Département informatique
UFR de sciences

École doctorale SOFT
Université de Versailles Saint-Quentin en Yvelines

Thèse d'habilitation à diriger les recherches
Spécialité Informatique
présentée par

Henri-Pierre Charles

Sujet

Contributions à la génération de code d'applications multimédia sur processeurs spécialisés

Habilitation à diriger les recherches
présentée et soutenue publiquement le 8 Décembre 2008 devant le jury composé de

- Président : Bertil Foliot, Professeur, Université de Paris 6
- Rapporteurs :
  - Albert Cohen, Directeur de recherche INRIA
  - Lawrence Rauchwerger, Professeur, Université Texas A&M,
  - Tanguy Risset, Professeur, Insa Lyon
- Examinateurs
  - François Bodin, Directeur Technique, CAPS Entreprise
  - Jean Claude Fernandez, Professeur à l'Université Grenoble I
  - William Jalby, Professeur, Université de Versailles Saint-Quentin en Yvelines

Résumé

Du texte écrit par un programmeur jusqu'au programme exécutable, une chaîne de logiciels est utilisée pour produire un programme exécutable. Cette chaîne de compilation s'est raffinée au fur et à mesure des évolutions respectives des processeurs, des machines et des applications.

Dans cette thèse, je montrerai que pour les applications nécessitant de hautes performances [1] les compilateurs n'arrivent pas à exploiter au mieux les performances disponibles dans les processeurs, nous verrons quelles en sont les raisons.

Dans un nombre grandissant de domaines, la nature des données manipulées par le programme impacte la performance. Par exemple dans les applications multimédias. Ce phénomène n'est pas pris en compte par les compilateurs statiques.

La prise en compte de l'empreinte mémoire de l'application finale, la spécialisation de code et la génération dynamique de code paramétrée par les données seront les trois exemples de modification de la chaîne classique de compilation qui illustreront ma présentation.

Abstract

From the text writed by a programmer to the executable program, a sofware toolchain is used to produce executable. This compiler toolchain has been refined during processors, computers and application evolution.

In this thesis, I will show that HPC domain applications cannot reach peak performances because compilers had some drawbacks. Some of them will be explained.

In a growing number of domains, data set has a major impact on performance. In multimedia applications for instance.

Memory footprint of final application as a parameter, code specialization and data driven dynamic code generation will be three examples of compilation chain modification during shown in this thesis.

[1] HPC ne sont pas uniquement mes initiales `http://fr.wikipedia.org/wiki/High_performance_computing`

Remerciements

À Martine, Delphine et Marion qui partagent, avec patience, l'existence d'un chercheur en informatique.

Je remercie tous les étudiants que j'ai pu croiser qui par leurs questions, même les plus naïves, m'ont permis d'avancer dans la recherche de la connaissance.

Les membres de mon jury et particulièrement les rapporteurs, me permettent de réaliser cette soutenance et leurs commentaires sont pour moi un encouragement et une ouverture vers des projets futurs, je les en remercie.

Merci à William Jalby qui m'a permis de redémarrer une activité de recherche que je n'avais plus il y a quelques années. Cela ma permis de rééquilibrer mon activité universitaire.

Merci enfin à tous ceux qui se battent pour que les progrès de l'informatique soient utiles pour le plus grand nombre. Les logiciels libres ne sont pas une fin en soit, mais un moyen de rééquilibrage et de justice.

Table des matières

Table des figures

Chapitre 1

Introduction

Lorsque j'ai soutenu ma thèse en 1993, le "super-ordinateur" parallèle de l'époque était l'Intel iPSC/860, machine fabriquée par Intel, basée sur le microprocesseur i860 (ou 80860[1]).

Ce processeur était révolutionnaire pour l'époque (en même temps que le x486), son argument de vente était "a Cray on one chip". En effet c'était un processeur RISC, avec des opérateurs pipe-line, capable d'effectuer 80 MFlops, ce qui était une performance. Malheureusement, même en le programmant manuellement en assembleur, il était très difficile d'obtenir ces performances crêtes. Les meilleurs compilateurs arrivaient péniblement à en tirer 10 MFlops, les difficultés étaient liées à l'utilisation du pipe-line, les instructions de type SIMD, la bonne gestion des caches [7] et la gestion des registres. Quelques années plus tard, les compilateurs spécialement conçu pour cette architecture commençaient à en tirer de meilleures performances

De plus l'architecture de la machine iPSC était basé sur un hyper-cube, architecture élégante et puissante car elle permet d'augmenter exponentiellement le nombre de processeurs en ne faisant augmenter la distance entre noeuds que linéairement. Mais cette architecture était difficilement exploitable à cause de la complexité de la topologie et du peu de ressource sur chaque noeud qui ne contenait pas un système d'exploitation entier, ni de stockage local. Arriver à recouvrir calculs et communications était considéré comme un exploit [8].

15 ans plus tard, le processeur CELL [2] né d'un consortium entre Sony, Toshiba et IBM, concentre toutes ces technologies (RISC, opérateurs pipe-lines, opérateurs SIMD, topologie en anneau entre noeuds de calculs, pas de stockage local, etc) en rajoute d'autres (multiples ISA, gestion de DMA) dans un seul microprocesseur. Les performances crêtes sont au rendez-vous : 1 Teraflop par processeur (avec une version contenant 32 SPE) ; c'est à dire 12 million de fois plus puissant que le i860 (ce qui dépasse les prévision de M. Moore). Mais là encore les performances que l'on peut obtenir dans la pratique sont sans commune mesure. Aucun compilateur ne sait tirer partie de cette puissance.

Pourquoi en 15 ans de recherche dans la compilation, des problèmes qui paraissent similaires n'ont pas été résolus ? Quelles sont les transformations qui ont conduit à cette situation ? Pourquoi la recherche en compilation n'a pas permis d'exploiter cette puissance ?

À défaut de résoudre les problèmes évoqués plus haut, j'espère convaincre le lecteur que des changements radicaux doivent être réalisés dans le domaine de la compilation afin de permettre d'utiliser au mieux les ressources offertes par les nouvelles architectures.

[1] `http://en.wikipedia.org/wiki/Intel_i860`

[2] `http://en.wikipedia.org/wiki/Cell_(microprocessor)`

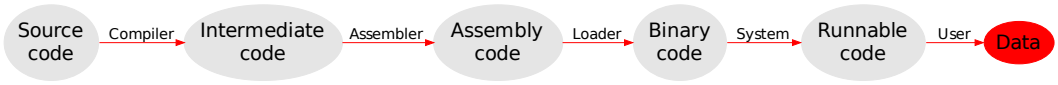


FIG. 1.1 – La chaîne usuelle de compilation : du programme vers les données

Les chapitres suivants montreront quelques aspects des recherches que j'ai mené dans le domaine de la compilation, qui ne résolvent malheureusement pas les problèmes que j'ai évoqués mais que l'on peut voir comme des tentatives d'approches par des angles différents.

Dans ce premier chapitre d'introduction j'essayerai de montrer quels sont les évolutions et les ruptures qui ont conduit à cette situation mais également donner les raisons de l'inadaptation des compilateurs actuels à exploiter les nouvelles architectures et les nouvelles applications.

Dans le chapitre 2 je montrerai une adaptation des chaînes de compilation de façon à prendre en compte la taille de l'empreinte mémoire de la partie exécutable d'un programme ce qui est un critère important dans certains domaines. Dans le chapitre 3 je montrerai comment on peut aider un compilateur à générer un code de meilleure qualité en utilisant la spécialisation et le chapitre 4 montrera quelques réalisation dans la production de code au moment de l'usage des données.

Une conclusion esquissera quelque pistes possibles d'évolutions de ces recherches.

1.1 Le compilateur : boite de Pandore

Le compilateur est un élément central des systèmes informatique moderne, il permet de transformer automatiquement un programme rédigé dans un langage informatique rédigé par un programmeur vers un autre langage informatique, en général de plus bas niveau comme le langage machine d'un processeur.

Cette chaîne est symbolisée sur la figure 1.1 dans laquelle les différentes version d'un même programme sont représentés par des cercles et les outils sur les transitions.

Les transformations successives adaptent le programme du contexte du programme vers le contexte du processeur que l'on utilise. Malheureusement pour les écrivains de compilateurs les différents contextes d'utilisation des compilateurs ont beaucoup évolués, nous allons voir dans les sections suivantes quelles sont ces évolutions et leurs conséquences sur notre domaine de recherche.

1.2 L'architecture des processeurs

1.2.1 Les ISAs

Il y a 15 ans les ISAs (Instruction Set Architecture ou Jeu d'instruction) des processeurs étaient très simples et une opération élémentaire d'un programme se décomposait en une séquence d'instruction machine.

Un compilateur pouvait était conçu comme une décomposition d'un programme vers des séquences d'instructions élémentaires. Le programme décomposé en arbre d'expression, est ensuite parcouru pour remplacer les arbres en séquences d'expression, c'est ce qui est décrit dans les manuels standards sur la compilation [2] et implémenté dans les premières versions du compilateur GCC [12].

Pour illustrer l'évolution des architectures des processeurs nous prendront l'exemple de deux processeurs, l'Itanium qui est un processeur récent et l'ARM plus ancien et surtout successeur de la série 680X0 de Motorola de la fin des années 1980. Prenons la fonction suivante :

```
float MulAdd(float *a, float *b, int len)
{
  int i;
  float s = 0;

  for (i = 0; i < len; i++)
    s += a[i]*b[i];
  return s;
}
```

est décomposée pour le processeur ARM dans la séquence d'instructions suivantes (on n'a indiqué que les instructions correspondant à la boucle) :

```
.L5:
    ldr    r1, [r4, r7]    @ float
    ldr    r0, [r4, r8]    @ float
    bl    __mulsf3  // Math library call
    mov    r1, r0
    mov    r0, r6
    bl    __addsf3 // Math library call
    add    r5, r5, #1
    cmp    r5, sl
    mov    r6, r0
    add    r4, r4, #4
    bne    .L5
```

Sur une architecture plus récente comme l'Itanium, le programme donne la séquence suivante :

```
.L5:
    .mmi // Instruction 1
    add r14 = r32, r16
    add r15 = r16, r33
    adds r16 = 4, r16
    ;;
    .mmi // Instruction 2
    ldfs f7 = [r14]
    ldfs f6 = [r15]
    nop 0
    ;;
    .mfb  // Instruction 3
    nop 0
    fma.s f8 = f7, f6, f8
    br.cloop.sptk.few .L5
    ;;
```

Le changement n'est pas forcément flagrant, les codes sont de longueur semblable, pourtant un grand nombre de différences peuvent se voir sur ces codes :

– le processeur ARM n'a pas d'opérateur pour les nombres réels, le code doit donc faire des appels à des fonctions spécialisées pour réaliser les opérations d'addition (`bl _mulsf3`) et de multiplication (`bl __addsf3`)
  l'architecture Itanium a, non seulement les opérateurs pour les nombres flottants, mais également une opération `fma` qui permet de réaliser en un cycle l'opération de multiplication addition.
  l'architecture Itanium a intégré des opérateurs flottants et une opération complexe "multiply and add" pour les nombres réels.

- le processeur ARM utilise un mode adressage complexe `ldr r1,[r4, r7]`. Le registres `r7` contient l'adresse de base du tableau, le registre `r4`, le décalage par rapport à la base. C'est un adressage indirect indexé.
  Cet adressage complexe permet un accès aisé aux données, mais est une instruction complexe qui nécessite de réaliser une addition avant l'accès à la mémoire, le processeur Itanium est un processeur RISC les adressages sont plus simples et le grand nombre de registres permettant de conserver les calculs intermédiaires compense ce manque de souplesse.
- le processeur ARM utilise 11 instructions pour la fonction, le processeur Itanium qui utilise beaucoup d'unités fonctionnelles en parallèle n'en utilise que 3.
  L'ILP `Instruction Level Parallelism` de l'Itanium permet de paralléliser le code au niveau du processeur (même si cet exemple n'est pas le plus convaincant).

Cet exemple illustre l'évolution de l'ISA des processeurs qui propose des instructions de plus en plus spécialisées et proches des instructions des langages de haut niveau. Mais cette évolution est allée plus loin et les processeurs actuels ont des instructions qui ne permettent plus de construire un compilateur en décomposant un programme de haut niveau, mais en recherchant les algorithmes utilisés. La construction d'un compilateur n'est plus une série d'étapes décomposant un programme de haut niveau vers le bas niveau, il faut effectuer des reconnaissance d'algorithmes.

Un autre exemple plus complexe illustrera mon propos. La compression d'image MPEG est basée sur des recherches de blocs similaires dans des images voisines. Une des opérations de base est SAD qui réalise la somme des valeurs absolues des différences de plusieurs pixels. Par exemple le compresseur X264 du projet videolan [23] l'implémente de la façon suivante[3] :

```
static int SAD ( uint8_t *pix1, int i_stride_pix1,
                 uint8_t *pix2, int i_stride_pix2 )
{
    int i_sum = 0;
    int x, y;
    for( y = 0; y < 8; y++ )
    {
        for( x = 0; x < 8; x++ )
        {
            i_sum + = abs( pix1[x] - pix2[x] );
        }
        pix1 += i_stride_pix1;
        pix2 += i_stride_pix2;
    }
    return i_sum;
}
```

"On reconnaît facilement" l'instruction `psad1` dont le schéma explicatif est représenté sur la figure 1.2. Cette instruction est capable de réaliser en une seule instruction l'intégralité de la boucle `for (x = 0 ...` du programme précédent.

Cette fois ci un compilateur ne doit plus décomposer un programme en sous instructions simples mais analyser et reconnaître la partie de programme qui pourra être remplacée par une seule instruction assembleur.

[3] Ce code est l'instance 8x8 (fichier `common/pixel.c`). Sept instances existent au total 16x16 8x16, 16x8, 8x8, 8x4, 4x8 et 4x4.

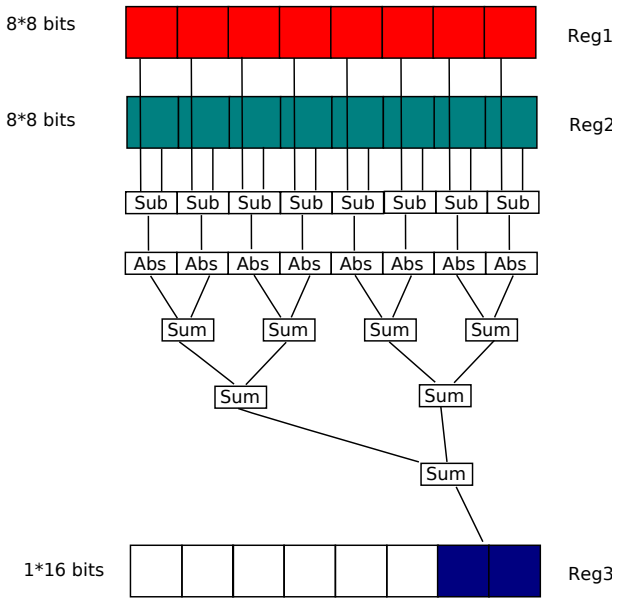


FIG. 1.2 – L'instruction `psad1` du processeur Itanium, utilisée pour la compression vidéo

L'expressivité des langages de programmation et la multitude de possibilité permettant d'écrire un tel algorithme font qu'un compilateur ne sera jamais capable d'exploiter ce genre d'instruction.

C'est une des raisons pour lesquelles un compilateur ne pourra jamais exploiter toutes les ressources d'un processeur et arriver à exploiter les performances crêtes : les instructions permettant d'utiliser la performances sont trop complexes pour être utilisée par un compilateur d'un langage de type C ou Fortran.

1.2.2 L'accès aux données

Sur un autre axe, l'évolution des processeurs et l'évolution des finesses de gravure a permis d'atteindre pour une même architecture des fréquences de fonctionnement très élevées. La figure 1.3 indique l'évolution des fréquence des processeurs comparée à l'évolution des temps d'accès à la mémoire.

Les mémoires ne peuvent donc pas alimenter les processeurs en données, les processeurs et architectes de machines ont mis en place plusieurs niveau de cache permettant, dans le meilleurs des cas de rejoindre les performances crête des processeurs.

