5 Experiments and performance evaluation

As for the system in chapter 2, the current implementation of this system requires a load to be monitored 1000 times before an average is calculated. If the average latency is above 15 cycles, the load is considered delinquent. Once the memory access buffer is filled, the Markovian model creates a fourth order model, meaning that the last 4 accesses are used to calculate the next one. When detecting cycles in the model, the maximum length of a cycle is set to 10. Finally, the prefetch distances tested ranged from 1 to 64. Though these are arbitrary values, better settings probably exist; however, since the results are considered good, we feel that on a different architecture, correct parameters would not be difficult to obtain.

The operating system is Linux with a kernel version 2.6.9-42. The version of glibc is 2.3.4 and we used both `gcc` version 4.1 and `intel icc` version 8.1 with optimization option `O3` to compile the benchmark programs. Our system was tested on nine SPECFP2000 benchmarks, seven SPECINT2000 benchmarks with reference inputs, four Pointer Intensive benchmarks (`ft`, `bc`, `anagram` and `bc`) and one Olden benchmark (`treeadd`). Finally, every result is the average of 3 distinct runs.

5.5.1 System overhead

The system overhead was measured by removing the prefetch instruction of the instrumentation. Though this modifies the general behavior of the system, it is impossible to perfectly calculate the overhead incurred. Figures 5.5 and 5.6 shows the overhead of the system using both compilers. Finally, for every statistical information given, the information cannot reflect exactly what happens during the optimized version of the system since the instrumentation version slightly

