Recompilation et optimisation dynamique de code

Florent Bouchez

Lundi 31 janvier 2005

Table des matières

La plupart des méthodes d'optimisation de code sont faites statiquement sur le code lors de la compilation du programme source. Les méthodes spécifiques exploitent les caractéristiques du matériel sur lequel le code sera exécuté, comme par exemple des unités fonctionnelles parallèles ou des *pipelines*. Mais il existe des moyens d'améliorer les performances de manière dynamique en exploitant des informations tirées de l'observation de l'exécution en cours d'un programme. On distingue trois types de techniques : les techniques *matérielles* telles que l'exécution dans le désordre (le processeur exécute n'importe quelle instruction « prête ») ou la prédiction de branche ; les techniques *logicielles* qui commencent par récupérer des traces d'exécution à la volée : chemins les plus fréquemment choisis par exemple. Ces traces peuvent être grandes et même recouvrir des appels de fonctions, même dans des librairies. Et enfin, la *recompilation dynamique de code* où l'on retarde la compilation de certaines instructions jusqu'à l'exécution : par exemple, les compilateurs *Just-In-Time* (JIT) que ne compilent qu'au dernier moment, ou le processeur Crusoe™ [4] qui dont le CPU est un VLIW (*Very Long Instruction Word*) mais le code natif est de l'assembleur x86 : la transformation est faite pendant l'exécution. Ceci permet de choisir à la volée les instructions qui seront faites en parallèle et donc de tirer meilleur parti des capacités du processeur.

Je parlerai tout d'abord des processeurs Crusoe™, puis du projet Dynamo présenté en 2000 dans [1] ainsi que quelques mots sur le projet Deli [2], avant de terminer par quelques mots sur l'utilisation de ces techniques d'optimisation dynamique : les problèmes qu'elles posent et des idées tirées de [3] pour les améliorer.

1 Crusoe™

Les processeurs Crusoe sont produits par Transmeto Corporation et sont une famille de processeurs à faible consommation dont le jeu d'instructions est compatible avec l'assembleur x86. Le CPU est en fait de type VLIW et est capable d'exécuter jusqu'à quatre opérations en parallèle : une opération à virgule flottante, une sur nombres entiers, un accès mémoire (*load/store*) et un branchement. L'assembleur VLIW n'est absolument pas compatible avec la norme x86, c'est pourquoi les processeurs Crusoe contiennent une couche logicielle appelée *CodeMorphing*™chargée de recompiler à la volée les instructions x86 vers du code VLIW. Si les spécificités du processeur changent légèrement, entre deux modèles Crusoe par exemple, il suffit de modifier le logiciel de transformation de code alors qu'en général, il faut recompiler tous les programmes pour tirer le meilleur parti du processeur (ce qui signifie également que l'on est en possession du code source, ce qui n'est pas toujours le cas).

Le désavantage est que quelques cycles d'horloges sont nécessaires au processeur pour exécuter la couche logicielle *CodeMorphing*. Pour réduire le sur-coût occasionné, il est important de bien choisir la limite entre le logiciel et le matériel ; dans Crusoe, le logiciel a la tâche de décoder les instructions x86 et de produire des instructions VLIW explicitement parallèles, qui sont ensuite exécutées directement, mais aussi mises en mémoire cache en vue d'une réutilisation ultérieure, si le même groupe d'instructions est rencontré plus tard. Il est également possible d'optimiser différemment une même région de code, en fonction de son contexte d'utilisation.

La compilation dynamique à l'exécution gagne par rapport à la compilation normale dans les cas où elle peut se baser sur des informations disponibles à l'exécution, comme par exemple une branche beaucoup plus souvent prise qu'une autre, pour créer des versions optimisées du code. Crusoe utilise ces techniques qui relèvent de l'optimisation dynamique, et que je présente à travers le projet Dynamo dans la partie suivante.

2 Dynamo [1]

Dynamo est un projet créé en 1996 dont le but est de proposer un système dédié à l'optimisation dynamique de code. Il fonctionne de manière totalement logicielle sur un flot d'instructions natives, soit provenant d'une compilation statique classique, soit d'une application de type JIT.