Les mécanismes de cache fonctionnent très bien pour des codes réguliers, mais dans le cas de codes irréguliers ou de codes dont le comportement dépend de la nature des données (tris, multimédia), l'utilisation des caches peut avoir un effet catastrophique.

Il faut savoir qu'un défaut de cache sur une architecture Itanium peut bloquer le processeur pendant des centaines de cycles. Il est important de noter que cela peut se produire malgré tous les efforts que le compilateur peut produire pour planifier les instructions du programme.

L'impact de l'accès aux données est devenu plus important que la planification des instructions

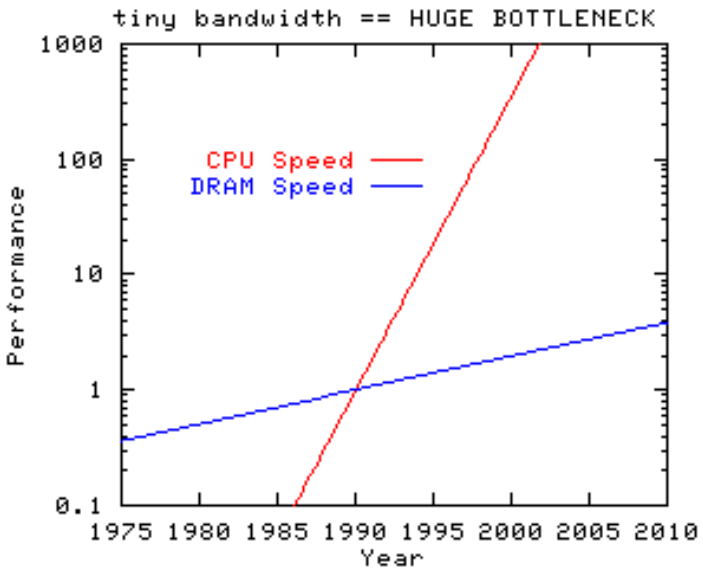


FIG. 1.3 – Évolution relatives des fréquences de fonctionnement des processeurs et des mémoires (Extrait de `http://www.cs.virginia.edu/stream/`)

1.2.3 Les processeurs aux jeux d'instruction multiples

Il existe de nombreux processeurs qui acceptent plusieurs jeux d'instruction, soit parce qu'ils possèdent des extensions que l'on peut programmer, soit parce qu'ils acceptent plusieurs jeux d'instructions différents.

La génération de code binaire est plus ou moins difficile suivant les cas. Trois exemples de processeurs illustreront ce propos

Le processeur x86

La famille des processeurs x86 possèdent de nombreuses extensions permettant d'accélérer des calculs multimédia (MMX, SSE). Ces extensions ont connus plusieurs variantes ajoutant quelques instructions spécialisées à chaque variante [4].

L'utilisation de ces instructions se fait soit :

- en utilisant des compilateurs vectorisants capable de reconnaître les possibilités d'usage de ces instructions
- dans des librairies qui permettent de s'assurer que les données d'entrées sont au bon format et dans un alignement favorable avant l'utilisation des données.
- ou encore par l'emploi d'extensions du compilateur permettant d'insérer dans le flux d'instructions une instruction spécialisée.

C'est la famille de processeurs généraliste que l'on retrouve dans beaucoup de machines de bureau ou de serveurs.

Le processeur CELL

Le processeur CELL est un autre exemple car il permet d'utiliser 3 jeux d'instruction qui ne sont pas des extensions mais 3 jeux spécialisés.

power est le jeu d'instruction RISC général de la famille POWER d'IBM

altivec est le jeu d'instruction vectoriel que l'on peut entrelacer avec le jeu d'instruction power

[4] L'extension SSE5 sera disponible dans les prochains processeurs AMD

spe[5] est le troisième jeu que l'on peut utiliser sur les unités SPE qui fonctionnent en parallèle avec l'unité power.

L'usage de ces jeux d'instruction séparés est toujours délicat et il n'existe pas de solution satisfaisante pour un programmeur non averti.

Ce processeur est utilisé pour des machines spécialisées (la PS3 de sony) ou comme accélarateur.

Le processeur ARM

Le processeur ARM possède également plusieurs jeux d'instructions mais si les extensions SSE ou le CELL permettent d'ajouter des fonctionnalités non présentes dans le jeu initial, l'ARM permet d'utiliser des codage différents d'instructions :

ARM32 est le jeu d'instruction 3 adresse codé sur 32 bits, classique dans un processeur RISC

THUMB est un jeu d'instruction codé sur 16 bits, 3 adresses, mais avec un nombre réduit de registres.

Jazelle est un jeu d'instructions codé sur 8 bits, 0 adresse, qui permet d'exécuter nativement la quasi totalité du bytecode Java.

Le jeu d'instruction 32 bits permet d'utiliser la totalité des registres et toutes les instructions ainsi que les instructions multimédia, permettant d'obtenir de bonnes performances. Les autres (Thumb et Jazelle) permettent d'obtenir un code beaucoup plus compact au prix de performances moindres.

Il est possible au cours de l'exécution de passer d'un codage à un autre pour utiliser le jeu 32 bits pour les parties du code nécessitant de bonnes performances et les autres codages pour les parties nécessitant une petite empreinte mémoire.

Cette famille est très utilisée pour des machines spécialisées : téléphones, machines embarquées, etc.

1.3 Les utilisateurs

Le marché de l'informatique connaît une croissance sans égale. Et si l'on considère que n'importe que périphérique "numérique" contient un processeur et de la mémoire (téléphone mobile, appareil photo numérique, baladeur MP3, GPS, "box multiplay" des fournisseur d'accès, disque USB, ordinateur de bord de voiture, satellite de télécommunication, télévision, console de jeux, imprimante laser, carte à puce, tous les ordinateurs plus classiques du mini portable au serveur internet en passant par les modems ADSL et les routeurs), on peut dire que toute personne sur terre est utilisatrice d'un ou plusieurs ordinateurs.

Cette croissance à fait naître un besoin énorme d'outils de développement pour des besoins extrêmement variés et nombreux. Cela a provoqué une évolution des langages de programmation dont nous reparlerons dans la section 1.6, mais surtout une évolution d'un marché mondial gigantesque couvrant une multitude de domaines : marché du divertissement (films, musique et jeux), marché de la télécommunication (téléphones mobiles, fournisseurs d'accès à internet), marché de l'informatique personnelle etc.

Si l'on peut se réjouir de la popularisation des outils informatiques, on peut regretter que les enjeux économiques n'orientent pas le marché vers une utilisation optimale des outils informatique. Les objectifs à atteindre sont de réduire le "time to market" et d'augmenter les marges à tout prix, pas d'optimiser le l'utilisation des ressources, ni d'éduquer l'utilisateur.

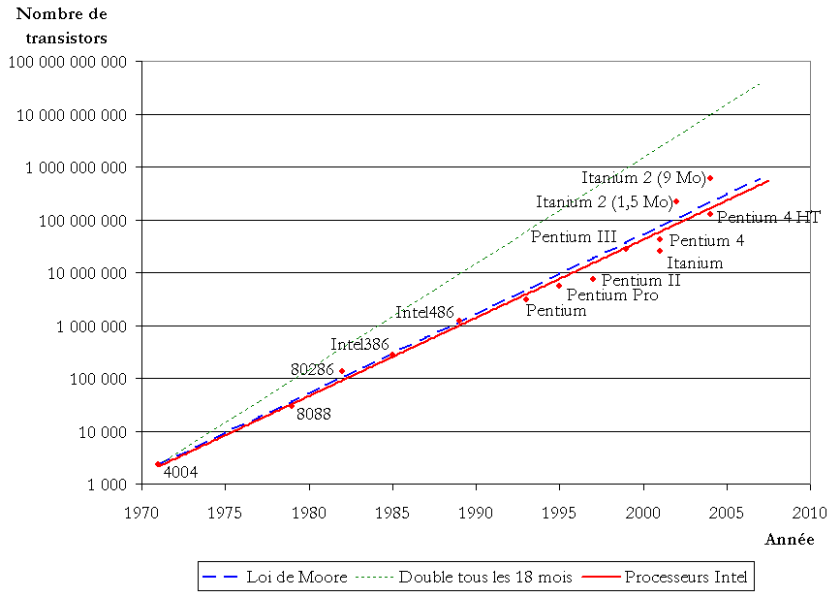


FIG. 1.4 – La loi de Moore et l'évolution des processeurs Intel

Dans ce contexte il paraît paradoxal que l'utilisateur ne bénéficie pas de la loi de Moore qui se révèle exacte depuis plus de 30 ans. Elle est représentée sur la figure 1.4 et décrite dans un article célèbre [20].

Si les processeurs évoluent si vite, le démarrage d'un système ou d'une applications devraient être instantané. L'utilisation quotidienne montre qu'il n'en est rien car dans le même temps les applications grossissent encore plus vite. Ce paradoxe a été décrit dans un article de N. Wirth [28], on le nomme maintenant la loi de Wirth.

Il existe des domaines dans lesquels l'optimisation de code est une demande forte (calcul scientifique, application multimédia, cryptographie), mais dans la majeure partie des codes applicatifs, le contexte économique ne pousse pas à exploiter au mieux les ressources des processeurs, c'est même économiquement logique de ne pas optimiser.

1.4 La multiplicité des applications

Comme nous l'avons vu dans la section précédente les usages des ordinateurs sont très variés, ce qui a conduit à une multiplicité d'applications et d'algorithmes inimaginable il y a seulement 30 ans.

La multiplicité des données manipulée est énorme, et leur traitement est souvent spécifique. La majeure partie des langages permet d'utiliser des données de type entier ou réels et leurs variantes (variantes de longueurs, variantes d'arithmétique signée ou non).

Ces langages ont été normalisés afin de pérenniser les investissements des entreprises. Un programme écrit dans un langage peut être facilement recompilé pour une autre architecture et donc avoir une durée de vie très grande.

Les traitements pour ces données ont naturellement été implémentés dans tous les processeurs, nous avons vus qu'un processeur 68000 ne possède pas d'arithmétique flottante mais qu'un Itanium en implémente plusieurs variantes.

Mais l'évolution des architectures ne s'est pas arrêtée et a continué d'implémenter des instructions spécialisée (comme le `psad1` vu plus haut), mais également des

Parallel Add Saturation Limits

Size	Element Width	Result r_j signed		Result r_j unsigned	
		Upper Limit	Lower Limit	Upper Limit	Lower Limit
1	8 bit	0x7f	0x80	0xff	0x00
2	16 bit	0x7fff	0x8000	0xffff	0x0000

Parallel Add Examples

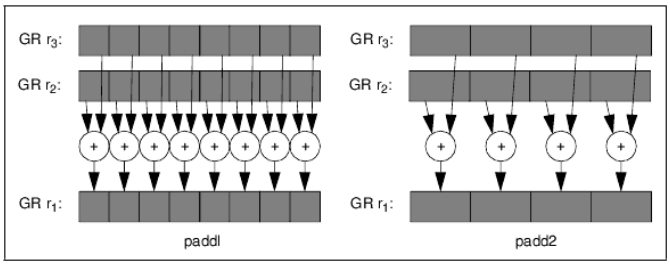


FIG. 1.5 – Exemple de l'instruction padd de l'Itanium

type de données et d'arithmétique qui n'existent pas dans les langages de programmation.

Deux exemples typique viennent du domaine du multimédia :

l'arithmétique saturée le codage et la manipulation des pixels utilisent cette arithmétique particulière dans laquelle les résultats sont bornés dans un intervalle (généralement $[0, 255]$).

Tous les processeurs récents implémentent cette arithmétique dans des extensions d'ISA (MMX et SSE pour Intel, Altivec pour Power, PTX cell pour Sony, Vis pour Sparc). Par exemple les règles de saturation de l'instruction `padd` de l'Itanium sont résumées dans la figure 1.5.

Le programmeur n'ayant pas le moyen de l'exprimer dans un langage de programmation, il la simulera par logiciel et, là encore, devant la multiplicité des implémentations possibles, aucun compilateur ne pourra reconnaître l'utilisation de cette arithmétique et utiliser les instructions correspondantes.

la vectorisation les applications 3D utilisent énormément des vecteurs de dimension 4 qui permettent d'utiliser les coordonnées homogènes ou la représentation RGB des couleurs.

Mais le choix des types de données et des structures de données utilisés est laissé à l'entière liberté du programmeur qui n'est pas forcément au courant des possibilités que lui offre un processeur ou de ses spécificités qui permettent d'utiliser au mieux le processeur.

Par exemple la figure 1.6 indique quelles sont les instructions permettant de réaliser une opération d'addition pour la plateforme Itanium : 5 types d'arithmétiques sont disponibles (complément a 2, réel IEEE 754, virgule fixe et pointeur), la taille des vecteurs peut aller de 1 à 8 et la taille des résultats de 8 a 128 bits. Je n'ai indiqué ici que les instructions qui réalisent une addition directement et non comme effet de bord, comme l'incrémentation, etc.

Un compilateur ne pourra prendre en compte que les instructions utilisant les arithmétiques habituelles (entière et flottante), qui correspondent aux feuilles vertes de la figure 1.6. Un compilateur optimisant peut dans certains cas favorables utiliser des instructions vectorielles correspondant aux feuilles bleues. Les feuilles rouges correspondent aux instructions d'addition qui ne pourront pas être utilisées, c'est à dire la majorité des instructions.

Bien sûr de nombreux travaux permettent la détection et l'utilisation des opérations vectorielles [4],[10] mais si ces travaux donnent de bon résultats sur des

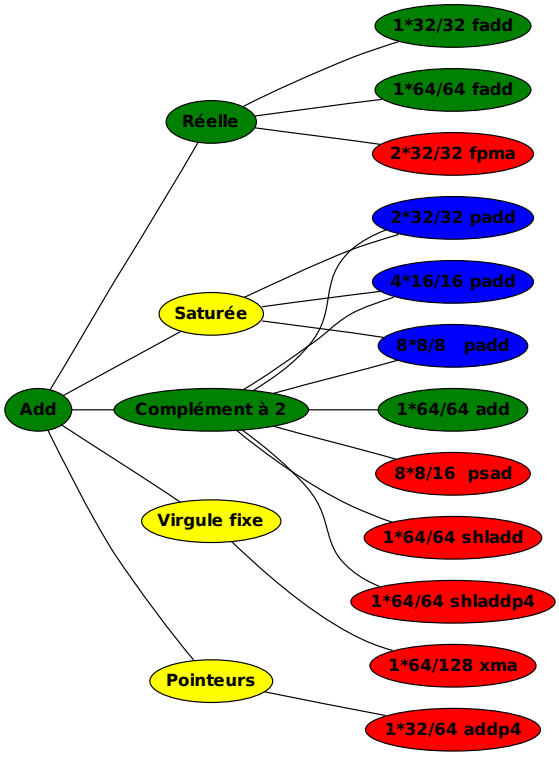


FIG. 1.6 – Instructions d'addition disponibles pour Itanium

codes assez réguliers, il semble irréaliste de croire que cette multiplicité d'opérateur et de typage des données pourra être utilisée dans des compilateurs avec des langages conçus 30 ans plus tôt.

Ces problématiques ont longtemps été le souci des administrateurs de gros systèmes parallèles mais ce n'est plus le cas depuis l'émergence de systèmes hybrides (téléphones, PDA, appareils photos, caméra). Par exemple les téléphones portables récents sont capable d'encoder des vidéos avant de les transmettre à des correspondants via une connection sans fil.

1.5 Les données

Les données traitées par un programme jouent un rôle de plus en plus important sur les performances des programmes, nous avons vus que l'impact des caches peut être énorme.

Les benchmark SPEC [9] fournissent des jeux de données de référence qui permettent de valider la vitesse d'exécution sur différents jeux de données. Mais ces jeux de données sont trop peu nombreux pour valider toutes les situations possibles pour des codes complexes.

Par exemple l'organisation MPEG [21] a définit un jeux de données vidéo contenant 32 vidéos dont les tailles et les caractéristiques changent.

1.5.1 La taille des données

L'augmentation des fréquences de traitement et de la quantité de mémoire permettent de traiter des données de plus en plus volumineuses.

Les disquettes 3 pouces $\frac{1}{4}$ contenant 1 Moctet utilisées pour distribuer des logiciels il y a 15 ans semblent ridicules devant les 24 Goctets d'un disque Blu Ray.