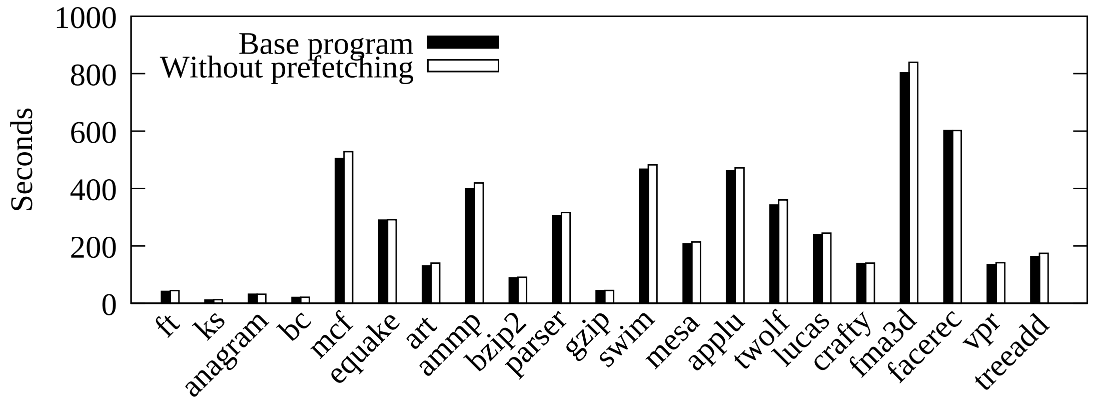


Figure 5.5: The overhead of the system using the `gcc` compiler

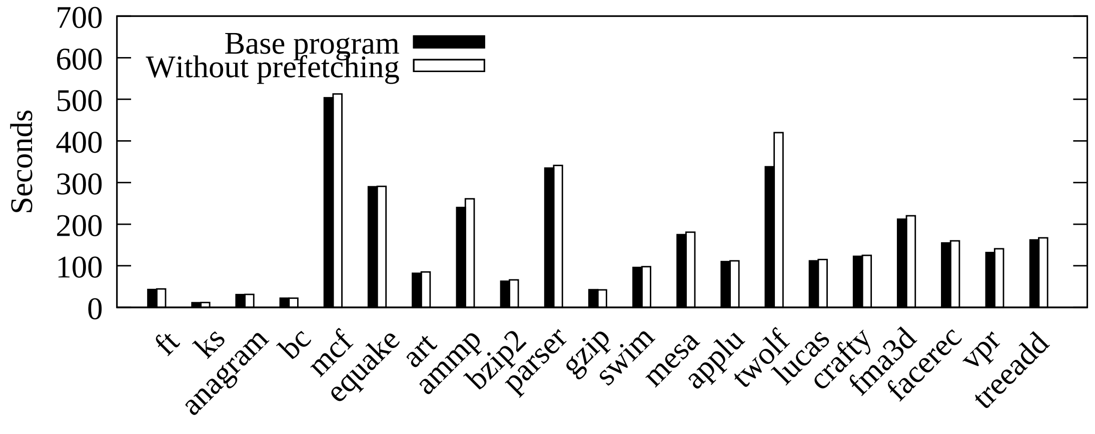


Figure 5.6: The overhead of the system using the `icc` compiler

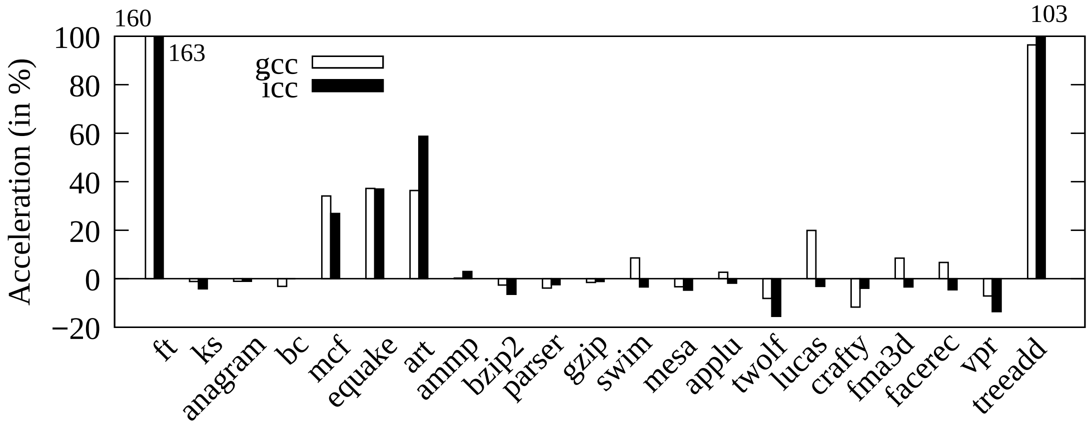


Figure 5.7: Acceleration achieved for the benchmark programs using the `gcc` and `icc` compilers

modifies the results. The average overhead is 3% of the execution time and is mostly due to the instrumentation of the code.

5.5.2 Performance analysis

Figure 5.7 shows the performance of our system on the different benchmark programs using both `gcc` and `icc` compilers. Ten benchmark programs out of the 21 tested show speed-ups ranging from 2% to 163%. The worst case scenario is the `twolf` program. However, simply modifying the register allocation system already had an important impact on its performance. Since the system is unable to find a load that can be optimized correctly, it is unable to lower that cost or even reach a speed-up.

In the cases where the system does not find a single load it can optimize correctly, all the loads enter the uninstrumented state and are slowly never instrumented again. This implicitly shuts down the general system. For programs such as `ks` or `bc`, the cost is important in percentage but these are very short programs. The initial cost of parsing the code and instrumenting it is not entirely hidden compared to the case of `gzip` for example.

When comparing the code generated by both compilers, we notice some differences. The `icc` version of `art` obtains a better speed-up than the `gcc` version. Some fortran codes such as `lucas` or `fma3d` obtain speed-ups with `gcc` but not with `icc`. Though both compilers used the optimization level O3, `icc` uses more aggressive optimizations and inserts prefetch instructions. Our system then works as a complementary prefetching system instead of a competitor.

For `mesa` and `ammp`, the results can be explained by the fact that our system does not instrument every function since the most important ones use all the processor registers.

Finally, for program `swim`, a big difference in execution time was noticed between the codes compiled with `gfortran -O3` and `ifort -O3`. With `gfortran`, the execution lasted about 11 minutes, compared to the 90 seconds of the `ifort` version. A significant speed-up was obtained using the `gfortran` compiled code as expected for such an execution time.

5.5.3 Prefetch distance

The prefetch distance selected for a load can need adjusting during the execution of a program. Table 5.3 shows, for each benchmark program, the number of times a load changes its best prefetch distance. Benchmark programs with no loads that used different prefetch distances

Program	Distance changes		Loads changing distances		Program	Distance changes		Loads changing distances	
	gcc	icc	gcc	icc		gcc	icc	gcc	icc
mcf	29.67	18.33	16	11	applu	44	0.33	17	0.33
equake	11	7	5.67	3.33	twolf	45.33	32	20.67	18.33
art	6.67	2	4	2	lucas	47	0	18.67	0
ammp	171.33	138	72.33	62.33	fma3d	188.67	4.67	76.67	3
bzip2	4.33	3	4.33	3	vpr	23.33	18.33	15	12
parser	32.67	31.67	19.67	16.67	treeadd	10	1.67	6.67	1
swim	43.67	0	20	0					

Table 5.3: Prefetch distance variations

were removed from the table. The second and third column are the number of distance changes for each program depending on the compiler. The fourth and fifth column are the number of loads that changed at least once their prefetch distance. On average, each load changed about 2 times its prefetch distance with our system.

5.6 Conclusion

In this chapter, we presented a dynamic optimization system using the Esodyp model from chapter 3 to decide whether to optimize the delinquent loads. Though this may seem to have a high overhead for memory usage and for the CPU, we have shown how to minimize both. To minimize the memory usage, we use two abstraction levels to limit the size of the instrumentation. This abstraction gives the system the possibility to use complex systems such as a Markovian model to understand the memory behavior. To lower the CPU overhead, we continuously instrument and de-instrument the code and obtain an average overhead of only 3%. This can be seen as a load-independent sampling technique since every load is considered separately compared to most trace-based sampling techniques.

Though the comparison of the overhead of this system with other works is difficult, we will still try to assess its possibilities. However, this is difficult since the underlying architectures, the compilers, and the usage or not of hardware mechanisms make each system unique. The overhead of each method can still be compared in the sense that a too high overhead will never permit an optimizer to obtain a speedup. From the highest overheads to the least, an enumeration starts with the dynamic instrumentors such as Pin [76] that have an overhead prohibitively high to dynamic optimizers such as Dynamo [5] and Deli [35] that, on average, have an overhead of 13%. For example, UMI [121] is a lightweight system using DynamoRIO [16] as a backend. The system gathers short memory reference profiles that are used to simulate cache behavior before integrating stride prefetching to obtain a speed-up. UMI has an overhead of an average of 14% compared to the 3% of our system. On the other extreme, dynamic systems using hardware components are able to maintain a very low overhead such as the 2% of Adore [73]. Our system is totally software and obtains an overhead of 3%, making it a compromise between hybrid methods and totally generic dynamic instrumentation systems.