2.1 Principe de fonctionnement

On peut voir Dynamo comme une machine virtuelle qui prend en entrée du code pour PA-8000 (un processeur RISC de Hewlett Packard) et qui s'exécute sur un PA-8000. C'est donc la méthode de l'interprétation qui a été gardée, dont le principal inconvénient est la lenteur mais dont les avantages sont de pouvoir s'abstraire des problèmes liés à la perturbation de l'exécution normale du code binaire par rapport à des méthodes d'échantillonnage.

Pour gagner en efficacité par rapport à une exécution directe, certains blocs d'instructions *clés* provoquent un changement du comportement de Dynamo :

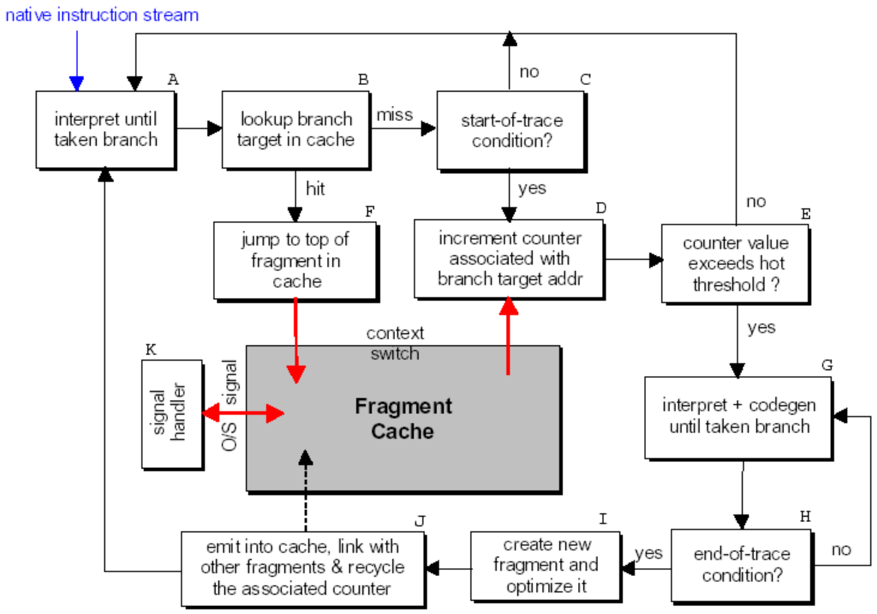

Figure 1. How Dynamo works

FIG. 1 – Fonctionnement de dynamo

il continue d'interpréter les instructions et mais génère à la volée une version optimisée, un *fragment*, qu'il stocke dans une mémoire cache jusqu'à un autre bloc clé qui indique à Dynamo d'arrêter la trace. La prochaine fois que l'exécution emprunte à nouveau la même branche, la partie optimisée est récupérée du cache et exécutée. Ces exécutions s'exécutent directement sur le processeur, et il n'y a donc plus de sur-coût dû à l'interprétation. L'intérêt de ces traces par rapport à une optimisation statique à la compilation est que le code des traces est en général linéaire : il y a peu de sauts puisque les branchements les plus utilisés sont écrits à la suite, et même les appels de fonctions sont remplacés par un déplacement du code correspondant à l'endroit nécessaire.

2.2 Fonctionnement général

La figure 1 montre le principe de fonctionnement de Dynamo. Initialement, on se trouve dans l'état A :

A. interprétation du code en entrée jusqu'à ce qu'on rencontre un branchement ⇒ B ;

B. la branche à prendre correspond-elle à un fragment optimisé et mis dans le cache ? Oui ⇒ F, Non ⇒ C ;

C. le branchement satisfait-il les conditions pour démarrer une trace ? Oui ⇒ D, Non ⇒ A ; pour vérifier les conditions, il faut soit être la cible d'un branchement qui revient en arrière (c'est le cas du début des boucles par exemple), ou être la suite d'un fragment du cache (il est alors avantageux de rester dans le cache lors de l'exécution) ;

D. incrémenter le compteur correspondant à ce branchement ⇒ E ;

E. si le compteur dépasse un seuil (i.e. la branche est souvent choisie) ⇒ G, sinon on continue l'interprétation ⇒ A ;

F. continuer l'exécution à l'endroit du cache correspondant au fragment (sans interprétation : exécution directe sur le processeur) ;

G. interpréter le code et enregistrer la suite des instructions (trace) dans un tampon jusqu'au prochain branchement ⇒ H ;