Lorsque les principaux langages ont été conçus, il y a 30 ans, les tailles mémoires se comptaient en Kilo Octets, et les optimisations mises en place dans les compilateurs étaient en relations avec cette taille. En schématisant un peu on pourrait mettre en relation les optimisations à effectuer en fonction de la taille des données.

La principale opération à effectuer sur un ensemble de donnée est le parcours que l'on décrit en général par un code du genre :

```
for (i = 0; i < N, i++)
{
   /* Traitement de la donnée Nro i */
}
```

En fonction de l'ordre de grandeur de `N` différentes stratégies de compilations pourront être mises en œuvre :

N = 10 Le nombre d'itération est petit : on pourra
- déplier complètement la boucle de manière à pouvoir optimiser tout le code du corps de la boucle
- utiliser de la micro-optimisation, calculer les dépendances, utiliser des allocations de registres complexes
- utiliser la planification des instructions (scheduling) qui sera d'autant plus efficace que le code déplié sera grand.
- utiliser des préchargements pour optimiser l'accès aux données via les caches.

Les exemples de ce type de code abondent dans le domaine du multimédia, traitement de vecteurs, couleurs sont exprimés par de petites boucles.

N = 10^2 à 10^5 pour ce type de dimension il n'est pas souhaitable de tout dérouler, mais on peut
- dérouler et/ou restructuration les boucles, de façon à trouver un facteur de déroulage utilisant au mieux les ressources disponibles.
- pré-charger des données de façon à ce que le code ne soit pas ralentit par des problèmes d'accès à la mémoire

N = 10^6 à 10^9 Pour ces dimensions encore plus grande on peut mettre en oeuvre les optimisations précédentes mais également tenter d'exploiter le multithreading. Il permettra d'exploiter les architectures multi-thread ou multiprocesseur à mémoire partagées.

Les exemples typique de ces dimensions sont les flux multimédia.

N = 10^{10} et au delà Pour ces dimensions les outils à utiliser seront plutôt de la parallélisation haut niveau (MPI, grid, ...) qui fonctionneront sur des grappes d'ordinateurs.

L'enseignement de la programmation est, entre autre, basé sur la réutilisation, on enseigne donc à écrire des codes les plus génériques possibles de façon à pouvoir le réutiliser le plus fréquemment possible.

Mais que faire au moment de la compilation statique lorsque la valeur de N n'est pas connue ? Quelle stratégie, technologie utiliser ? Est-ce une problématique liée à l'architecture, à la compilation aux systèmes d'exploitation ?

1.5.2 La variabilité des données

L'augmentation des capacités mémoire a aussi permis d'élaborer des programmes de plus en plus complexes. On n'aurai pas pu imaginer il y a 30 ans que l'on pourrait utiliser un ordinateur pour visionner un film.

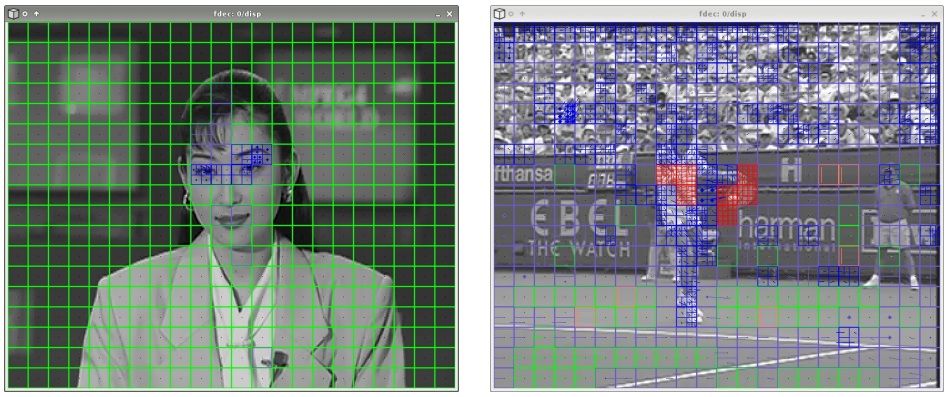


FIG. 1.7 – Comportement d'un compresseur vidéo en fonction de la scène. (Blocs verts inchangés : pas de calcul pour le rafraîchissement, blocs bleus : nécessité de déplacer des blocs, blocs rouge : nouveaux blocs)

Cette complexité s'accompagne d'une grande variabilité des performances en fonction des jeux de données. Par exemple la compression ou la visualisation d'un flux vidéo aura un comportement complètement différent en fonction du type de scène.

Par exemple une scène dans laquelle deux personnages s'embrassent sur le pont d'un bateau ne demandera pas beaucoup de ressources à la compression, car le fond de l'image (le bateau) reste fixe et n'a pas besoin d'être ré-encodé. De la même façon à la visualisation, les blocs constituant le fond de l'image n'auront pas besoin d'être modifié.

Mais si le film montre ensuite la mer agitée par une houle, la situation change totalement, la compression va se complexifier pour rechercher des blocs similaires (qu'elle a peu de chance de trouver) et la visualisation aura à reconstituer tous les blocs d'une image. C'est ce que j'appelle l'effet "Titanic".

Un même film en fonction de ses différentes scènes provoquera un comportement complètement différent du processus de compression ou de décompression.

La figure 1.7 illustre ce problème sur 2 vidéos issues des jeux de données du groupe MPEG. La vidéo de gauche est un extrait d'un journal télévisé. On voit sur l'image que le compresseur n'a détecté aucune zone mobile excepté les yeux. Lors de la visualisation de cette vidéo, le dé-compresseur conservera la majorité de l'image (tous les blocs verts), et n'aura qu'un peu de déplacement de données a faire dans les zones bleues. La vidéo de droite est une scène d'un match de tennis. On constate que le nombre de zone verte est très faible qu'une grande quantité de zones bleues sont à déplacer et que la partie rouge est nouvelle.

Il est donc illusoire de tenter de prévoir au moment de la compilation statique le comportement le plus probable afin de l'optimiser car il ne dépend que des données d'entrée qui ne peuvent pas être simulées par quelques jeux d'essais car leur variabilité est trop grande. Un processeur de très faible capacité pourra retransmettre correctement une scène statique mais sera incapable de rendre une vidéo avec beaucoup de changements.

1.6 Les langages de programmation

Les langages de programmation ont évolués suivant plusieurs directions. Il existe bien sûr la classique dichotomie langage compilés / langages interprétés, mais les choses sont un peu plus complexes.

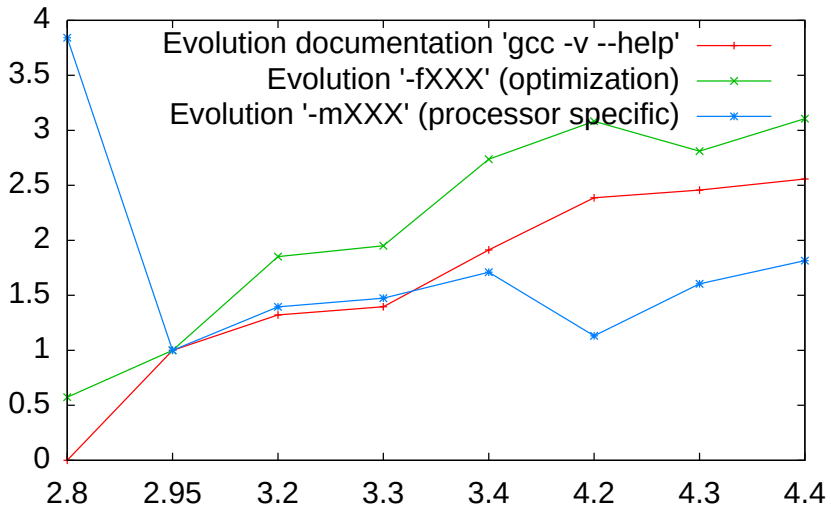


FIG. 1.8 – Évolution du nombre d'options de compilation (-f et -m) de GCC en fonction de son numéro majeur de version (base 1 pour la version 2.95)

Passons en revue les stratégies de compilation que l'on peut mettre en oeuvre pour 3 principales catégories de langage.

1.6.1 Langage C ou Fortran

L'ancienneté du langage C permet d'utiliser différentes approches pour la recherche de performances :

Il est possible de développer intégralement en C, la recherche de performances cherchera alors à utiliser au mieux le compilateur disponible pour le processeur cible.

L'utilisation du compilateur

Les compilateurs ont de nombreux paramètres permettant d'optimiser la production d'un code pour un processeur spécifique. La figure 1.8 représente l'évolution du nombre de ces options au cours du temps en fonction du numéro de version.

L'axe des X représente le numéro de version majeur de GCC. La courbe rouge représente l'évolution du volume de documentation produite par la commande (`gcc -v -help`) qui indique toutes les options disponibles (Base 1 sur la version 2.95, la version 2.8 n'en dispose pas). Les courbes vertes et bleues représentent l'évolution des options du style `-fXXX` qui représente l'activation d'une option d'optimisation et du style `-mXXX` qui représente l'activation d'une option spécifique à une architecture[6].

Cette figure permet de voir qu'entre la version 2.95 et la 4.4 le nombre d'optimisation a plus que triplé et le nombre options pour les architectures a augmenté de 50%. (L'infléchissement pour la version 4.2 s'explique par une réorganisation du code de GCC dans la branche 4).

Cette augmentation d'options disponibles entraîne une explosion combinatoire des possibilités d'optimisations, qui ne sont malheureusement pas exploitables facilement. Le programmeur choisit en général de sélectionner globalement les options

[6]La version 2.8 utilisait une syntaxe différente pour les options -m qui explique cette divergence

de compilation pour l'ensemble de ses sources car les outils de compilation comme `make` ou `gmake` permettent de les sélectionner globalement.

On peut noter que différentes options de compilations `-O1`, `-O2` et `-O3` activent (dans la version 4.2.1) respectivement 18 optimisations différentes pour `-O1`, 45 optimisations pour `-O2` et 48 pour `-O3`.

Le C comme langage intermédiaire

Le C ayant été conçu comme un langage pour assurer la portabilité est finalement assez (trop ?) proche de l'architecture. Il est donc parfois utilisé comme langage de bas niveau pour des DSL.

FFTW3 [11] et SPIRAL [24] en sont des exemples. FFTW3 est un générateur de librairies de DFT. Un méta programme en objective CAML permet de générer un ensemble de routine spécialisées qui sont ensuite évaluée sur la plateforme cible. Au run-time, un planificateur effectuera une décomposition du problème demandé par l'utilisateur en fonction des routines produisant le meilleurs résultat.

SPIRAL a un fonctionnement similaire bien qu'il soit capable de générer du code pour un spectre plus large de fonctions mathématiques.

Dans l'exemple de FFTW3, la version 3.1.2 portée pour le système FreeBSD est composée de 503 fichiers en langage C dont la plupart ont été générés automatiquement. Les seules options de compilations utilisées pour tous les fichiers sont les options :

-fno-strict-aliasing qui permet des optimisations sur les données de types équivalent

-O2 qui active 45 séquences d'optimisations

-ffast-math qui réduit les règles d'arithmétiques IEEE, permettant par la même occasion des optimisation plus agressives

-fomit-frame-pointer qui permet d'accélérer les appels de fonctions.

Les fonctions générées ont pourtant des caractéristiques différentes comme les tailles des fonctions, de boucles et des tailles des données manipulées qui permettraient une utilisation plus fine des fonctions d'optimisation du compilateur. Elles ne sont pas utilisées dans la pratique, l'optimisation étant focalisée sur la génération de code C et la sélection des meilleurs candidats.

L'utilisation des librairies

Les librairies permettent un développement rapide d'applications en isolant des fonctionnalités spécialisées dans des parties différentes du codes.

C'est un mode de développement efficace d'un point de vue génie logiciel car il permet de favoriser la réutilisation de code et également l'occupation mémoire pour un système multiprogrammé. Par exemple la librairie C `libc.so` qui est utilisée par toutes les applications ne sera chargée qu'une seule fois en mémoire physique, au lieu d'être dupliquée pour toutes les applications.

L'utilisation de librairies permet également d'isoler les routines nécessitant une optimisation spécifique. J'ai donné l'exemple de la libraire FFTW3 et de SPIRAL, mais ATLAS [27] qui est une librairie d'algèbre linéaire isole également les fonctions réalisant du calcul intensif dans des fonctions.

L'inconvénient de cette utilisation est qu'il n'est pas possible d'entrelacer le code provenant des librairies avec le code de l'application. Cela a pour conséquence :

- d'obliger le programmeur à n'utiliser que des librairies dont le "coût" d'utilisation est suffisant pour que le sur-coût d'appel ne soit pas trop fort.

Par exemple l'utilisation d'un appel à une librairie dans la boucle la plus interne d'une application doit avoir un temps d'utilisation suffisant pour recouvrir le temps de changement de contexte. Sinon il eut mieux valu reprogrammer la fonction dans la boucle pour économiser le temps d'appel.

– le corollaire est qu'un compilateur ne peut pas entrelacer le code d'une librairie avec le code du programme à compiler.
  À ma connaissance seul le compilateur C de Sun permet un tel entrelacement pour l'utilisation des instructions multimédia VIS de son processeur SPARC. Le code multimédia est inséré dans le flot d'instructions en train d'être compilé et peut être entrelacé et planifié avec le code de l'application.

L'usage ou la conception de librairies est orthogonal à l'utilisation d'un compilateur, il est difficile de bénéficier des avantages d'une librairie (code optimisé, spécialisé) dans un compilateur optimisant.

L'interface avec le système

Si la production de code était relativement indépendante du système d'exploitation pour lequel on produit le code, les évolutions récentes ont réduit cette indépendance.

Historiquement un compilateur a une dépendance avec le système d'exploitation pour lequel il produit le code pour les raisons suivantes :

– le chargement du code à l'exécution, le système est chargé de créer le processus, d'allouer la mémoire, de charger le code en mémoire et de charger les librairies nécessaires
– l'ABI (Application Binary Interface) utilisée qui dépend du processeur mais le changement de contexte effectué par le système joue un rôle.
– la libraire C utilisée qui est une dépendance assez forte. Par exemple un code contenant des copies de structures comme le suivant :

```
typedef struct{int a[1000]; } mastruct;

mastruct copy(mastruct a)
{
  mastruct b;
  b = a;
  b.a[1]++;
  return b;
}
```

  utilisera pour recopier la structure `a` dans la structure `b` la fonction `memcopy()` de la librairie C qui est supposées être optimisée pour cette plateforme. Les compilateurs qui produisent du code pour une plateforme embarquée, i.e. sans système d'exploitation, ne peuvent pas faire la même hypothèse.
– la résolution des adresses des appels de fonction : les binaires des applications ne contiennent pas les adresses des fonctions systèmes. Ces adresses sont calculées et remplacées dans le programme binaire lors du premier appel à cette fonction. Cela a pour objectif de réduire le temps de démarrage d'une application.
  Cela implique qu'un code binaire se modifie au cours de son exécution.

Plus récemment les compilateurs optimisant ayant augmenté leurs fonctionnalités ont par la même occasion augmenté les dépendances avec le système d'exploitation spécialement pour les applications parallèles :

– la gestion des liens entre processeurs et processus/threads. Les systèmes multiprocesseurs et/ou multithreads permettent aux applications de choisir les processeurs sur lesquels les threads peuvent fonctionner. Cela a un impact sur les caches données et instructions.

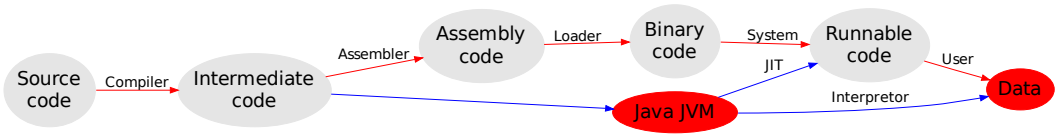


FIG. 1.9 – La chaîne de compilation pour Java

Nombre de fichiers	Extension	Type de fichier
139	1	page de manuel
267	gmk	partie de Makefile
315	idt	
335	properties	Messages d'erreurs
401	html	Documentation
537	gif	Image
744	hpp	Source C++
954	cpp	Sources C++
1213	h	Entete C
1303	c	Souces C
11007	java	Sources Java

TAB. 1.1 – Statistiques de type de fichiers pour le JDK 1.5

– la gestion des pages mémoire pour les applications parallèles utilisant une mémoire NUMA. La localité des données est importante pour le chargement des données pour un processeur, dans les machines NUMA c'est encore plus important.