Using the Esodyp model, the system is able to find stride-based and cycle-based loads and tests a set of prefetch distances before inserting a prefetch instruction into the instruction flow. The performance gain ranges from 2% to 163% on various benchmark programs.

Conclusion

5.7 Contributions of this thesis

This thesis has presented some original approaches to dynamic data cache optimizations.Modern dynamic frameworks are able to study, modify and optimize target programs with a low overhead but most systems are trace-based and do not target instructions directly. Furthermore, dynamic prefetchers rarely consider each load independently which generally hinders the adopted solutions.

The first part of this thesis presents, in chapter 2, a dynamic optimization system. In this chapter, we show that a simple stride-based predictor, with the right dynamic instrumentor, can achieve significant speed-ups. The system has no prior knowledge of the target programs and is initialized using a dynamic C-library entry point. Furthermore, the system uses a state machine that considers each load independently and, with the use of confidence levels, dynamically de-instruments and re-instruments loads. Each load is assigned a state in the state machine and, depending on its behavior, a transition is performed by a second thread called the *Monitoring thread*. This second thread periodically checks the states of the loads and updates them when necessary. If the load is non-delinquent and remains so throughout the execution, it is rarely instrumented. On the other hand, if the load is delinquent, the system tries to find a stride and a prefetch distance that lowers its latency. If it is found, the prefetch instruction is inserted into the code and the program is generally accelerated. The framework dynamically modifies the binary code in order to handle state transitions and has an average overhead of only 5%. On a large set of benchmark programs compiled with either `gcc -O3` or `icc -O3`, the system is able to speed up some programs by 2%-143%.

However, this first system has a few drawbacks. First, the stride and prefetch distances are calculated without any real knowledge of the memory access behavior. This trial-and-error solution is practical since it does not need any complex algorithms to decide which load to optimize. However, the fact that each delinquent load must be tested, greatly reduces the remaining execution time ressources of the system. For example, the current system can only test each power of 2 between 2^1 and 2^6 for the prefetch distances. If we add more distances, the overhead incurred increases.

In the second part of this thesis (chapter 3), we address this issue. This is achieved by creating a Markovian model named Esodyp. We present the theory behind Markovian models and, since the purpose of Esodyp is the integration into a dynamic framework, the overhead of the construction and the use of the model are taken into account. A Markovian model considers past elements in a sequence in order to calculate and predict the future elements. This is possible since it considers that the information retrieved from past elements can help determine future elements in the case of relative constant behavior in the sequence. The number of past elements used to predict is called the *depth* or the *order* of the model. We also show that a table representation is not possible with such constraints since the memory overhead and the time used to correctly maintain and predict with such a representation is too important. Therefore, Esodyp dynamically creates a graph representation of the current memory access behavior. An advantage of the graph representation is the fact that Esodyp can, without any additional costs, handle the lesser depths of the graph while maintaining the depth.

Since the construction and maintenance of the model has a high cost, the it is divided into two phases. First, the construction phase creates the graph that will be used by studying a certain number of memory accesses. Then, when the model is created, the system uses the model as a predictor. In this stage, the model is no longer updated except for the edge labels that are used to prioritize the different sons of each node. When the model is no longer accurate, it is flushed and a new model is created to represent the new memory behavior.

Finally, using precalculated predictions, priority links to determine the most probable next element in the sequence, the overhead is lowered enough to give correct predictions and even achieve speed-ups for certain benchmark programs.

The problem with the Esodyp model is that, even with the different optimization techniques used to reduce the overhead of the system to a minimum, the continuous use of the model has an impact on performance. However, depending on the integration of the model, the system can still be used. The third part of this thesis studies the application of the Esodyp model in two cases.

First, in chapter 4, the Esodyp system is integrated into a *Distributed Shared Memory* (DSM) system called Jackal. Compiling Java code into native executables, the Jackal compiler adds access checks to ensure that the data is in the local memory when required. If not, the system halts the execution and requests the data from the distant node that contains its master copy. Since, in a DSM, the cost of messages outweighs the overhead incurred by calculations, the use of Esodyp is opportune. The integration was divided into two steps.

First, a new system Esodyp+, which extends Esodyp, was implemented. Using an internal structure to handle the *Global Address References* of the Jackal system, Esodyp+ is able to modelize and predict the next accesses a program would make. By predicting the next N accesses in the form of bulk transfers, the number of messages of a given program is reduced. Finally, the Esodyp model is compared to two other known predictors. The Stride predictor is a simple stride-based predictor. The Delphi predictor uses a hashtable and the last three elements to calculate the hash-key. This predictor can also be considered as a Markovian model but only considers the fixed depth of 3. The graph representation of Esodyp gives the model the possibility to handle every inferior depth at the same time. The results showed that Esodyp generally maintained the highest accuracy while having a relatively low overhead. The Stride predictor suffered from the simplicity of the model but maintained a low overhead. Delphi, though it is able to keep a good accuracy level, suffered from a high overhead due to the continuous calculations needed for the hash-keys.