H. le branchement satisfait-il les conditions pour arrêter la trace ? Oui ⇒ I, Non, continuer la trace ⇒ G ; les conditions sont soit un branchement vers l'arrière ou les branches qui pointent vers des fragments mis en cache ;

I. générer à partir du tampon de la trace un fragment optimizé : une séquence d'instructions natives ⇒ J ;

J. mettre en mémoire cache le fragment généré, établir les connexions avec les autres fragments si nécessaire (évite les retours, coûteux, à dynamo entre fragments), et placer une instruction qui émet un signal quand le fragment se termine si la main doit revenir à l'interpréteur ; Puis ⇒ A ;

K. récupérer les signaux émis par le cache, et rendre la main à l'interpréteur.

Au fur et à mesure que l'exécution s'effectue, les parties de code les plus souvent exécutées sont transformées en fragments et très peu de code a encore besoin d'être interprété.

2.3 Optimisation des traces

Une fois la trace terminée, il reste à produire du code natif optimisé que l'on gardera dans le cache. L'optimisation consiste à « linéariser » l'exécution de manière à ce que les branches choisies les plus souvent soient directement à la suite des branchement, les autres résultant en général en une sortie de la

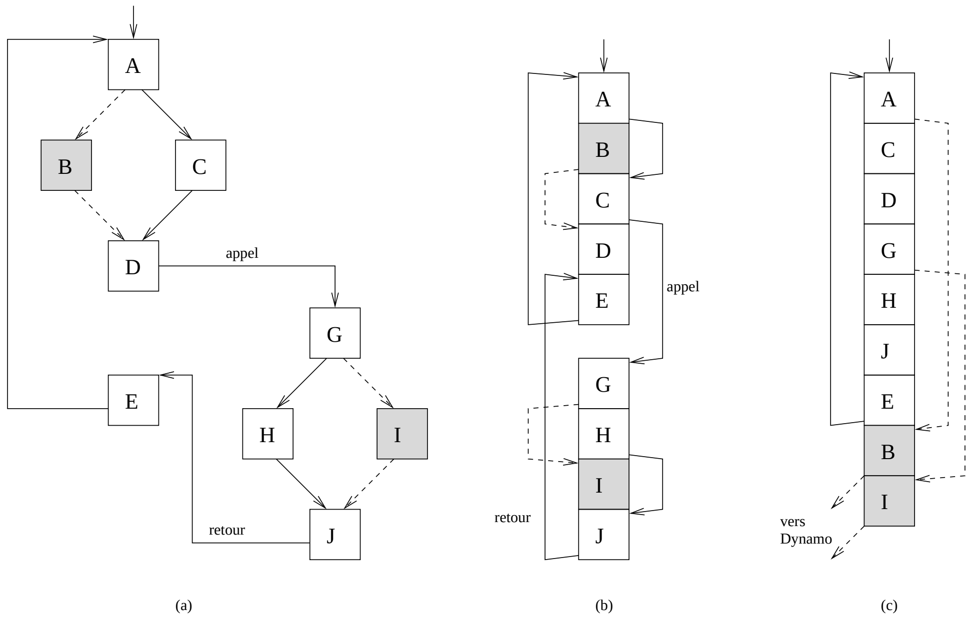

FIG. 2 – Exemple d'optimisation de trace.

trace, i.e. un retour à Dynamo. Les boucles sont autorisées seulement si le retour s'effectue vers le début de la trace, dans le cas contraire, on sort de la trace. L'exemple de la figure 2 est tiré de [1] et montre :

(a) le graphe de flot de contrôle d'une trace à optimiser ; les liens en pointillés et les blocs grisés sont peu utilisés ; on remarque que la trace recouvre un appel de fonction ;

(b) la disposition standard courante des instructions dans le code binaire ; en l'absence d'informations complémentaires pas de disposition particulière n'est privilégiée ;

(c) Dynamo sait que les blocs B et I sont peu utilisés à ce moment là de l'exécution du programme, il met à profit cette information pour éviter les sauts entre les blocs souvent exécutés, même le saut correspondant au changement de procédure ; on a ainsi un code linéaire avec un seul retour de boucle au lieu de cinq sauts, et les blocs peu exécutés quittent la trace et sont gérés par l'interpréteur de Dynamo.