Le compilateur est de plus en plus lié aux fonctionnalités avancé des machines parallèles pour la gestion des processeurs, des threads et l'accès à la mémoire.

1.6.2 Langage Java

On a coutume de dire que Java [17] est un langage hybride, compilé en bytecode puis soit un interprété soit compilé au vol par un JIT. Mais le fonctionnement réel est un peu plus complexe.

La distribution source de Java par Sun Microsystems est composée de fichiers sources de différents langages :

C++ principalement pour le compilateur au vol Hotspot et des librairies graphique du package `java.awt`. Le compilateur peut générer du code natif pour les processeurs des familles (`i486`, `amd64`, `ia64` et `sparc`)

C principalement pour des packages qui nécessitent une implémentation rapide (traitement d'image, de son, de fenêtrage, calcul en multiprécision, etc)

java pour les packages (librairies) du JDK.

Hotspot est un compilateur au vol capable d'effectuer un compilation itérative. Il commence par interpréter le byte code java puis compile en code natif les zones les plus souvents exécutées (hotspot) et applique des optimisations de plus en plus complexes si en fonction des zones les plus souvents exécutées[18].

L'inconvénient majeur de cette technologie est le coût de la détection qui est effectuée au run-time. Les articles décrivant cette technologie indiquent que les meilleurs résultats (équivalents aux meilleurs compilateurs C++) ne sont obtenus qu'après un temps asymptotique au run-time ce qui est particulièrement gênant pour des applications dont le comportement est dynamique.

La fréquence des principaux type de fichiers pour le jdk1.5, résumée dans le tableau 1.1. On peut voir qu'il contient des fichiers écrits en java pour les librairies (packages java), mais également une partie est écrite en C ou en C++.

L'exécution d'un programme java passe donc

- soit par l'interprétation de byte-code dont les performances sont médiocres, car elle nécessite d'émuler chaque instruction séquentiellement, octet par octet (le byte code java est codé sur un octet).
- soit par la compilation et l'optimisation en code natif par le compilateur HotSpot, qui détecte au fur et à mesure de l'exécution d'une application les "hotspots" sur lesquels le compilateur doit se focaliser.
  Dans ce cas on arrive a de bonnes performances mais sur des temps d'exécution extrêmement longs [18], car la détection des "hotspots" et les temps de compilation sont prohibitifs.
- soit par l'exécution de code de librairies qui utilisent du code natif écrit en C ou en C++ pour les parties les plus critiques comme la gestion du fenêtrage ou la gestion (compression/décompression) des images en mémoire.
- soit enfin l'exécution de code natif préparé à part et chargé dynamiquement via l'interface JNI (Java Native Interface). Cette interface permet de charger lors de l'exécution des librairies de code natif et de les appeler à partir de Java.

La machine virtuelle Java utilise des modes d'exécution qui entrelacent exécution native de code compilé statiquement, exécution native de code compilé dynamiquement, chargement dynamique de code natif et interprétation.

L'optimisation de code dans ce cadre est un exercice complexe

- soit parce que le temps d'optimisation annule tout espoir de gain pour un code dont la durée de vie est rapide
- soit parce que l'entrelacement de code natif optimisé et de code interprété ne permet pas d'avoir de bonne performances soutenues durant toute la durée d'exécution du code.

Java malgré son succès auprès des développeurs auxquels il fournit un environnement de développement de haut niveau, n'est pas un langage pour lequel l'optimisation de code classique est aisée. D'une part à cause de son haut niveau d'abstraction et d'autre part par la complexité du modèle d'exécution.

1.6.3 Langage PHP

Le langage PHP [26] est un langage purement interprété utilisé principalement pour le développement de sites internet interactifs. C'est un langage de type "glue" c'est à dire dont l'objectif est de rassembler tous les appels systèmes permettant de construire une page internet dynamique (appel aux services d'identification, base de données, système de palment en ligne, etc).

Néanmoins si un langage de ce type peut suffire pour une application fonctionnant en mono-poste, le contexte client-serveur des applications internet fait que l'exécution de multiples instances de ce code peut devenir critique pour un site ayant de nombreux clients. Il s'agira alors non seulement d'optimiser l'exécution d'une instance de code mais également de multiples instances simultanées.

La distribution 5.2.5 du langage PHP contient 7171 fichiers. La distribution des types de fichiers utilisé est résumée dans le tableau 1.2

On peut voir sur ce tableau que l'interpréteur est principalement écrit en C. Le fonctionnement d'une application mélange :

l'interpréteur le programme est purement interprété ce qui conduit à une exécution particulièrement inefficace.

Nombre de fichiers	Extension	Type de fichier
75	xml	Documentations et tests
82	w32	Fichiers de configuration pour windows
93	php	Programmes PHP
111	m4	Programmes de réécriture de texte (Configuration Unix)
147	inc	Partie de programme PHP
560	h	En-tête programmes C
807	c	Programmes C
4473	phpt	Tests unitaire de validation de PHP

TAB. 1.2 – Répartition de l'extension des fichiers dans la distribution PHP 5.2.5

appel de fonctions natives PHP est un langage qui utilise extensivement des librairies externes (glue language) pour les fonctionnalités spécialisées (accès aux base de données, manipulation XML, etc).

l'appel de services distants requêtes à une base de données, services distant, etc.

L'exécution d'un programme PHP entrelacera l'utilisation de l'interpréteur interne et des appels à des librairies spécialisés compilées de façon externe à l'interpréteur PHP. Dans ce contexte il n'existe pas de liens entre l'interpréteur et les librairies utilisées.

La encore l'optimisation de code classique n'est pas possible car de nombreux programmes et services collaborent à l'exécution d'un programme global. Seule une exécution run-time pourrait tirer partie des données à l'exécution.

1.6.4 Openoffice.org

Openoffice.org [1] est une suite bureautique dont les sources sont publiques puisque c'est un logiciel libre dont le développement est organisé par Sun Microsystems.

Openoffice.org n'est pas un langage de programmation mais cet exemple me servira à montrer que les applications modernes pour lesquelles les compilateurs sont conçus sont, par essence, difficiles à optimiser parce qu'ils sont un assemblage complexe de briques logicielles très variées.

La version 2.4.1 d'openoffice.org contient 62646 fichiers pour une taille totale des sources de 1,7 Giga octets. Sans compter les 33 librairies open-source utilisées par le projet (Accès aux scanners, librairies pour les images, Beanshell permettant d'interpréter des programmes source java, etc).

On peut voir sur le tableau 1.3 que les fichiers sources les plus fréquents sont des ... images ! En effet ce sont les différentes images dans les différentes langues. L'icône permettant de mettre en gras représente un "G" en français et un "B" en anglais (25 langages sont supportées).

Les autres fichiers correspondent a des codes sources en C++ majoritairement mais aussi en Java, en C et en python.

Cette agrégation de différents langages s'explique par le fonctionnement du projet Openoffice.org qui a rassemblé un grand nombre de projets open-source. La encore l'optimisation de code ne peut pas être vue de façon globale car la complexité du projet est trop grande.

Nombre de fichiers	Extension	Type de fichier
995	h	Fichier d'entête C
1140	xlb	
1158	xml	
1191	xba	Macro
1603	ott	Template Texte Openoffice
2243	xhp	Texte d'aide
2480	mk	Partie de Makefile
3639	java	Programmes Java
3958	idl	Interface Definition language
10015	cxx	Programmes sources C++
10392	hxx	En-tête C++
11706	png	Fichiers d'image

TAB. 1.3 – Répartition de l'extension des fichiers dans la distribution Openoffice.org

1.7 L'architecture des systèmes

En parallèle à l'évolution des processeurs et des langages de programmation les systèmes d'exploitation ont beaucoup évolués. D'une architecture monolithique et mono-processeurs ils sont passés à une architecture modulaire et permettant d'exploiter les architectures multi-threadées, multiprocesseurs avec mémoire partagées ou non.

1.7.1 L'accès à la mémoire

Le chemin d'accès à la mémoire n'a pas cessé de se complexifier.

Le mot clef `register` existe dans le langage C depuis sa création. Il permet au programmeur de demander au compilateur de placer explicitement une variable dans un registre du processeur.

Dans les compilateurs modernes, ce mot clef n'est pas pris en compte par l'allocateur de registre car il est plus efficace que ce que le programmeur peut donner comme indication. Néanmoins tous les programmeurs C savent qu'il vaut mieux accéder aux données d'un tableau par ligne plutôt que par colonne afin d'augmenter le taux de hit d'un cache.

1.7.2 La gestion d'architectures hétérogènes

Nous avons vu qu'il existe des processeur avec de multiples jeux d'instructions pour lesquels la génération de code binaire est complexe.

Le problème est encore plus complexe lorsqu'il s'agit de générer du code pour des architectures avec des accélérateurs graphiques (GPU). Dans ce cas non seulement les jeux d'instructions sont différents mais la mémoire est également séparée et les modèles de programmations sont différents. Les GPU n'ont pas de système d'exploitation ce qui nécessite d'embarquer dans le code toutes les librairies dont le programme aura besoin lors de son exécution.

Les chapitres suivants montreront différents aspects des recherche que j'ai menées pour transformer les différentes façons de générer du code binaire.

Chapitre 2

Réduction de l'empreinte mémoire

2.1 Introduction

L'objet de cette recherche menée a l'IRISA de Rennes en collaboration avec Melle Karine Heydemann était de montrer qu'il est possible d'entrelacer du code natif généré par un compilateur optimisant, et un bytecode interprété et compact dans le but de réduire l'empreinte globale de l'application tout en ne perdant que peu en performances dans les parties sensibles.

Bien que l'évolution des machines n'aille pas vers la réduction des capacités mémoire, la taille mémoire d'une application est un critère important qui n'est pas pris en compte par les compilateurs actuels.

Les systèmes embarqués ont des contraintes importantes en capacités mémoire, un téléphone portable, un lecteur de DVD ou un décodeur mp3 sont restreints

- d'une part par la quantité de mémoire embarquée qui impacte directement le prix du produit final
- mais également par les problèmes de consommations électrique induits qui nécessitent un dimensionement des batteries de la machine.

Dans ce projet nous avons transformé une chaîne de compilation (cf schéma page 7 du document suivant) dans laquelle nous

1. sélectionnons après profilage les régions qui sont candidates pour la compression (ou plus exactement la conversion en code compact).
2. les zones sélectionnées sont analysées afin de générer un jeu d'instruction compact spécialisé pour cette application.

   L'analyse permet de détecter quels sont les suites d'instructions les plus fréquentes. Les séquences les plus fréquentes sont alors synthétisée en une seule.
3. les zones sélectionnées sont ensuite réécrites du code natif vers le code compressé.

   Le code natif est un processeur RISC 3 adresses, le code compressé est l'équivalent d'un byte code 0 adresses.

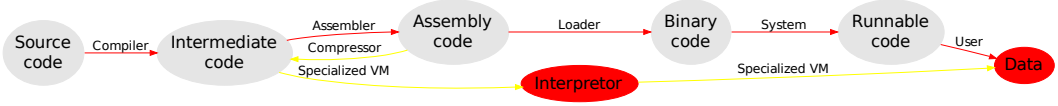


FIG. 2.1 – La chaîne de compilation pour la compression de code

4. une machine virtuelle pour le code compressé est ensuite ajouté à l'ensemble

L'exécution alternera entre les zones dans lesquelles le code natif est exécuté directement par le processeur et les zones compressées qui seront interprétées par la machine virtuelle dont le jeu d'instruction est spécifique de l'application.

Il faut noter que toutes les transformations sont effectuées au niveau source assembleur ce qui permet :

- d'effectuer des transformations de façon indépendante du compilateur.
- d'effectuer des transformations sur un code déjà compilé, dont on n'a pas les sources, bien que nous ne l'ayons pas réalisé durant cette expérimentation.

2.2 Résultats obtenus

Nous avons obtenus une chaîne de compilation fonctionnelle permettant de compiler des codes complexes comme X264, ghostscript ou MPEG4.

Les résultats chiffrés sont indiqués dans l'article suivant. Ils ne sont pas impressionnants comme peuvent l'être le résultat de certaines optimisations, mais de façon plus importante cette expérimentation nous a permis de montrer :

- qu'il est facilement possible de prendre en compte des paramètres comme un compromis entre la taille d'un code et sa performance alors qu'habituellement l'objectif est uniquement un objectif de performance et d'efficacité.
- que la génération d'une isa compacte spécialisée pour une application permet des taux de compression intéressants
- que l'optimisation hybride réalisée est une optimisation globale, elle prend en compte la totalité de l'application alors que la plupart des optimisations (pour la performance ou pour la taille) sont des optimisations locales.

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

A compression scheme for code size versus performance trade-offs

Karine Heydemann * and Henri Pierre Charles ** and Francois Bodin ***

Thème 4 — Simulation et optimisation
de systèmes complexes
Projets Caps

Publication interne n ° 1574 — Novembre 2003 — 16 pages

Abstract: In this paper we propose a new software compression technique for embedded systems. This technique aims at helping embedded system developers to find good trade-offs between applications code size and performance. The proposed compression scheme is machine independent and compatible with multi-task systems. In this paper we focus on the efficiency of the compression scheme and we analyze the main factors influencing the code compression mechanism. We show that significant compression (13% to 33% reduction in code size) can be achieved while not slowing down computation intensive part of the code.

Key-words: compression, trade-off, emulator, superinstruction

(Résumé : tsvp)

* karine.heydemann@irisa.fr
** hpc@irisa.fr
*** bodin@irisa.fr

Centre National de la Recherche Scientifique
(UPRESSA 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Schéma de compression pour la recherche de compromis entre la taille d'une application et sa performance

Résumé : Nous proposons dans cet article une nouvelle technique de compression logicielle pour les systèmes enfouis. Cette technique permet de réduire la place mémoire nécessaire à une application. Notre shéma de compression vise à ne sélectionner que des portions de code peu exécutées grâce à des profils d'exécution afin de limiter le surcoût en temps d'exécution dû à l'émulation du code compressé. Notre shéma à granularité fine nous permet de compresser de très petites régions. Nous pouvons sélectionner des portions de code de quelques blocs de base peu couteuses en temps d'execution même dans une fonction fréquemment exécutée. La dégradation des performances peut ainsi être limitée sans diminuer la quantité de code candidate à la compression. Notre technique étant globale à l'application, elle offre un moyen de trouver de bons compromis entre la taille du code exécutable et son temps d'exécution. Nous montrons que nous obtenons un taux de compression significatif (réduction de 13% à 33% du code).

Mots clés : Compression de code, compromis, émulateur

1 Introduction

Designing an embedded system is searching for a trade-off between hardware and software. Developers must achieve fast design taking into account various constraints such as memory space, power consumption and application speed. Memory space is a strong constraint as it directly impacts on the cost and the functionalities of the systems. One would like to minimize the amount of memory space allocated to programs to allow more applications to fit in the device and to reduce the amount of memory chips. Finding a global tradeoff between code size and performance consists in spending code size where it provides important speedup while reducing the code space of infrequently executed code section. Figure 1 summarizes our approach to code compression.

Code compression has been extensively studied and the fundamental techniques are well known (Huffman 1952) (Ziv & Lempel 1978). However, considering code optimizations, especially architecture dependent one, and compression in the same framework generates important constraints on the compression scheme run-time. Infrequently executed code regions are spread all over the code and corresponds to small sets of basic blocks (Hoogerbrugge and al. 1999). As a consequence, granularity of the compression scheme is a major issue in the context of code size/speed tradeoff. Granularity of a compression scheme is characterized by two main properties: the minimal amount of contiguous instructions that can be compressed with benefit and the cost of the runtime system to switch from compressed to uncompressed code regions.

In this paper we propose a new software, compiler driven, compression scheme for embedded applications. The compression scheme can be used in multi-task systems, does not use RAM memories to decompress the code and is machine independent. It has been designed for a small compression granularity. The method achieves a high compression ratio [1]. The compressed code (denoted bytecode in the remainder of the paper) is typically half the size of the original code. Considering the size of the bytecode and of the non compressed code we reduce the instruction memory space from 13% up to 33%. The target processors considered are 32/64 bit RISC processors.