Second, an automatic binary rewriting mechanism was added to the Jackal RunTime System (RTS) in order to automatically de-instrument and re-instrument access checks. Each access check is associated to a state in a finite state machine that resembles the one presented in chapter 2. Though certain technical details from the Jackal RTS implied differences between both state machines, the underlying theory is the same. The access checks are thus handled independently and depending on their memory access behavior, they are sent to the prefetching state.

The final part of this thesis returns to the data cache optimization for the Itanium processor. The system presented in chapter 2 has its strong points but also its drawbacks. The major drawback is the selection of the loads it tried to optimize. Each delinquent load is tested by selecting a stride and testing various prefetch distances. Though this solution is able to determine if the memory behavior of a load is stride-based, it wastes time testing loads that have more complex behaviors.

Using the Esodyp framework, the system presented in chapter 5 tries to optimize this selection phase. However, a direct inclusion of the Esodyp framework into the system would increase the overhead of the system. To compensate this problem, a code abstraction level is created, allowing the overall system to handle state transitions internally without the use of the monitoring thread. Furthermore, this code abstraction level lowers the overall memory consumption of the system while allowing multiple optimizations. For example, more complex code is added into the instrumentation pool to better handle the selection of the loads. Another example is the fact that every distance ranging from 2 to 64 is tested instead of only considering certain powers of 2.

This final system obtains an overhead of 3% which is lower than the system described in chapter 2. However, using the Esodyp model, the system is more accurate and is able to determine which loads to optimize. This brings the system to a performance gain up to 163% on various benchmark programs.

5.8 Future work

The various state machines can be modified to handle different architectures or to be integrated into different systems. This was noticed when the system from chapter 2 was adapted to include Esodyp into the Jackal RunTime System (RTS). The same finite state machine could not be used since adding a monitoring thread is impractical in the Jackal RTS. Instead, the access check is sent to a *waiting* state that tests if it is time to re-monitor the check. These differences are of course dependent of the architecture of the machine and of the system in which we are trying to implement our optimization framework.

Since the systems are totally software, we are currently working on porting the system to other architectures such as the x86 architecture or embedded systems. Optimization frameworks on embedded systems must use limited amounts of memory. The code abstraction system is a good solution in order to use complex algorithms on such architectures. Though this demands a lot of implementation efforts, we are also trying to abstract the core code from the more general framework in order to have a back-end dependent of the underlying architecture, while the front-end remains identical. The system Pin [76] includes a similar separation to allow the use of common *PinTools* to be used on different architectures. The separation of the core code from the user code will probably necessitate a special form of intermediate language using a *just-in-time* compiler that will be dependent of the target architecture.

The integration of Esodyp into the Jackal RTS gave some interesting results and we are working on extending the system. For example, instead of using the *Global Object Representation* of the accesses, we are trying to find solutions to modelize the structural information of the memory accesses. For example, if we know that a sequence of code $m = o.p; q = m.n;$ occurs, it can be interesting to prefetch the objects $o.p$ and $o.p.n$. However, the object $o.p.n$ is not known before-hand, therefore a structural representation is needed in order to perform such requests. This kind of work is already complex with mono-threaded applications, it is even more so with multi-threaded applications.

Another direction this research has provided is the implementation of a dynamic optimizer in the domain of multi-core or many-core processors. Using one or more idle cores to execute complex models, it is possible to predict and optimize the code with virtually no overhead. Though this requires a certain knowledge and precautions in order to maintain the semantics of the program, it is possible that a dynamic optimizer can achieve good results where static compilers are unable to create optimal binaries. The complexity of these processors will likely bring interesting problems and be handled with equally interesting solutions.

Finally, since the Esodyp code is a single-entry C code and is integrated into the system presented in chapter 5, we intend to write a user-friendly system that can integrate any type of statistical model to help study the memory behavior of programs. This will give a chance for any user to directly plug in his mathematical model and see if it represents correctly the behavior of the program. Memory access behaviors are not the only elements our systems can handle. For example, it can easily be extended to study branches and function calls.

We are also working on writing another abstraction level of the system presented in chapter 5. This abstraction will allow users to implement their state machine using the C language. At startup, the system will load a configuration file and dynamically compile the user state machine. By using a quick register allocation system, it is possible to directly use this state machine at run-time.

These different extensions will allow users define exactly what elements of a program will be modelized, how the state machine is to be programmed and what mathematical model to use. Though frameworks such as Pin, DynInst, DTrace exist, they are generally instrumentation frameworks with relatively high overheads. It is difficult to find an optimization framework that allows a user to perfectly define what he wants to achieve and this summarizes the next step of this work.

Bibliography

[1] TOP500 List. http://www.top500.org/, 2006.

[2] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society.

[3] Hassan F. Al-Sukhni, James C. Holt, and Daniel A. Connors. Improved stride prefetching using extrinsic stream characteristics. In *ISPASS '06: Proceedings of the 2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 166–176, Austin, Texas, March 2006. IEEE Computer Society.

[4] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 168–179, Snowbird, Utah, United States, 2001. ACM Press.

[5] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, Vancouver, British Columbia, Canada, 2000. ACM Press.

[6] Thomas Ball and James R. Larus. Efficient path profiling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57, Paris, France, 1996. IEEE Computer Society.