3 Deli [2]

Deli signifie *Dynamic Execution Layer Interface*. L'article [2] date de 2002 et a été publié par des chercheurs travaillant dans les laboratoires de Hewlett-Packard, et la première référence bibliographique est l'article [1] sur le projet Dynamo ; un des auteurs a d'ailleurs participé à la rédaction des deux articles. Deli diffère de Dynamo car il fonctionne à base de points de contrôles durant l'exécution plutôt que par interprétation. Le gros apport de Deli est que c'est un outil permettant la modification des instructions binaires à la volée et fournit pour cela une API, *Application Programming Interface*, dont les fonctions de base permettent par exemple de déclencher la création des fragments, ou de donner ou reprendre la main à Deli.

4 État de l'art

Le problème de l'optimisation dynamique est qu'elle fonctionne bien si le code en entrée a été généré avec peu d'optimisations ; par contre, si des optimisations aggressives ont été faites, les optimisations dynamiques faites dessus sont faibles, et le sur-coût que nécessite la création de traces n'est plus contrebalancé par le gain des optimisations. En effet, les techniques de compilation statiques aggressives se basent notamment sur des informations récoltées à l'aide de traces d'exécution de programmes, ce qui ne laisse à l'optimisation dynamique que l'avantage de la localité : elle est par exemple capable de repérer qu'à un moment précis de l'exécution du programme, une branche est souvent choisie, alors qu'en moyenne elle l'est peu.

D'après David Hodgdon dans [3], pour que l'optimisation dynamique ait vraiment un intérêt, il faut qu'elle ait accès à des informations non disponibles par le processus de *profiling* sur des traces d'exécution : par exemple les valeurs des données sur lequel travaille le programme et leur état.

4.1 Problèmes de l'optimisation dynamique

Des trois techniques mentionnées dans l'introduction, seules les méthodes matérielles sont de nos jours vraiment utilisées dans le milieu industriel. Les optimiseurs dynamiques et compilateurs dynamiques sont assez peu acceptés et David Hodgdon expose quatre raisons à cela pour chaque méthode.

– Les optimiseurs dynamiques logiciels tels que Dynamo :
    1. sur-coût d'observation nécessaire à l'interprétation ou à l'exécution de code marqué qui masque le bénéfit des optimisations ;
    2. décisions faibles : les analyses sont limitées dans le temps, et les décisions qui en résultent sont donc parfois peu appropriées et mènent à une perte de temps (construction d'un fragment inutile par exemple) ;
    3. les optimisations aggressives faites durant la compilation statique à partir de *profiling* laissent peu de place aux améliorations dynamiques ;
    4. fiabilité : les optimiseurs dynamiques sont par nature amenés à contenir des bogues de type non-reproductibles, qu'il est donc difficile à la fois de détecter mais aussi de corriger.
– Les compilateurs dynamiques comme celui utilisé dans Crusoe :
    1. sur-coût pour gérer la compilation à la volée du code, en particulier si une même région est optimisée différement en fonction de l'utilisation, il faut alors retrouver la bonne spécialisation ;
    2. travail nécessaire de la part du programmeur qui doit souvent annoter son code pour indiquer les régions à optimiser par exemple ;
    3. le code source doit être disponible puisque la compilation se fait dynamiquement à l'exécution et ce n'est pas souhaitable dans le cas de programmes commerciaux ;
    4. fiabilité : les compilateurs dynamiques peuvent aussi contenir des bogues.

David Hodgdon propose alors des pistes d'exploration, dans lequelles les compilateurs et les optimiseurs dynamiques coopèrent : les premiers fournissent aux seconds des informations clés qui ne peuvent être obtenues que par une analyse générale du programme, lesquelles montrent où l'analyse dynamique pourra améliorer le code. Cette coopération permet de réduire le sur-coût grâce à des informations déjà disponibles, de gagner en efficacité même sur des programmes fortement optimisés, et d'aider à la fiabilité dans le cas de la dernière amélioration (tests des limites).

4.2 Améliorations possibles

Constantes à l'exécution. Beaucoup de possibilités d'optimisations sont perdues car on ne connait pas à la compilation les valeurs des constantes à l'exécution : ce sont des constantes pour une exécution donnée du programme mais peuvent varier selon les exécutions. C'est par exemple les valeurs déterminées dans des fichiers de configuration, ou par les options passées en ligne de commande. En revanche, un compilateur peut souvent déterminer qu'une variable est en fait une constante d'exécution, il peut alors en informer l'optimiseur dynamique qui, lui, saura utiliser cette donnée (par exemple pour de la propagation de constante, ou du déroulage de boucle).