The first part of the paper is devoted to the description of the compression scheme. The second part presents the experiments and analyses the various factors influencing the compression ratio. The experiments have been performed for a Sparc processor on large applications such as MPEG4. The related works are presented in Section 4.

2 Compression Scheme

The compressing scheme is based on the following program property : given one program, it only requires a small subset of the target processor instructions. As a consequence, it is possible to recode the used instruction subset to a new instructions set architecture (ISA). This ISA is then specific to one application code. We use a predefined ISA format to achieve this

[1] We define the compression ratio as $\frac{total\ final\ code\ size}{origin\ code\ size}$ and the compression rate as $100 - compression\ ratio$

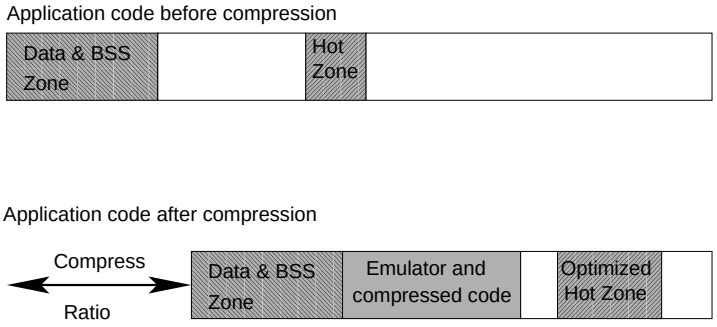


Figure 1: Partial Code Compression.

new ISA design. We instantiate the predefined ISA by type field dictionary encoding, i.e each field type is encoded with a different dictionary. We get an ISA composed of generic instructions (target processor independent), application specific instructions and superinstructions. Superinstructions (Proebsting 1995) are sequence of instructions frequently appearing in the compressed code (typically `muladd` example). For each application a specialized emulator is generated at compile time and linked with the application.

The emulator is based on a RISC like bytecode format specifically instantiated according to the instruction mix found in the code regions of the application that are selected for compression. For instance, if no floating point operations are used in the code regions, no bytecode implements these operations. Construction of the bytecode is described in Section 2.5. The ISA is register-based in order to reduce memory traffic when switching from compressed code to not compressed regions and vice versa. This allows a faster emulation(Brocard & Sugalski 2001). The compression can be performed at basic block or function level. Granularity of the compressible regions is particularly important as it directly impacts on the compression ratio and the code regions that can be selected for compression. The stubs needed to switch from compressed to/from uncompressed code regions must be limited to a few instructions if ones want to compress, inside hot functions the various basic blocks infrequently executed.

The emulated code runs slower but this does not necessarily impact greatly on the global application performance as it is not often executed. The application developer sets the performance constraints. This constraint is then used to select the infrequently executed code regions as candidates for compression. The faster the emulator, the more code regions can be selected, the more compression is achieved. For instance, if 10% of the code corresponds to 90% of the execution time, compressing the 90% of the remaining code, at the expense of a 10 fold slowdown, leads to a total execution time of the compressed code at most twice of the one of the original code. If a large reduction of the memory space is achieved this may

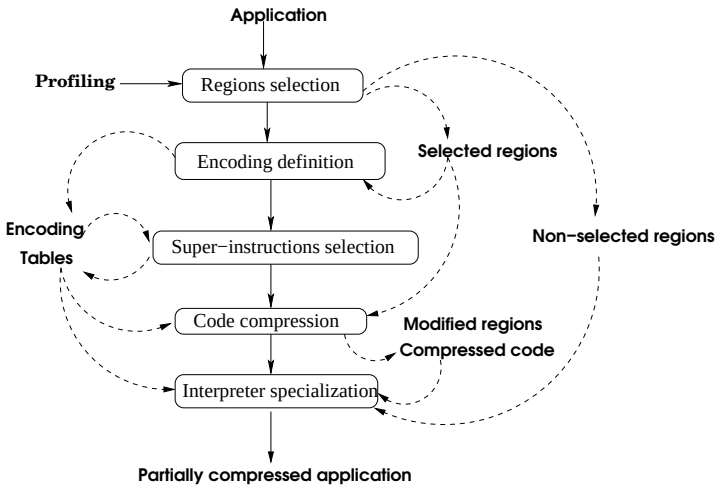


Figure 2: Global compression process

finally be an acceptable tradeoff (for instance, if it gives space for a new application on the system or in case of a strong memory constraint).

The following sections detail the steps to achieve the code compression. First we give an overview of the bytecode used. Section 2.2 briefly describes the run-time model used. Section 2.4 shows how the instruction encoding is performed. Section 2.5 details the compression algorithm steps after having introduce a short example in Section Section 2.3. Finally, in Section 2.6 we describe the interpreter/emulator specialization that occurs after the definition of the bytecode. Figure 2 summarizes the different steps of our compression process.

2.1 The Emulated Computer and Bytecode Format

The compressed code is implemented by an emulated bytecode that is similar to a 2-address RISC ISA. A bytecode can be coded on 1, 2 3 or 4 bytes. The maximum number of operation codes (denoted opcode in the remainder) is 64. The emulated ISA is limited to 32 registers (considering integer and floating point registers)[2]. The ISA instructions are not typed but have typed addressing mode. For instance, the floating point `add` and the integer `add` shares the same opcode, whereas the `add_imm` opcode (adding an immediate to a register) is different from the `add` adding registers values. According to the target application instruction mix, the ISA is built from generic instructions, specific ones and

[2]This restriction can be alleviated by applying it to each region : a candidate code region for compression should not used more than 32 registers even if many more are used in the whole application

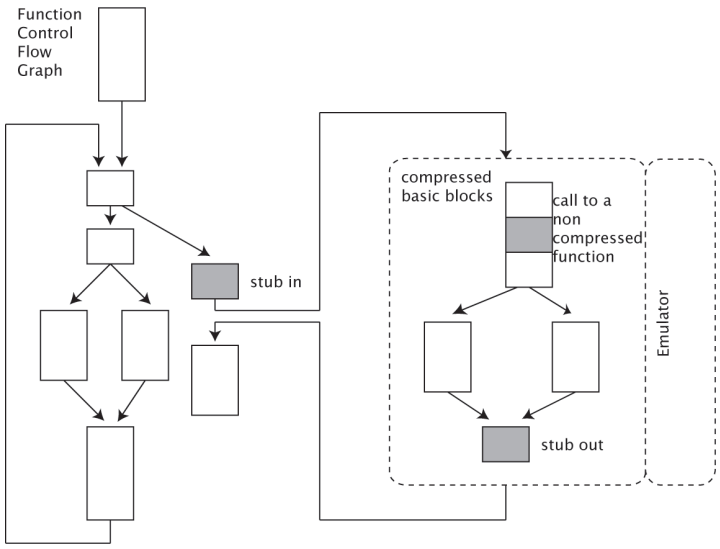


Figure 3: Runtime model.

superinstructions. Generic instructions are machine independent (for instance load/store data) while specific ones correspond to operations found on the target machine ISA. The number of superinstructions is limited by available opcodes once the generic instructions and the specific ones have been determined. Superinstructions are particularly interesting since they increase the compression ratio and allow in the same time the emulator to run faster (Proebsting 1995).

Thanks to the register-based bytecode, it is easy to compare the behavior of the compressed and non compressed code to simplify the debugging of systems. At the end of each basic block the contents of the register set and the memory can be compared.

2.2 Run Time Model

The compressed region is replaced by a small stub. At runtime, when the region is executed, the stub call the emulator. Figure 3 illustrates the stub insertion in the code. Emulated code can call non compressed code (for instance the case for all external libraries -I/O, math, etc...) which executes at full speed in native mode.

The granularity of the compression scheme is defendant of the size of the **stub** needed to switch from compress to uncompress region (and vice-versa). Furthermore, for small regions, the execution cost of the stub can slowdown a lot the execution of the emulated code. Thanks to the register-based emulated ISA and shared data space between emulated

Source code	Assembly code 17 instructions = 68 bytes	Compressed code 11 instructions = 22 bytes

`char msg[] = "Hello world: %d\n";`
`extern int add (int, int);`
`int main(int argc, char * argv[])`
`{`
`  printf(msg, add(37, 5));`
`  return 7;`
`}`
`int add(int a, int b)`
`{`
`  int tmp = a+b;`
`  printf("Not interpreted code: %d\n",`
`  tmp);`
`  return tmp;`
`} /* add */`
	`main:`
	`  !#PROLOGUE# 0`
	`  save %sp, -112, %sp`
	`  !#PROLOGUE# 1`
	`  st  %i0, [%fp+68]`
	`  st  %i1, [%fp+72]`
	`  mov  37, %o0`
	`  mov  5, %o1`
	`  call add, 0`
	`  nop`
	`  mov  %o0, %o1`
	`  sethi %hi(msg), %o2`
	`  or  %o2, %lo(msg), %o0`
	`  call  printf, 0`
	`  nop`
	`  mov  7, %i0`
	`  b  .LL2`
	`  nop.`
	`LL2:`
	`  ret`
	`  restore`
		`unsigned char TableByteCode [] = {`
		`/* 00 STFP68  */ 0x07, 0x02,`
		`/* 02 STFP72  */ 0x08, 0x12,`
		`/* 04 MOVcte37 */ 0x09, 0x03,`
		`/* 06 MOVcte5  */ 0x0a, 0x04,`
		`/* 08 CALL  */ 0x02, 0x00,`
		`/* 0a MOVcte5  */ 0x03, 0x34,`
		`/* 0c LD  */ 0x04, 0x1a, 0x04,`
		`/* 0f CALL  */ 0x02, 0x01,`
		`/* 11 MOVcte7  */ 0x0b, 0x00,`
		`/* 13 B  */ 0x05, 0x01,`
		`/* 15  .LL2  */`
		`/* 15 RET  */ 0x06`
		`};`

Table 1: Compression example.

and executed code, the stub is limited to a few register copies. There is no need to set-up a stack or other buffers.

Current size of the stub is 6 assembly instructions. Part of the stub is private to a compressed code region while the other part is factorized. If an original assembly instruction is encoded in average 20 bits (instead of 32), then the compression ratio of instruction is around 40% (see Section 3.2). Based on this ratio, the minimum code size providing compression benefit is 15 instructions, which is about two basic blocks.

The run-time model allows to use the compression scheme in multi-task environment. The emulator and the application code being one binary executable it is seen as one task from the OS. Furthermore, no decompression buffer to share or to duplicate between tasks is needed.

In the next section, we illustrate the compression on a simple program running on a Sparc processor used for the experiments.

2.3 Compression Example

Table 1 shows a code example which is used in the remaining of the paper. We assume that the `main` function is selected for compression and that the `add` function is not.

Registers	Constants	Jump addresses	Opcodes	Potential superinstructions
%i0; %i6; %i1; %o0; %o1;	68; 72; 37; 5; msg; 7;	add; printf;	STFP; MOVcte; CALL; MOV; LD; B; RET;	MOVcte;5;2 STFP;68;1 STFP;72;1 MOVcte;37;1 MOVcte;7;1

Table 2: Encoding tables

The first column shows the C code with 2 functions, the second column the assembly code generated with a C compiler for the `main` function. The last one gives the encoded instructions after our compression process with the corresponding opcode in comments. The code size is reduced from 68 bytes to 22 bytes.

The first four instructions in the last column are examples of specialized instructions. `STFP` are instructions to save register on the stack. Since the values 68 and 72 are used very often in all functions, the compression specializes these instructions and creates 2 new instructions `STFP68` and `STFP72`.

We can see the assembly instructions `sethi` followed by `or` that are replaced by the `LD` single instruction .

2.4 Encoding Definition

To instantiate the bytecode, the compression process globally analyzes all selected code regions to build encoding tables which are lately used as dictionaries for the operands (register, constants values, branch addresses), and the instructions. Theses tables are shown in Table 2 for the example given in Section 2.3. The first one contains the set of registers used in the application, the second one all the numeric constants and variable addresses. The third one gives the called addresses inside the application like `add`, or outside like `printf`. The fourth table contains the instructions and the last gives the potential superinstructions. In our small example, all the superinstructions are used once except the `MOVcte5` which is used 2 times.

We use a greedy algorithm to select the superinstructions that maximize the bytecode gain. The code of these superinstructions is automatically generated using the basic instructions contained in the emulator. In our example, all the superinstructions are selected accordingly to the remaining available opcodes. In this example each superinstruction saves one byte, the total gain is 5 byte. In the next section, we describe the compression process that makes use of the dictionaries for the bytecode encoding.

2.5 Compression Process

To achieve a machine independent compression scheme we have divided the compression process in multiple phases. Only first phase is target dependent. The code compression steps are the following:

From assembly code to 3 addresses : we first use a 3-addresses intermediate representation (IR) as a first architectural neutral program description. The parsing step from the assembly code to this IR deletes all unused instructions such as `nop` or context save/restore instructions. This is illustrated on the example in Table 1 : during the parsing the instructions `save` `restore` and `nop` are deleted.

Optimization: the previous representation is used to make a semantic upgrade phase (grouping of operations decomposed by the compiler). For example the SPARC instructions `sethi` followed by `or` which is used to load a constant address in a register, is replaced by a single `ld` instructions. In the example the semantic upgrade transforms the instructions `sethi %hi(msg) %o2` and `or %o2 %lo(msg) %o0` into `load msg %o0`.

This technique is also used to remove unnecessary operands, like for writing in a constant register. This kind of instruction exists because an hardware ISA is not as flexible as a software ISA. For instance SPARC instructions imposes 3 operands and so use a dummy one (register `%g0` whose value is 0) to encode a 2-operand instruction.

From 3 to 2 addresses: We apply a new transformation to get a more compact code with 2 addresses instructions. This step may add some new instructions. In our experiments we add around 1% more instructions which is at least three times less than the number of instructions combined or dropped at the previous steps.

Bytecode generation: the final coding of the bytecode is performed using the 2 addresses IR and the tables. A two passes algorithm is used to resolve relative branch label. Call values and constant values are coded with a variable size in order to optimize the size of the instructions.

For the example, the bytecode is generated from the tables (shown Table 2) and the instructions. The branch addresses are computed. For instance, the branch instruction at the address `0x0013` in the bytecode 1 need a one byte size relative jump.

Once the bytecode had been defined by the compression process, the emulator is built. This is presented in next section.

2.6 Emulator Construction

The previous steps create all the information needed by the emulator: the tables and the bytecode. The emulator is then compiled and only the needed instructions and superinstructions are kept for this application. This compilation process produces a library containing the bytecode, the tables and the emulator specialized to the given application.

Application	initial size (kB)	size (kB) without data	% code	bytecode size (kB)	final size(kB)	compression ratio	compression rate
H263	330	290.6	95%	149	192	0.67	33%
ghostscript	1645	1488.5	92.8%	781	1017	0.68	32%
MPEG4	1804	1677	88.4%	841.2	1118.5	0.66	33.3%

Table 3: Compression results without any compiler optimization.

Application	initial size (kB)	size (kB) without data	% code	bytecode size (kB)	final size(kB)	compression ratio	compression rate
H263	184.3	144.4	84%	75	119.8	0.83	17%
ghostscript	942.8	787.1	75%	397.5	685.6	0.87	13%
MPEG4	1048	921.5	70%	403.8	744.7	0.81	19%

Table 4: Compression results without O1 optimization enabled

Current implementation tests show that a simple version of the emulator is less than 60 times slower than the original code. However, there is room for many optimizations. Other works (Fraser & Proebsting 1995), (Ernst and al. 1997) on interpreters or virtual machines have reported slowdown factors from 5 to 20 times using optimizing techniques that can be applied to our interpretor.

The interpreter is about 15kB but compared to the size of large applications, this overhead is relatively low. Furthermore, parts of the emulation library can be shared between applications.

3 Experiments

We have implemented the compression scheme for a SPARC processor. The code is compiled using *gcc*(GCC). The interpreter is written in C language. For the experiment we compress as much as possible the code (i.e. no tradeoff is performed).

This section is organized as follow. First, we present, in Section 3.1, the results obtained using three large applications. In Section 3.2 we analyze the compression ratio according to the different features of the compression scheme.