[7] Gretta Bartels, Anna Karlin, Darrell Anderson, Jeffrey Chase, Henry Levy, and Geoffrey Voelker. Potentials and limitations of fault-based markov prefetching for virtual memory pages. In *SIGMETRICS '99: Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 206–207, Atlanta, Georgia, United States, 1999. ACM Press.

[8] Ron Begleiter, Ran El-Yaniv, and Golan Yona. On prediction using variable order markov models. *Journal of Artificial Intelligence Research (JAIR)*, 22:385–421, 2004.

[9] Jean Christophe Beyler and Philippe Clauss. ESODYP: An Entirely Software and Dynamic Data Prefetcher based on a Markov Model. In *Proc. 12th Workshop on Compilers for Parallel Computers*, pages 118–132, A Coruna, Spain, 2006.

[10] Jean Christophe Beyler and Philippe Clauss. Advanced instrumentation code abstraction for dynamic data cache prefetching. Technical report ICPS/LSIIT, Submitted for publication, August 2007.

[11] Jean Christophe Beyler and Philippe Clauss. Lightweight profiling for memory access optimizations. Technical report ICPS/LSIIT, Submitted for publication, October 2007.

[12] Jean Christophe Beyler and Philippe Clauss. Performance driven data cache prefetching in a dynamic software optimization system. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 202–209, Seattle, Washington, 2007. ACM Press.

[13] Jean Christophe Beyler, Michael Klemm, Michael Philippsen, and Philippe Clauss. Markov-based prefetching with binary code rewriting in object-based dsms. Technical report ICPS/LSIIT - University of Erlangen-Nuremberg, Submitted for publication, October 2007.

[14] Ricardo Bianchini, Raquel Pinto, and Claudio L. Amorim. Data prefetching for software dsms. In *ICS '98: Proceedings of the 12th international conference on Supercomputing*, pages 385–392, Melbourne, Australia, 1998. ACM Press.

[15] Francois Bodin, T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback Directed Compilation*, Paris, October 1998.

[16] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 265–275, San Francisco, California, 2003. IEEE Computer Society.

[17] Derek Bruening, Vladimir Kiriansky, Timothy Garnett, and Sanjeev Banerji. Thread-shared software code caches. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 28–38, Washington, DC, USA, 2006. IEEE Computer Society.

[18] Bryan Buck and Jeffrey K. Hollingsworth. An api for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[19] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *ATEC'04: Proceedings of the USENIX Annual Technical Conference 2004 on USENIX Annual Technical Conference*, pages 2–2, Boston, MA, 2004. USENIX Association.

[20] M. J. Charney and T. R. Puzak. Prefetching and memory system behavior of the spec95 benchmark suite. *IBM J. Res. Dev.*, 41(3):265–286, 1997.

[21] I-Cheng K. Chen, John T. Coffey, and Trevor N. Mudge. Analysis of branch prediction via data compression. *SIGOPS Operating Systems Review*, 30(5):128–137, 1996.

[22] Trishul M. Chilimbi and Martin 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, Berlin, Germany, 2002. ACM Press.

[23] Chang-Burm Cho and Tao Li. Complexity-based program phase analysis and classification. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 105–113, Seattle, Washington, USA, 2006. ACM Press.

[24] Seungryul Choi, Nicholas Kohout, Sumit Pamnani, Dongkeun Kim, and Donald Yeung. A general framework for prefetch scheduling in linked data structures and its application to multi-chain prefetching. *ACM Trans. Comput. Syst.*, 22(2):214–280, 2004.

[25] Philippe Clauss, Bénédicte Kenmei, and Jean Christophe Beyler. The periodic-linear model of program behavior capture. In *Euro-Par 2005*, volume 3648 of *LNCS*, pages 325–335. Springer, 2005.

[26] Jamison Collins, Suleyman Sair, Brad Calder, and Dean M. Tullsen. Pointer cache assisted prefetching. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 62–73, Istanbul, Turkey, 2002. IEEE Computer Society Press.

[27] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. Speculative precomputation: long-range prefetching of delinquent loads. *SIGARCH Comput. Archit. News*, 29(2):14–25, 2001.

[28] Thomas M. Conte, Burzin A. Patel, Kishore N. Menezes, and J. Stan Cox. Hardware-based profiling: An effective technique for profile-driven optimization. *International Journal of Parallel Programming*, 24(2):187–206, 1996.

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

[30] Intel Corporation. Vtune performance analyzer.
http://www.developper.intel.com/software/products/VTune/.

[31] Abhinav Das, Rao Fu, Antonia Zhai, and Wei-Chung Hsu. Issues and support for dynamic register allocation. In *Asia-Pacific Computer Systems Architecture Conf.*, pages 351–358, 2006.

[32] Abhinav Das, Jiwei Lu, and Wei-Chung Hsu. Region monitoring for local phase detection in dynamic optimization systems. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 124–134, Washington, DC, USA, 2006. IEEE Computer Society.

[33] Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Z. Chrysos. Profileme : Hardware support for instruction-level profiling on out-of-order processors. In *International Symposium on Microarchitecture*, pages 292–302, 1997.

[34] Mukund Deshpande and George Karypis. Selective markov models for predicting web page accesses. *ACM Trans. Inter. Tech.*, 4(2):163–184, 2004.

[35] 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.