De même, il existe des variables *presque*-constantes, c'est-à-dire dont la valeur change peu au cours de l'exécution, et donc apparait comme constante sur de longues périodes, le compilateur peut connaitre les points de changement et donner ces informations à l'optimiseur dynamique, qui prendra soin d'invalider tout fragment optimisé sur la base de cette presque-constante à ces instants et de reconstruire de nouvelles traces optimisées.

Transformations des accès mémoire. Les accès à la mémoire sont toujours lents, ce qui conduit les compilateurs à essayer de placer les *load* le plus tôt possible avant l'instruction qui en a besoin, et de servir du délai pour exécuter d'autres instructions. Il y a certaines conditions à respecter pour assurer la validité de l'opération, par exemple s'assurer que l'adresse mémoire est la bonne et que l'endroit de la mémoire contient effectivement la donnée. Par exemple dans le cas d'un *load* utilisé par une instruction conditionnelle, il est possible que la donnée ne soit pas définie si la condition est fausse, et il est alors incorrect de placer le *load* avant la condition.

Il est évident que si l'analyse statique ne peut prévoir ce genre d'événements, un optimiseur dynamique peut par contre recueillir les informations nécessaires au cours de l'exécution : quels objets sont vivants, quelles sont les valeurs des pointeurs, etc. Le compilateur peut alors insérer des notes à l'optimiseur dynamique près des appels mémoire lui indiquant les conditions à remplir pour effectuer plus tôt les *load*.

Pointeurs et limites. Les bogues concernant des problèmes de pointeurs ou de dépassement de limites (comme l'accès à une case d'un tableau en dehors de son domaine de définition) sont souvent difficiles à corriger, aussi la tendance est-elle à ajouter des tests dans le code pour éviter ces problèmes. Mais ces tests sont souvent coûteux par rapport à un accès direct sans vérification. Des tests peuvent se révéler être inutiles durant l'exécution, par exemple car on connait alors la valeur des constantes d'exécution ; l'optimiseur dynamique est alors capable de supprimer ces tests ce qui apporte un gain significatif dans le cas de boucles sur des tableaux par exemple, tout en assurant que les accès restent valides.

Environnements distribués. Ce dernier paragraphe concerne la remarque suivante : dans un environnement distribué, utilisé par exemple pour faire du calcul scientifique, de nombreux processus travaillent en parallèle et vont créer des traces similaires, donc perdre tous du temps à faire les mêmes optimisations. David Hodgdon a développé l'idée d'un processus central dont le travail serait de coordonner les traces et profils créés par les clients pour éviter ce problème. Des problèmes liés à la latence apparaissent : si un client demande l'accès à une trace qui n'existe pas par exemple, on peut imaginer alors que le client commence à faire sa trace par mesure de prudence, et utilisera celle fournie par le serveur plus tard le cas échéant.

5 Conclusion

Les compilateurs et optimiseurs dynamiques sont donc un autre moyen d'améliorer l'exécution de code, en optimisant à la volée en fonction du comportement du programme. On peut pour cela utiliser des traces dans lesquelles les chemins les plus empruntés sont réordonnés pour bénéficier d'une optimisation locale impossible à déterminer statiquement.

Cependant, ces optimiseurs gagneraient à être utilisés en coopération avec des compilateurs statiques traditionnels qui parfois disposent d'informations importantes obtenues de leur analyse de programme ou de *profiling* sur de nombreuses entrées, mais ne peuvent les utiliser, en l'absence de données dynamiques déterminées uniquement pendant l'exécution de programmes.

Dans le cadre des systèmes embarqués, ces méthodes permettent de gagner encore un peu en efficacité, ce qui au choix permet d'exécuter des programmes plus rapidement ou de réduire la fréquence d'horloge des processeurs. Par contre, ces techniques nécessitent de la mémoire cache supplémentaire et il serait intéressant d'évaluer par exemple l'énergie nécessaire pour entretenir soit un cache supplémentaire, soit un agrandissement des caches utilisés.

Références

[1] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo : a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5) :1–12, 2000.

[2] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. Fisher. Deli : A new run-time control point, 2002.

[3] David Hodgdon. Efficient, compiler-aided dynamic optimization, 2004.

[4] Alexander Klaiber. The technology behind crusoe$^{\mathrm{TM}}$processors, 2000.