3.1 Results

We have experimented our compression scheme using 3 large benchmarks `H263` (h263), `ghostscript` (Ghostscript) and `MPEG4` (MPEG4). We applied compression both on the non optimized code (-O0) and the optimized code with first level of optimizations (-O1). The -01 optimization level uses code size friendly transformations, whereas higher optimization levels may increase code size and should be limited to hot spots.

Application	Call Table	Constant Table	Register Table
H263	176	1170	32
MPEG4	2077	3832	32
ghostscript	1401	3788	32

Table 5: Number of entries in the encoding tables.

Tables 3 and Table 4 show the results for the three benchmarks. The second column is the executable total size. The third column provides the size of the code (Sparc instructions) in the executable. The fourth column gives the percentage of the code that has been compressed. Non compressed code are:

1. Functions using more than 32 registers (integer and floating point registers added)[3].
2. Functions containing less than 15 instructions. This is a very small amount of code.

Column 5 gives the code size after compression and column 6 the final code size including the interpreter and encoding tables. The seventh column is the compression ratio equal to $\frac{column\ 6}{column\ 3}$. Finally, column 8 gives the global compression rate, equal to $\frac{total\ code\ size}{original\ code\ size}$.

On non optimized code, the results show compression rate from 32% up to 33.3%. On optimized code the compression rate varies from 13% up to 19%. On the non optimized code, the ratio is higher than on optimized code. This is due to useless instructions in the non optimized code. We explain in details this difference in Section. 3.2.

The compression ratio is related to the size of the instructions mix of the code region to compress, to the nature of the most static frequent instructions and to the size of the encoding tables. If the most static frequent instructions in the original application is encoded with two bytes the compression rate will be higher than if they were encoded in three bytes. Moreover, the larger are the table, the more bits are used in encoding constants or call addresses. Table 5 expresses the number of entries of the different encoding tables to give an order of magnitude.

Remember that our interpreter is about 15kB without encoding tables which are the major overhead of our compression scheme. However, compared to the size of the application, this overhead is relatively low.

3.2 Compression Ratio Analysis

We analyze the way instructions are encoded. An assembly instruction can be dropped at parsing time (`nop`, useless `save` or `restore`), combined at the semantic upgrade step, or compressed in 1,2,3 or 4 bytecodes. Tables 6 and 7 show the distribution of the code according to the length of their compressed representation for the non optimized and optimized applications. 75% to 80% of instructions are encoded in 2 or 3 bytes, which explain the

[3]Current implementation does not apply this constraint to each region but to the whole application

Application	0 Byte	1Byte	2 Bytes	3 Bytes	4 Bytes	Average bytecode size
H263	16.7%	0.2%	39.9%	34.4%	8.7%	17.4 bits
ghostscript	17.4 %	0.8%	32.4%	36.7 %	12.7 %	18.1 bits
MPEG4	15.2%	0.32%	34.5%	42%	7.8%	18.1 bits

Table 6: Compression distribution without any compiler optimization.

Application	0 Byte	1Byte	2 Bytes	3 Bytes	4 Bytes	Average bytecode size
H263	4.3%	0.7%	51.3%	31%	12.6%	19.7 bits
ghostscript	2.7%	2.5%	40.4%	32.6%	21.7%	21.4 bits
MPEG4	3.7%	0.9%	48%	34.5%	12.4%	20 bits

Table 7: Compression distribution with -O1 optimization level.

Application	SuperInstruction Nb	Profits	bytecode%	rate profits
H263	9	9.9kb	6.6%	7.6%
ghostscript	4	37.8 kb	3.2%	9%
MPEG4	4	27.3 kb	4.8%	4.8%

Table 8: SuperInstruction profits on non optimized code

efficiency of the compression. Results show that, in the non optimized code, around 15% of the assembly code is dropped or combined at the semantic upgrade step, whereas only 3% in average is dropped in the optimized versions. The compiler has performed instructions removal we did when no compiler optimization was allowed. Without data and interpreter with tables, the compression ratio (encoded code size divided by original selected code size) is 55% to 60%. The average size of the encoded instruction is 20 bits for H263 and MPEG4 optimized compressed code, 21.4 bits for Ghostscript optimized compressed code. For the non optimized code, compressed instructions need 17 bits for H263 and 18.1 bits for Ghostscript and MPEG4.

Superinstructions Table 8 shows the benefits in size (kBytes) provided by superinstructions. We save from 9.9 kbytes up to 37.8 kbytes, which can also be expressed by the compression rate increase from 4.8% up to 9% on the compression rate and a bytecode reduction from 3.2% up to 6.6%. This is very promising since we have only considered simple superinstructions. The number of superinstructions selected depends on the instruction mix of the application. The larger is the application, the higher is the number of different instructions used. As a consequence, fewer superinstructions can be defined due to the lack of available opcodes. Nevertheless superinstructions enable to save significant memory space,

since the most frequent instructions (static occurrences) are likely to appear more often in larger code.

4 Related works

Lot of research works have be devoted to code compression. In this section we review the main works related to this study.

Hoogerbrugge et al (Hoogerbrugge and al. 1999) have developed a compression system for Trimedia VLIW architecture. Their approach uses bytecode compression which is interpreted by a stack-based virtual machine. They notice that, using their technique, it is difficult to compress smaller code units than functions. This is due to the needed moves from register to the virtual stack for the interpretation. The interpreter, written in assembly, is not easily retargetable. They report a good compression ratio, but the Trimedia code exposes more unused instructions (nop in the VLIW instructions for instance) than SPARC code.

Debray (Debray and al. 1999) et al have investigating trade-offs between the compression ratio and the slowdown of the application. They show that they can achieve a good compression ratio by compressing infrequently executed code, without decreasing drastically performances. Our approach differs from their compression scheme (based on Huffman statistic compression algorithm) as they require a complex execution RAM buffer management. As a consequence, their technique, to our knowledge, is not suitable for multi-task environment.

Lain et al (Larin & Conte 1999) propose the design of a tailored new ISA for each application. They investigate hardware decompression between memory and cache. According to the application, they choose new ISA design or statistic encoding in streams compression scheme to maximize the compression ratio. Our work uses the same basis for the ISA design. In their approach, decompression is achieved using a specific hardware mechanism.

A technique complementary to other compression methods is code factorization (Kirovski and al. 1997) : common parts of code are replaced by a function call to the common sequence. We use this technique to factorize the common part of stub. This approach avoids overhead in code size (interpreter) but needs more efforts to expose common sequences. The reported compression ratio is low (from 5% to 12%) if used alone.

Ernst et al (Ernst and al. 1997) have developed BRISC which is an interpretable compressed program format for the Omniware virtual machine. Repeated sequences of instructions in the Omni VM RISC code are replaced with a bytecode that refers to a macro-instructions. This is an application of superinstructions. BRISC can also be decompressed and compiled in native x86 instructions. They achieve compact code for an execution slowdown of 13. Their technique is not retargetable and do not address the granularity issue.

5 Conclusion

In this paper we propose a machine independent, multi task, software compression scheme. The process is machine independent and can be applied to most 32/64bit RISC like architectures.

The compression scheme achieves a high compression ratio of the applications (from 13% up to 33% reduction of the total size of the application code). The design of the compression scheme has been optimized to allow fast context switching from (un)compressed to (un)compressed code regions. This allows to consider in the same framework optimizing code techniques that would expand code size (unrolling, inlining, ...) for hot spots while infrequently executed code sections are compressed.

The next steps of this work will focus on improving the speed of the interpretor for a better tradeoff between code size and performance.

References

Brocard L & Sugalski D 2001 Parrot virtual machine. Parrot is a virtual machine used to efficiently execute bytecode for interpreted languages.
*http://www.parrotcode.org/

Debray S, Evans W & Muth R 1999 Compiler techniques for code compression Technical Report TR99-07.
*http://citeseer.nj.nec.com/debray99compiler.html

Ernst J, Evans W, Fraser C, S.Lucco & Proebsting T 1997 *in* 'In Proceedings of ACM SIGPLAN'97'.

Fraser C & Proebsting T 1995 Custom instructions sets for code compression. Unpublished manuscrit.
*http://research.microsoft.com/~toddpro/papers/pldi2.s

GCC . GNU C Compiler.
*http://www.cl.cam.ac.uk/texinfodoc/gcc_toc.html

Ghostscript Postscript interpretor. .
*http://www.cs.wisc.edu/~ghost/index.html

h263 . .
*http://www.ee.ubc.ca/image/h263plus/

Hoogerbrugge J, Augusteijn L, Trum J & van de Wiel R 1999 *Software Practice and Experience* **29**(11), 1005–1023.
*http://citeseer.nj.nec.com/hoogerbrugge99code.html

Huffman D A 1952 *in* 'Proc. IERE' Vol. 40 pp. 1098–1101.

Kirovski D, Kin J & Mangione-Smith W H 1997 *in* 'International Symposium on Microarchitecture' pp. 204–213.
*citeseer.nj.nec.com/kirovski97procedure.html

Larin S Y & Conte T M 1999 *in* 'Proceedings: 32nd Annual International Symposium on Microarchitecture: Haifa, Israel, November 16–18, 1999' IEEE Computer Society Press 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA pp. 82–92.
*http://citeseer.nj.nec.com/larin99compilerdriven.html

MPEG4 . .
*http://www.tnt.uni-hannover.de/project/eu/momusys/mom-overview.html

Proebsting T A 1995 *in* 'Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages' San Francisco, California pp. 322–332.
*http://citeseer.nj.nec.com/proebsting95optimizing.html

Ziv J & Lempel A 1978 *IEEE Transactions on Information Theory* **24**(5), 530–536.

Contents

Chapitre 3

Rendre un compilateur plus efficace : la spécialisation de code

3.1 Introduction

Dans l'introduction de cette thèse nous avons vu que le nombre important de paramètres à prendre en compte lors de la compilation rend difficile l'obtention de bonnes performances.

Par ailleurs les techniques d'analyse partielle de code sont connues depuis plusieurs années [22], mais n'ont jamais fonctionné et été évaluées sur des codes réalistes, parce que les évaluateurs partiels étaient basés sur des langages non utilisés en production, ou parce que le surcoût nécessité par la spécialisation était prohibitif.

L'objet de cette étude qui a été l'objet de la thèse de M. Minhaj Khan [14] a été de mettre en oeuvre et d'évaluer les technique de spécialisations de code en utilisant des technologies de production :

- Les compilateurs les plus récents et performants pour la cible utilisée (ici les compilateurs Intel et gcc pour le processeur Itanium)
- Les codes les plus gourmands en calculs. L'article suivant montre les résultats obtenus avec des FFT, l'article [15] fournit des résultats obtenus sur les benchmarck SPEC.

L'idée de base de ces expérimentations est d'aider le compilateur en remplaçant certains paramètres par des constantes et en effectuant plusieurs compilations pour des valeurs différentes. Les codes obtenus sont peu nombreux et peuvent être classés en "templates" que l'on peut ensuite utiliser lors du "runtime" en ne modifiant que quelques instructions.

3.2 Résultats obtenus

3.2.1 La connaissance des paramètres clefs

Cette étude a permis de montrer que même sur des codes a priori simples les compilateurs sont encore loin de fournir tout le potentiel possible parce qu'ils n'ont pas la connaissance des paramètres importants d'une application :

- Les dimensions des tableaux
- La dimension des itérations
- Les décalages utilisés à l'intérieur d'un tableau (les offsets)

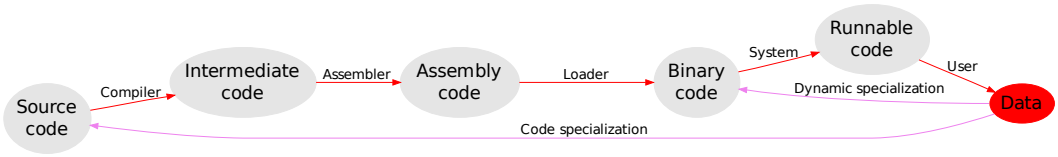


FIG. 3.1 – La chaîne de compilation pour la spécialisation

– L'adresse des tableaux alloués en mémoire

Ces paramètres ne sont connus qu'au moment de l'exécution et changent d'une exécution à l'autre car c'est l'utilisateur qui choisit ces paramètres à l'exécution (les tailles des images, le volume d'un son), voire même les données traitées qui contiennent des variabilités que l'on ne peut pas prédire au moment de la compilation ("l'effet Titanic").

On peut remarquer que les seuls paramètres que l'on peut spécialiser de la sorte sont les variables scalaires de type entier. D'une part car ce sont les variables de ce type qui sont le plus souvent utilisées pour décrire le flot de contrôle d'un programme, les programmes contrôlés par des valeurs numériques réelles sont plus rare.

3.2.2 La vitesse de spécialisation

Alors que les articles sur l'évaluation partielle ne traitent pas du temps de spécialisation (car il est très long), les différents articles que nous avons publiés sur le sujet ont montré que la spécialisation de code peut se faire de façon très rapide.

Au lieu d'utiliser une infrastructure complexe au moment de l'exécution du programme, nous avons utilisé la puissance du compilateur au moment de la compilation statique pour ne laisser que quelques instructions binaires à modifier au moment de l'exécution.

Les temps de spécialisation sont alors de l'ordre de 9 cycles machine par instruction à spécialiser (pour un Itanium) sachant qu'il n'y a que quelques instructions à spécialiser par fonctions (moins de 10%).

3.2.3 Le comportement des compilateurs

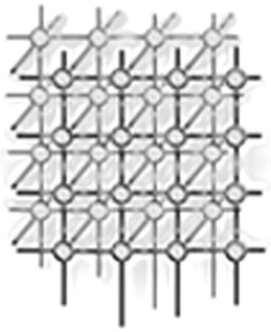
Les phases et passes de compilations sont de plus en plus nombreuses et complexes au point qu'il est difficile de comprendre son comportement [19].

Nous avons constaté lors de cette étude que malgré leur capacité à générer du code pour des processeurs complexes, les compilateurs ne fournissent pas un grand nombre de "template" différents lors des phases de spécialisation lorsque l'on fait varier les paramètres de spécialisation.

Je présume que, sauf pour les paramètres des boucles, les compilateurs sont peu sensibles aux informations de dimensionnement (tableaux, strides, etc) le plus souvent parce qu'ils ne sont pas connus lors de la compilation statique. Mais lorsqu'on fixe ces paramètres ils produisent un meilleur code grâce à la propagation de constantes et l'évaluation partielle.

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2000; **00**:1–6 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Improving Performance of Optimized Kernels Through Fast Instantiations of Templates



Minhaj Ahmad Khan*, H.-P. Charles†, D. Barthou‡

University of Versailles, France.

SUMMARY

To fully exploit instruction-level parallelism offered by modern processors, compilers need the necessary information available during execution of the program. This advocates for iterative or dynamic compilation. Unfortunately, dynamic compilation is suitable only for applications where the cost of compilation may be amortized by multiple invocations of the same code. Similarly, the cost of iterative compilation makes it impractical to be widely used for performance improvement.

In this article, we suggest a novel approach of improving performance of mathematical kernels through fast instantiations of templates. Optimized templates are generated at static compile time with a limited number of compilations. The initial instantiations of these templates are performed at static compile time, and the runtime instantiations are performed with a very small overhead through specialized data, requiring no computations at runtime. It represents the best solution in terms of reduced overhead incurring at static compile time and dynamic compile time.

The experiments have been performed on Itanium-II architecture using highly optimized kernels of `ATLAS` and `FFTW` with *icc* and *gcc* compilers.

1. Introduction

Code specialization [1, 2] has proved to be beneficial for many applications. Given specialized values, the compiler is able to generate highly optimized code. This lets the compiler fully exploit the ILP provided by modern processors. Code specialization is more effectively performed at runtime due to unavailability of values at static compile time. Runtime information may be used by many optimizations: it may remove dependencies by providing information related to array indexes or more aggressive optimizations can be invoked if the compiler is given loop counts. Most of the optimizations such as constant propagation, dead code elimination, loop unrolling are related to integer parameters, and therefore, we target only integer parameters in this article.