[36] Kemal Ebcioglu, Erik Altman, Michael Gschwind, and Sumedh Sathaye. Dynamic binary translation and optimization. *IEEE Trans. Comput.*, 50(6):529–548, 2001.

[37] Jan Elder and Marc D. Hill. Dinero IV trace-driven uniprocessor cache simulator. http://www.cs.wisc.edu/ markhill/DineroIV/.

[38] Philip G. Emma, Allan Hartstein, Thomas R. Puzak, and Vijayalakshmi Srinivasan. Exploring the limits of prefetching. *IBM J. Res. Dev.*, 49(1):127–144, 2005.

[39] Suhaib A. Fahmy, Peter Y. K. Cheung, and Wayne Luk. Hardware acceleration of hidden markov model decoding for person detection. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 8–13, Washington, DC, USA, 2005. IEEE Computer Society.

[40] Shai Fine, Yoram Singer, and Naftali Tishby. The hierarchical hidden markov model: Analysis and applications. *Machine Learning*, 32(1):41–62, 1998.

[41] S. Forchhammer and J. Rissanen. Coding with partially hidden markov models. In *DCC '95: Proceedings of the Conference on Data Compression*, page 92, Washington, DC, USA, 1995. IEEE Computer Society.

[42] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. Stride directed prefetching in scalar processors. In *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, pages 102–110, Portland, Oregon, United States, 1992. IEEE Computer Society Press.

[43] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A practical method for quickly evaluating program optimizations. Proceedings of the 1st International Conference on High Performance Embedded Architectures & Compilers (HiPEAC), LNCS-3793, Springer Verlag.

[44] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.

[45] B. Harvey and G. Tyson. Graphical user interface for compiler optimizations with simple-suif. Technical Report UCR-CS-96-5, Department of Computer Science, University of California Riverside, Riverside, CA, 1996.

[46] Kim Hazelwood and Robert Cohn. A cross-architectural interface for code cache manipulation. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 17–27, Washington, DC, USA, 2006. IEEE Computer Society.

[47] Kim Hazelwood and Michael D. Smith. Code cache management schemes for dynamic optimizers. In *Sixth Annual Workshop on Interaction between Compilers and Computer Architectures*, pages 102–110, Boston, MA, February 2002.

[48] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, and et al. The microarchitecture of the pentium 4 processor, February 2001.

[49] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4), December 2001.

[50] Mark A. Holliday. A program behavior model and its evaluation. In *MASCOTS '95: Proceedings of the 3rd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 232–236, Washington, DC, USA, 1995. IEEE Computer Society.

[51] Ibrahim Hur and Calvin Lin. Memory prefetching using adaptive stream detection. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 397–408, Orlando, Florida, December 2006. IEEE Computer Society.

[52] Sorin Iacobovici, Lawrence Spracklen, Sudarshan Kadambi, Yuan Chou, and Santosh G. Abraham. Effective stream-based and execution-based data prefetching. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 1–11, New York, NY, USA, 2004. ACM Press.

[53] Intel(R). *Itanium(R) 2 Processor Reference Manual for Software Development and Optimization*, chapter Optimal use of lfetch, pages 71–72. Intel Corporation, May 2004.

[54] W.-C. Jeun, Y.-S. Kee, and S. Ha. Improving Performance of OpenMP for SMP Clusters through Overlapping Page Migrations. In *Proc. Intl. Workshop on OpenMP*, Reims, France, 2006.

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

[56] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 364–373, Seattle, Washington, United States, 1990. ACM Press.

[57] Spiros Kalogeropulos, Mahadevan Rajagopalan, Vikram Rao, Yonghong Song, and Partha Tirumalai. Processor aware anticipatory prefetching in loops. In *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, page 106, Washington, DC, USA, 2004. IEEE Computer Society.

[58] P. Keleher, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. Winter 1994 USENIX Conf.*, pages 115–131, San Francisco, CA, 1994.

[59] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks: distributed shared memory on standard workstations and operating systems. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, pages 10–10, San Francisco, California, 1994. USENIX Association.

[60] Ando Ki and Alan E. Knowles. Adaptive data prefetching using cache information. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 204–212, Vienna, Austria, 1997. ACM Press.

[61] D. Kim, S.S. Liao, P.H. Wang, J. del Cuvillo, X. Tian, X. Zou, D. Yeung, M. Girkar, and J.P. 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.

[62] Jinwoo Kim, Krishna V. Palem, and Weng-Fai 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.

[63] Jinwoo Kim, Rodric M. Rabbah, Krishna V. Palem, and Weng-Fai Wong. Adaptive compiler directed prefetching for epic processors. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 495–501, Las Vegas, Nevada, June 2004.

[64] Michael Klemm, Jean Christophe Beyler, Ronny T. Lampert, Michael Philippsen, and Philippe Clauss. Esodyp+: Prefetching in the jackal software dsm. In Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol, editors, *Proceedings of the Euro-Par 2007 Conference*, pages 563–573, Rennes, France, 2007. Springer.

[65] John Koroma, Wei Li, and Demetrios Kazakos. A generalized model for network survivability. In *TAPIA '03: Proceedings of the 2003 conference on Diversity in computing*, pages 47–51, Atlanta, Georgia, USA, 2003. ACM Press.

[66] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. Finding effective optimization phase sequences. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 12–23, San Diego, California, USA, 2003. ACM Press.

[67] Shih-Chang Lai and Shih-Lien Lu. Hardware-based pointer data prefetcher. In *ICCD '03: Proceedings of the 21st International Conference on Computer Design*, page 290, Washington, DC, USA, 2003. IEEE Computer Society.

[68] James R. Larus. Whole program paths. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 259–269, Atlanta, Georgia, United States, 1999. ACM Press.

[69] James R. Larus and Eric Schnarr. Eel: machine-independent executable editing. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 291–300, La Jolla, California, United States, 1995. ACM Press.

[70] Jeremy Lau, Stefan Schoenmackers, and Brad Calder. Transition phase classification and prediction. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 278–289, Washington, DC, USA, 2005. IEEE Computer Society.

[71] Steve S.W. Liao, Perry H. Wang, Hong Wang, Gerolf Hoflehner, Daniel Lavery, and John P. Shen. Post-pass binary adaptation for software-based speculative precomputation. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 117–128, Berlin, Germany, 2002. ACM Press.

[72] Haiming Liu and Weiwu Hu. A Comparison of Two Strategies of Dynamic Data Prefetching in Software DSM. In *Proceedings 15th International Parallel & Distributed Processing Symposium*, pages 62–67, San Francisco, CA, 2001.

[73] Jiwei 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*, December 2003.

[74] Jiwei Lu, Howard Chen, Pen-Chung Yew, and Wei-Chung Hsu. Design and implementation of a lightweight dynamic optimization system. *Journal Instruction-Level Parallelism*, 6, 2004.

[75] Jiwei Lu, Abhinav Das, Wei-Chung Hsu, Khoa Nguyen, and Santosh G. Abraham. Dynamic helper threaded prefetching on the sun ultrasparc cmp processor. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 93–104, Barcelona, Spain, 2005. IEEE Computer Society.

[76] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, Chicago, IL, USA, 2005. ACM Press.

[77] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 222–233, Cambridge, Massachusetts, United States, 1996. ACM Press.

[78] Chi-Keung Luk, Robert Muth, Harish Patil, Robert Cohn, and Geoff Lowney. Ispike: A post-link optimizer for the intel itanium architecture. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 15, Palo Alto, California, 2004. IEEE Computer Society.

[79] Gurmeet Singh Manku, Mukul R. Prasad, and David A. Patterson. A new voting based hardware data prefetch scheme. In *HIPC '97: Proceedings of the Fourth International Conference on High-Performance Computing*, page 100, Washington, DC, USA, 1997. IEEE Computer Society.

[80] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–391, Long Beach, California, USA, 2005. ACM Press.

[81] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technical Journal*, 6(1):4–15, February 2002.

[82] Matthew C. Merten, Andrew R. Trick, Christopher N. George, John C. Gyllenhaal, and Wen mei W. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, pages 136–147, Atlanta, Georgia, United States, 1999. IEEE Computer Society.

[83] David Mosberger and Stéphane Eranian. *IA-64 Linux Kernel Design and Implementation*, chapter 9.3: Kernel Support for Performance Monitoring. Prentice Hall, 2002.

[84] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 27, pages 62–73, New York, NY, 1992. ACM Press.

[85] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, San Diego, California, USA, 2007. ACM Press.

[86] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 40(2/3):103–116, 1997.

[87] Craig G. Nevill-Manning and Ian H. Witten. Linear-time, incremental hierarchy inference for compression. In *DCC '97: Proceedings of the Conference on Data Compression*, page 3, Washington, DC, USA, 1997. IEEE Computer Society.

[88] Steven Novack and Alexandru Nicolau. Vista: The visual interface for scheduling transformations and analysis. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 449–460, London, UK, 1994. Springer-Verlag.

[89] The Olden benchmark suite. http://www.cs.princeton.edu/~mcc/olden.html.

[90] Pointer-intensive benchmark suite. http://www.cs.wisc.edu/~austin/ptr-dist.html.

[91] Louis-Noel Pouchet, Cedric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative optimization in the polyhedral model: Part i, one-dimensional time. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 144–156, Washington, DC, USA, 2007. IEEE Computer Society.

[92] Rodric M. Rabbah, Hariharan Sandanagobalane, Mongkol Ekpanyapong, and Weng-Fai Wong. Compiler orchestrated prefetching via speculation and predication. *SIGOPS Oper. Syst. Rev.*, 38(5):189–198, 2004.

[93] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and optimization of win32/intel executables using etch. In *NT'97: Proceedings of the USENIX Windows NT Workshop on The USENIX Windows NT Workshop 1997*, pages 1–1, Seattle, Washington, 1997. USENIX Association.

[94] Dana Ron, Yoram Singer, and Naftali Tishby. The power of amnesia: learning probabilistic automata with variable memory length. *Machine Learning*, 25(2-3):117–149, 1996.

[95] Amir Roth and Gurindar S. Sohi. Effective jump-pointer prefetching for linked data structures. In *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, pages 111–121, Atlanta, Georgia, 1999. IEEE Computer Society.

[96] M. J. Serrano and Youfeng Wu. Memory performance analysis of spec2000c for the intel(r) itanium processor. In *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 184–192, Washington, DC, USA, 2001. IEEE Computer Society.

[97] Saurabh Sharma, Jesse G. Beu, and Thomas M. Conte. Spectral prefetcher: An effective mechanism for l2 cache prefetching. *ACM Trans. Archit. Code Optim.*, 2(4):423–450, 2005.