*E-mail: Minhaj.Khan@prism.uvsq.fr
†E-mail: Henri-Pierre.Charles@prism.uvsq.fr
‡E-mail: Denis.Barthou@prism.uvsq.fr

Received 11 September 2007
Revised 11 September 2007

```
void ATL_UAXPY(const int N, const SCALAR alpha, const TYPE *X,
const int incX, TYPE *Y, const int incY)
{
   int i;
   for (i=0; i < N; i++, X += incX, Y += incY)
   *Y += alpha * *X;
}
```

Figure 1. ATLAS BLAS-1 KERNEL

A difficulty is that code specialization requires the information that is not available at static compile time. Therefore, keeping different code versions for all possible values actually degrades the performance. The dynamic compilation can mitigate this problem by generating code during execution of the application. Runtime optimization and specialization systems [2, 3, 4, 5, 11, 6] may improve performance but require many invocations of the same code to amortize the overhead of runtime code generation. Similarly, the overhead of iterative compilation [17, 14] has to be reduced to make it effective enough to achieve the best performance within a small time limit.

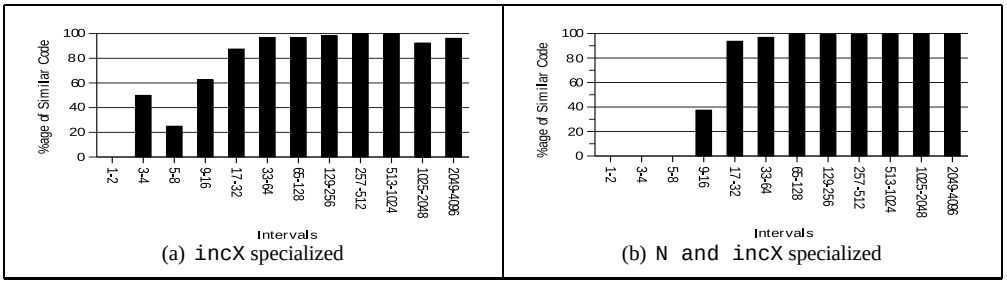
The optimization approach suggested in this article incorporates fast instantiations of the templates performed at both static compile time and dynamic compile time. An optimized template is generated after specializing code at static compile time. This template can be used for a large set of values and requires a limited set of binary instructions to be specialized during execution. This is referred to as the instantiation of the template. While instantiating template, we use specialized data also generated at static compile time. Since the optimizations have already been performed at static compile time, the overhead of code generation is much reduced than the overhead of other dynamic code generation and optimization systems. Moreover, the static compile time overhead is reduced by using analysis and code validation against the required criteria. This optimization approach produces significant improvement in the performance of FFTW and ATLAS kernels.

The remainder of the paper is organized as follows. Section 2 discusses compiler behavior with an example. Section 3 provides the conditions and the criteria which are essential to apply this technique. The steps included in the algorithm are elaborated in Section 4. The implementation framework has been discussed in Section 5 with experimental results presented in Section 6.

2. Motivating Example

The impact of code specialization varies depending upon the code and the use of parameter in the code. It may result in partial evaluation with constant folding, constant propagation and dead-code elimination. Similarly, for parameters involved in loop control (stride or bounds), it can lead the compiler to fully unroll, change the schedule of the loop (parameters of the software pipeline for instance) or perform other loop transformations (fusion for instance).

Consider the code in Figure 1 of a BLAS-1 kernel in ATLAS [7]. To specialize this code, one or more parameters are replaced with constants from 1 to 4096 to generate different versions. Analyzing object code generated by *icc* compiler V9.1 reveals that many of these versions differ only in immediate



(a) `incX` specialized (b) `N and incX` specialized

Figure 2. Code generated by *icc* compiler (v 9.1) after specialization of code

constants. The specialization produces versions which are similar for intervals of specialized values (shown in Figure 2). Any of these similar versions can be used as a template [11, 14] to perform the functionality of other versions by changing values that differ.

This runtime adaptation of templates can be improved by performing runtime activities that do not require any computations. Moreover, the template generation must be efficient enough so as to obtain the maximum benefit within a limited time instead of acquiring an exhaustive approach. Both of these characteristics enhance the scope of template-based specialization approaches.

This article proposes an optimization approach that is based on the efficient instantiations of the templates at static compile time and dynamic compile time. This procedure involves the instantiation of the templates at static compile time for the initial value followed by the generation of a lightweight runtime specializer. The runtime specializer performs specialization of a limited set of binary instructions to instantiate templates at runtime.

3. Template Generation and Validation

This section describes how a template is generated and the required context for which it is valid.

Given an interval of parameter values, a specialized binary code can be generated by static compiler. Let S be a function which takes as arguments the code of a function C and the value of some function parameter, and generates a specialized version. Given two values p_1 and p_2 of a parameter, we obtain two versions of specialized binary code, $S(C, p_1) = S_{p1}$ and $S(C, p_2) = S_{p2}$, which fulfill the below given criteria.

$$S_{p1} - S_{p2} = D_{p1}, \tag{1}$$

$$S_{p2} - S_{p1} = D_{p2}. \tag{2}$$

where D_{p1} and D_{p2} are the sets of some immediate values. This implies that the versions differ only in some constants.

Each $D_{p1}[i] \in D_{p1}$, and, $D_{p2}[i] \in D_{p2}$, should be based upon affine formulae as given in the following equations.

$$D_{p1}[i] = f_i(p_1) = p_1 * \alpha_i + \beta_i, \forall i = 1 \ldots n, \quad (3)$$

$$D_{p2}[i] = f_i(p_2) = p_2 * \alpha_i + \beta_i, \forall i = 1 \ldots n. \quad (4)$$

where α and β are constants, and, $n = |D_{p1}| = |D_{p2}|$.

Once identified, the set of such instructions is *annotated* and is used for generation of runtime specializer.

3.1. Code Validation

A runtime instantiation of the templates corresponds to the dynamic specialization performed at runtime. As this is only a substitution and the new value in template slot(immediate operand) must be valid to be compatible with maximum and minimum value of an instruction in the code. For runtime value $v \in V$, and the coefficients α and β corresponding to n *annotated* instructions, we must have,

$$I_{max} \geq v * \alpha_i + \beta_i \geq I_{min}, \forall i = 1 \ldots n, \quad (5)$$

where I_{max} and I_{min} are the maximum and minimum values that an immediate operand can have in *annotated* instruction.

For the runtime value v, the code must also be validated against the predicates. Let $T_i(v)$ be $i - th$ predicate involving parameter specialized with value v, the new instantiation for v should be valid iff the following predicates are equal.

$$T_i(p_1) = T_i(p_2) = T_i(v). \quad (6)$$

The specialized code can now be instantiated as described below.

3.2. Fast Template Instantiations for Mathematical Kernels

In order to reduce overhead at static compile time and at runtime, we generate static specialized data array. This is achieved by restricting the template values to be based only on affine functions. It means that the functions $f_1, \ldots, f_n$ required for the instantiation of a template must be of the form: $f_i(x) = \alpha_i.x + \beta_i$, where x is parameter value with which code is specialized.

Given an interval (of values) V , the specialized data corresponding to a runtime value $v \in V$, can be generated by solving the equations and evaluating the formula. Therefore, we have $D_v[\ i\ ] = \alpha_i.v + \beta_i$. The starting index of specialized data in the D_v array (corresponding to each instantiation of the template) is also computed at static compile time.

With the specialized data, it becomes easier to instantiate the template without calculating any runtime values for the instructions.

4. Approach of Fast Template-based Instantiations

For an interval in a value profile [8], the code is specialized by defining values (taken from the interval) for the parameters. Following steps are then required to perform fast instantiations of the optimized code obtained after specialization.

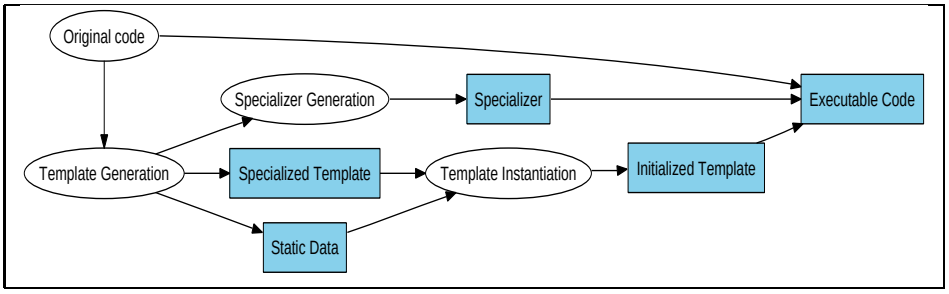


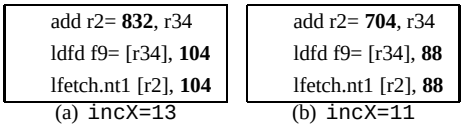
Figure 3. Overview of optimization approach

1. ***Code validation*** The specialized code is analyzed to obtain a template that can be used for a large range of values. A template is searched by finding the versions which meet the conditions (1-6) described in Section 3. The instructions differing (by immediate constants) in these versions are *annotated* to calculate the locations to be modified at runtime. Any of these versions can be used as a template, and may now be instantiated.
2. ***Initial instantiations of the template with specialized data*** The instantiations of the template require specialized data which can be generated at static compile time. For the equivalent versions specialized with $v_1, v_2, ..., v_k$, values of k parameters, and having n differences, we compute data through formulae of the form: $D_{v_1,v_2,..,v_k}[i] = \sum_{j=1}^{k} (\alpha_{ij} \times v_j + \beta_{ij})$ for $1 \leq i \leq n$. It represents the specialized data (for each *annotated* instruction in the template) that will be inserted during execution. The template is then instantiated at static compile time by modifying its object code instructions.
3. ***Runtime instantiations through specializer*** The runtime specializer contains the self-modifying code that can insert specialized data at specified locations. The information regarding precise location of each *annotated* instruction is also calculated at static compile time. Since the specialized data is already generated at static compile time, the runtime instantiations require no computations for specializing the code.

The static versions are used for interval values where the template could not be generated. A small piece of wrapper code is also generated to redirect control to template code, to statically specialized code and un-specialized code.

5. Implementation Framework and Experimentation

The main steps (shown in Figure 3) leading towards fast instantiations of templates have been automated in *HySpec*[11] framework. It supports specialization of function parameters (integral) as described below.



(a) `incX=13` (b) `incX=11`

Figure 4. Object code generated by `icc` over Itanium-II

5.1. Instrumentation and Code Specialization

The specialization of code proceeds by defining values (obtained through instrumentation for value profiling[8]) of function parameters to generate versions. The code is also instrumented to contain call to wrapper function which can redirect control to a specific version.

5.2. Generation of Data for Fast Template Instantiations

An analysis of object code versions is performed to validate so that the two versions differ only in immediate constants and these constants must be based on affine functions (shown in Figures 4(a) and 4(b)). After checking code equivalence, the formulae are generated by solving the system of equations. The formulae are assumed to be of the form: $v = \alpha \times specialized\_value + \beta$. For an interval, all the values v corresponding to each instruction differing in equivalent versions are evaluated through formula and a linear array of specialized data is generated. This results in minimum overhead incurring at static compile time.

The specialized data array represents the values with which the binary code will be specialized. The offset of data from where the values start for an instance of template, are also computed at static compile time.

5.3. Wrapper Code and Initial Template Instantiation

The call to wrapper is already instrumented, but its code is generated after the generation of templates. The branches in the wrapper redirect control to proper versions. For some valid range of interval, it contains calls to statically specialized code, dynamically specialized template, and original code as fallback. For dynamic templates, it first contains call to specializer followed by the call to template code.

The initial instantiation of the template takes place by modifying object code instructions at static compile time. The addresses of these instructions have already been computed after analysis and represent the template slots to be modified.

The runtime instantiation of the template is accomplished by a runtime specializer (shown in Figure 5) which is able to efficiently insert values at specified locations. The same locations in object code as computed for initial instantiation are re-used for runtime specialization. Each invocation of *Instruction Specializer* puts statically specialized data into these locations followed by the cache coherence.

```
D15 = Array of data for specialization with value 15
   InstructionSpecializer(Instruction Address0, D15[0]) //D15[0]=960
   InstructionSpecializer(Instruction Address1, D15[1]) //D15[1]=120
   InstructionSpecializer(Instruction Address2, D15[2]) //D15[2]=120
   CacheCoherence(Instruction Address0)
    .........
```

Figure 5. Binary Template Specializer for runtime value 15

For the template to be specialized with runtime value 15, the coefficients $\alpha_1 = 64$, $\alpha_2 = 8$, $\alpha_3 = 8$ and $\beta_1 = 0$, $\beta_2 = 0$, $\beta_3 = 0$ will generate 960, 120 and 120 respectively as elements of data array.

6. Experimental Results

The experiments for fast instantiations of the templates have been tested on IA-64 (1.5GHZ, 32KB L1I+D, 256KB L2 and 3MB L3 cache) platform with the *icc* compiler v 9.1 and *gcc* compiler v 4.3. For compilation, the *-O3* optimization has been used together with default parameters with which these libraries were configured.

6.1. ATLAS Benchmarks

ATLAS [7] library generates optimized code for the specific architecture by making use of optimizations offered by the processor and compiler with which it is configured. The BLAS-3(dgemm), BLAS-2(dgemv) and BLAS-1(daxpy) kernels in ATLAS library have been specialized. The speedup percentage obtained with respect to the standard ATLAS code has been shown in Figures 6, 7 and 8 respectively.

For BLAS-3, the code specialization has been applied to different modules* ATL_dJIK . Similarly, for BLAS-1 and BLAS-2, the routine $ATL\_daxpy\_xp1yp1aXbX$ has been specialized. The specialized codes (six versions) mostly benefit from loop based optimizations including data prefetching, software pipelining and loop unrolling.

A significant improvement in the performance of the BLAS kernels is obtained with *icc* compiler since ATLAS standard code (obtained after tuning) contains routines which are not very much specialized. This is similar in case of *gcc* compiler, for which performance is gained for dgemv and daxpy. However, in dgemm, the code is already specialized to a large extent. This restricts the compiler to only perform partial evaluation together with a small reduction in the number of loads after code specialization.

*functions with a1_b1 and aX_bX suffixes

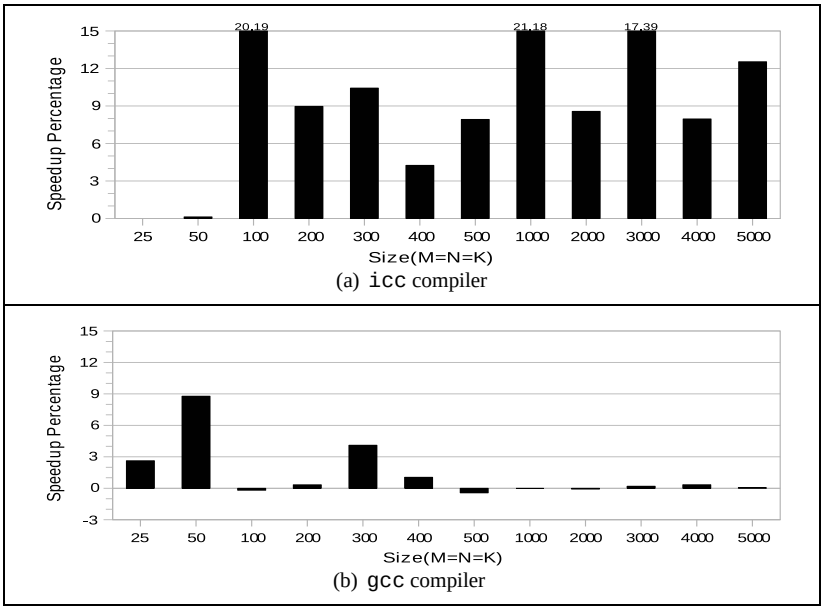


Figure 6. BLAS-3(DGEMM) speedup percentage w.r.t. standard BLAS-3 code

6.2. FFTW Benchmark

FFTW [9] library contains C routines called *codelets* to compute Discrete Fourier Transform (DFT) of real and complex data and of arbitrary input size in *O(nlogn)*. It makes use of the best configuration for the corresponding architecture called *wisdom*.

Figure 9 shows the speedup obtained for calculating complex DFTs of powers of 2. The calculation of single DFT may comprise repeated invocations of multiple codelets.

The overall behavior of the compilers optimization depends upon both, the value of the specialized parameter and the size of the specialized code. For large codelets in FFTW, the code generated by compiler becomes very similar to un-specialized code thereby producing less improvement in performance.