[98] Timothy Sherwood, Suleyman Sair, and Brad Calder. Predictor-directed stream buffers. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 42–53, Monterey, California, United States, 2000. ACM Press.

[99] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982.

[100] Noah Snavely, Saumya Debray, and Gregory Andrews. Predicate analysis and if-conversion in an itanium link-time optimizer. In *Predicate Analysis and If-Conversion in an Itanium Link-Time Optimizer*, Istanbul, Turkey, November 2002.

[101] SPEC CPU2000. http://www.spec.org/cpu2000/.

[102] E. Speight and M. Burtscher. Delphi: Prediction-Based Page Prefetching to Improve the Performance of Shared Virtual Memory Systems. In *Proc. Intl. Conf. on PDPTA*, pages 49–55, Las Vegas, NV, 2002.

[103] Santhosh Srinath, Onur Mutluand Hyesoon Kim, , and Yale N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proc. of the 13th Int. Symp. on High-Performance Computer Architecture (HPCA)*, February 2007.

[104] Amitabh Srivastava, Andrew Edwards, and Hoi Vo. Vulcan: Binary Transformation in a Distributed Environment. Technical Report MSR-TR-2001-50, Microsoft, 2001.

[105] Amitabh Srivastava and Alan Eustace. Atom: a system for building customized program analysis tools. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 196–205, Orlando, Florida, United States, 1994. ACM Press.

[106] Artour Stoutchinin, José N. Amaral, Guang R. Gao, James C. Dehnert, Suneel Jain, and Alban Douillet. Speculative prefetching of induction pointers. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 289–303, London, UK, 2001. Springer-Verlag.

[107] SUN Microsystems. RMI Specification, 1998. http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmi-title.doc.html.

[108] P. van der Mark, E. Rohou, F. Bodin, Z. Chamski, and C. Eisenbeis. Using iterative compilation for managing software pipeline – unrolling tradeoffs. In *SCOPES 99, 4th International Workshop on Software and Compilers for Embedded Systems*, 1999.

[109] Steven P. Vanderwiel and David J. Lilja. Data prefetch mechanisms. *ACM Comput. Surv.*, 32(2):174–199, 2000.

[110] R. Veldema, R. F. H. Hofman, R. A. F. Bhoedjang, C. J. H. Jacobs, and H. E. Bal. Source-level global optimizations for fine-grain distributed shared memory systems. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 83–92, Snowbird, Utah, United States, 2001. ACM Press.

[111] Ronald Veldema and Michael Philippsen. Using Object Combining for Object Prefetching in DSM Systems. In *Proc. 11th Workshop on Compilers for Parallel Computers*, Seeon, Germany, 2004.

[112] Hong Wang, Perry H. Wang, Ross Dave Weldon, Scott M. Ettinger, Hideki Saito, Milind Girkar, Steve Shih wei Liao, and John P. Shen. Speculative precomputation: Exploring the use of multithreading for latency tools. *Intel Technical Journal*, 6(1):22–35, February 2002.

[113] Perry H. Wang, Jamison D. Collins, Hong Wang, Dongkeun Kim, Bill Greene, Kai-Ming Chan, Aamir B. Yunus, Terry Sych, Stephen F. Moore, and John P. Shen. Helper threads via virtual multithreading on an experimental itanium-2 processor-based platform. *SIGOPS Oper. Syst. Rev.*, 38(5):144–155, 2004.

[114] D. W. Westcott and V. White. Instruction sampling instrumentation, September 1992.

[115] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36, S. Margherita Ligure, Italy, 1995. ACM Press.

[116] Qiang Wu, Artem Pyatakov, Alexey Spiridonov, Easwaran Raman, Douglas W. Clark, and David I. August. Exposing memory access regularities using object-relative memory profiling. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 315, Palo Alto, California, 2004. IEEE Computer Society.

[117] Youfeng Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 210–221, Berlin, Germany, 2002. ACM Press.

[118] Catherine Xiaolan Zhang, Zheng Wang, Nicholas C. Gloy, J. Bradley Chen, and Michael D. Smith. System support for automated profiling and optimization. In *Symposium on Operating Systems Principles*, pages 15–26, 1997.

[119] Weifeng Zhang, Brad Calder, and Dean M. Tullsen. An event-driven multithreaded dynamic optimization framework. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 87–98, Washington, DC, USA, 2005. IEEE Computer Society.

[120] Weifeng Zhang, Brad Calder, and Dean M. Tullsen. A self-repairing prefetcher in an event-driven dynamic optimization framework. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 50–64, Washington, DC, USA, 2006. IEEE Computer Society.

[121] Qin Zhao, Rodric Rabbah, Saman Amarasinghe, Larry Rudolph, and Weng-Fai Wong. Ubiquitous memory introspection. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 299–311, San Jose, California, March 2007. IEEE Computer Society.

[122] Qin Zhao, Rodric Rabbah, and Weng-Fai Wong. Dynamic memory optimization using pool allocation and prefetching. *SIGARCH Comput. Archit. News*, 33(5):27–32, 2005.

[123] Craig B. Zilles and Gurindar S. Sohi. A programmable co-processor for profiling. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, page 241, Washington, DC, USA, 2001. IEEE Computer Society.