6.3. Overhead of Runtime Instantiations

The overhead[†] of runtime instantiations (shown in Figure 10) is very small since the initial instantiations have already been made at static compile time. Moreover, the static specialized data

[†] calculated through profiling of wrapper which contains specializer invocation

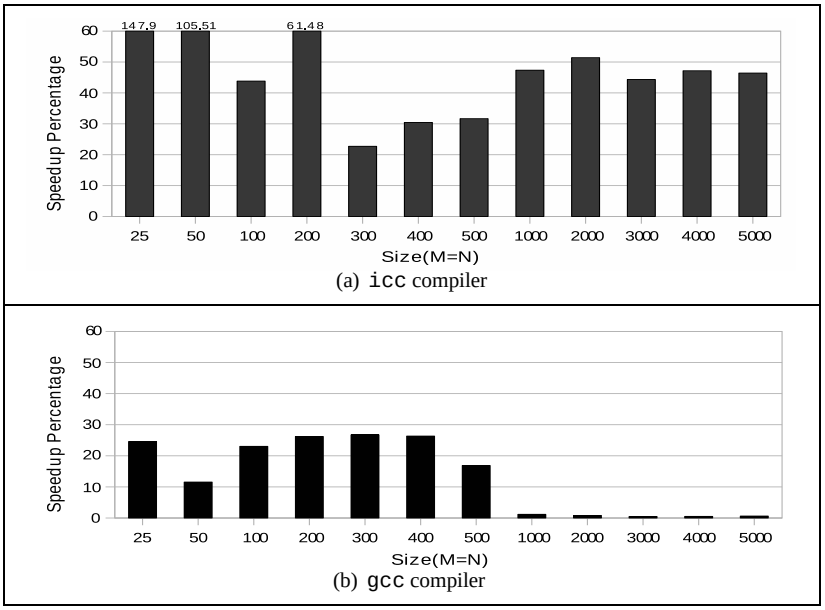


Figure 7. BLAS-2(DGEMV) speedup percentage w.r.t. standard BLAS-2 code

eliminates the need for any runtime computation. The modification of single instruction takes an average of 9 cycles on Itanium-II. The calculation of offset of specialized data and cache coherence are both performed once per runtime specialization of the entire template.

The size increase (SI) which is calculated as $\frac{Size\ of\ specialized\ code}{Size\ of\ un-specialized\ code}$, and percentage of unique instantiations (PUI), calculated as $\left(\frac{Number\ of\ unique\ runtime\ instantiations}{Number\ of\ calls}\right) * 100$, are also provided in Figure 10. The SI becomes large particularly for small un-specialized code for which addition of template size and static code becomes significant. The PUI for both the benchmarks is small, since the kernels do not frequently change values. Moreover, the initial instantiation eliminates the need for dynamic instantiation.

7. Related Work

Many code specialization and dynamic code generation systems use templates for performing optimizations at runtime.

The Tempo specializer [2] performs partial evaluation of code by propagating information after analysis of code. The dynamic code can then be generated by specializing code with runtime values. In contrast, our approach performs fast instantiations of the templates optimized at static compile time, and therefore, it incurs minimum overhead at runtime.

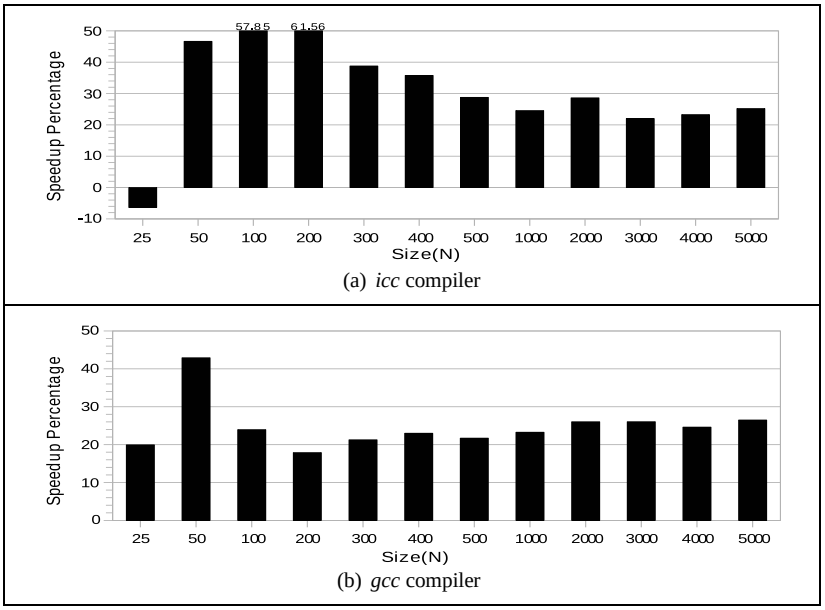
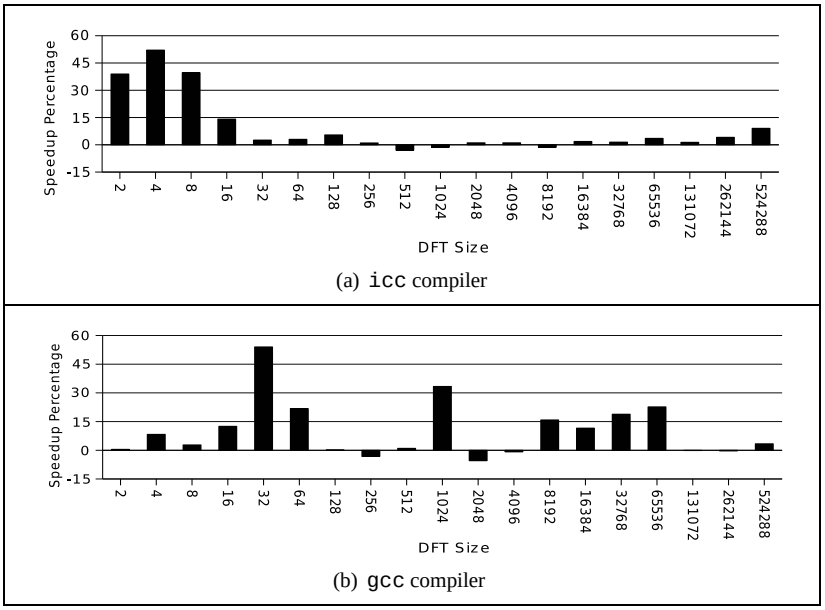


Figure 8. BLAS-1(DAXPY) speedup percentage w.r.t. standard BLAS-1 code

Our previous work on dynamic specialization [11] uses affine formulae to be computed at runtime. Although this approach incurs small overhead as compared to other specializers, the computations are required before generating new instruction. In contrast, the approach suggested in this paper incorporates specialized data computed at static compile time. This elminiates the need to perform any computations at runtime. The instantiations of the templates therefore incur the smallest possible overhead. Similarly, another specialization approach described in [14] uses exhaustive specialization of code at compile time to generate the templates. In contrast, the templates for fast instantiations are generated with minimum number of compilations. The overhead of iterative compilation is reduced by using the analysis and validation criteria which enables a large range of specialized data to be generated through computation of affine formulae.

C-Mix [12] partial evaluator performs source-to-source transformation at static compile time. It analyzes the code and makes use of specialized constructs to perform partial evaluation. Although it does not require runtime activities, it is limited to optimizing code for which the values already exist, thereby limiting the scope of candidate parameters for specialization. Similarly, Tick C [3] compiler generates optimized code at runtime. A significant improvement in performance is achieved but the runtime code generation activity and optimizations incur a large overhead. In contrast, we minimize the runtime overhead by optimizing templates and generating specialized data at static compile time.

Some other dynamic code generation systems have been suggested in [5, 13, 15] that categorize the compilation process into different stages at which different optimizations can be performed.



(a) `icc` compiler

(b) `gcc` compiler

Figure 9. `FFTW` speedup w.r.t. standard code

Benchmark	**Avg. Overhead**		**Avg. SI**		**Avg. PUI**	
	icc	*gcc*	*icc*	*gcc*	*icc*	*gcc*
FFTW	3.97%	3.91%	2.31	2.67	10.5%	9.8%
BLAS	1.60%	1.77%	3.14	3.49	7.0%	5.4%

Figure 10. Summary of overhead, code size increase and percentage of unique instantiations of the templates

These models are effective enough to improve performance, however they require the programmer intervention to decide which optimization could be appropriate at which stage.

In recent work related to dynamic optimization systems, the Dynamo [16] framework developed at HP Labs. can interpret instruction stream. The hot traces of code are searched and a fragment cache is incorporated to apply dynamic optimizations. Another dynamic optimization framework, ADORE [6], is used to perform dynamic cache prefetching. During execution, the hardware-based counters are used to compute the cache miss penalties which are reduced through prefetching. Similarly, the continuous compilation (CoCo) system [17] makes use of static and dynamic optimizations. The dynamic optimizations are continuously applied and are based on analysis of code and performance. The continuous monitoring/profiling implemented in these systems incurs large overhead and thereby limits these approaches to code having large number of invocations.

8. Conclusion

This article presents an optimization approach which is based on efficient instantiations of optimized templates. For complex benchmarks such as ATLAS and FFTW, we are able to achieve good speedup through this approach with minimum increase in code size.

The fast instantiations of the templates at static compile time and runtime reduce the overhead of runtime specialization. The optimized template is generated by specializing code at static compile time. The static analysis and validation criteria are used to generate specialized data which reduces the overhead of iterative specialization. The runtime instantiations of the templates use specialized data to ensure the minimum possible overhead incurring during execution.

The cost of runtime code generation in this approach is far less than that in existing specializers and code generators. The optimizations on the template which are already performed at static compile time (due to specialization) bring significant improvement in performance.

REFERENCES

1. Muth, R., Watterson, S.A., Debray, S.K.: Code specialization based on value profiles. In: Static Analysis Symposium. (2000) 340–359
2. Consel, C., Hornof, L., Marlet, R., Muller, G., Thibault, S., Volanschi, E.N.: Tempo: Specializing Systems Applications and Beyond. ACM Computing Surveys **30**(3es) (1998)
3. Poletto, M., Hsieh, W.C., Engler, D.R., Kaashoek, F.M.: 'c and tcc : A language and compiler for dynamic code generation. ACM Transactions on Programming Languages and Systems **21** (1999) 324–369
4. Leone, M., Dybvig, R.K.: Dynamo : A staged compiler architecture for dynamic program optimization. Technical report, Indiana University (1997)
5. Grant, B., Mock, M., Philipose, M., Chambers, C., Eggers, S.J.: DyC : An expressive annotation-directed dynamic compiler for c. Technical report, Department of Computer Science and Engineering,University of Washington (1999)
6. Lu, J., Chen, H., Yew, P.C., Hsu, W.C.: Design and Implementation of a Lightweight Dynamic Optimization System. Journal of Instruction-Level Parallelism **6** (2004)
7. Whaley, R.C., Dongarra, J.: Automatically Tuned Linear Algebra Software. Technical Report UT-CS-97-366, University of Tennessee (1997) URL : `http://www.netlib.org/lapack/lawns/lawn131.ps`.
8. Calder, B., Feller, P., Eustace, A.: Value profiling. In: International Symposium on Microarchitecture. (1997) 259–269
9. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. In: In proceedings of IEEE, Vol. 93, no. 2. (2005) 216–231
10. Minhaj Ahmad Khan, Henri-Pierre Charles: Applying code specialization to FFT libraries for integral parameters. In: 19th Intl. Workshop on Languages and Compilers for Parallel Computing, Nov. 2-4, 2006, New Orleans, USA. (2006)
11. Minhaj Ahmad Khan, H.P. Charles, D. Barthou: Reducing code size explosion through low-overhead specialization In: 11th Annual Workshop on the Interaction between Compilers and Computer Architecture, Phoenix, USA.. (2007)
12. Makholm, H.: Specializing C- an introduction to the principles behind C-Mix. Technical report, Computer Science Department, University of Copenhagen (1999)
13. Grant, B., Mock, M., Philipose, M., Chambers, C., Eggers, S.J.: Annotation-Directed Run-Time Specialization in C. In: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97), ACM (1997) 163–178
14. Minhaj Ahmad Khan, H.-P. Charles, D. Barthou An effective automated approach to specialization of code In: 20th Intl. Workshop on Languages and Compilers for Parallel Computing, Oct. 11-13, 2007, Urbana, Illinois, USA. (2007)
15. Consel, C., Hornof, L., François Noël, Noyé, J., Volanschi, N.: A uniform approach for compile-time and run-time specialization. In: Partial Evaluation. International Seminar., Dagstuhl Castle, Germany, Springer-Verlag, Berlin, Germany (1996) 54–72
16. Bala, V., Duesterwald, E., Banerjia, S.: Dynamo: a transparent dynamic optimization system. ACM SIGPLAN Notices **35**(5) (2000) 1–12
17. Childers, B.R., Davidson, J.W., soffa, M.L.: Continuous compilation: A new approach to aggressive and adaptive code transformation. In: NSF Workshop on Next Generation Software. (2003)

Chapitre 4

Mélanger les données et les instructions : les compilettes

4.1 Introduction

L'objet de cette étude qui a été l'objet de la thèse de Melle Karine Brifault [5] était

- d'une part de permettre l'utilisation d'opérateurs multimédias difficilement utilisables dans un compilateur statique pour plusieurs processeurs cible (Itanium, Sparc, PowerPC)
- et d'autre part d'explorer des pistes d'utilisation de la compilation au vol en exploitant les données run-time de l'application

Nous avons inventé la notion de "compilettes". Ce sont de mini-compilateurs embarqués dans une application, ils permettent lors de l'exécution du programme de produire un code efficace car elles peuvent s'appuyer sur la connaissance :

des valeurs des données qui permettent d'optimiser le code produit pour des valeurs. Par exemple un produit matrice par vecteur dont on sait que la matrice est constante pendant un certain temps et qu'elle va être utilisés pour multiplier des millions de vecteur peut être optimisée pour cette matrice.

Il n'est pas nécessaire d'implémenter des optimisations complexes, une simple propagation de constantes suffit pour générer un code optimal.

de la position des données en mémoire qui permettent de générer un accès efficace pour optimiser les accès aux caches et réduire les problèmes d'alignements

des opérations multimédia disponibles . En effet d'une part l'utilisation des opérateurs multimédia ou vectoriels est difficile parce que les langages de programmation classique n'ont pas de support pour cela. Et d'autre part un programme est parfois destiné a fonctionner sur plusieurs ordinateurs de générations différentes ou avec des caractéristiques variables.

Les compilettes peuvent s'adapter au type et à a génération de processeur sur lesquelles elles fonctionnent.

des registres disponibles dans le processeur . L'utilisation des registres par un compilateur est très conservatrice, car soumise aux algorithmes classiques d'allocation de registre. Ils sont alloués via une politique d'allocation statique plus ou moins complexe.

Les compilettes permettent d'en avoir un usage plus souple et de les utiliser comme des structures de données classiques (pile, file, tableaux, etc) et de les manipuler de façon globale ce qui est impossible avec des allocateurs statiques.

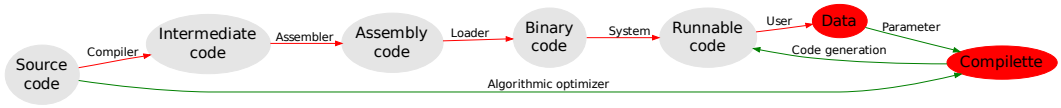


FIG. 4.1 – La chaîne de compilation les compilettes

4.2 Résultats obtenus

L'article suivant montre que l'on peut obtenir

- des résultats de speedup impressionnants face à des compilateurs optimisant pour un petit investissement en programmation
- des résultats de speedup indépendants de l'architecture de la machine car le critère d'optimisation principal est la valeur des paramètres.
  Dans ce cas on peut se contenter d'une planification d'instruction sous optimale, le gain obtenu grâce à l'optimisation sur les valeurs l'emporte largement.

Efficient data driven run-time code generation

Karine Brifault
PR*i*SM, Université de Versailles
Versailles, France
kbrifa@prism.uvsq.fr

Henri-Pierre Charles
PR*i*SM, Université de Versailles
Versailles, France
hpc@prism.uvsq.fr

ABSTRACT

Knowledge of data values at run-time allows us to generate better code in terms of efficiency, size and power consumption.

This paper introduces a low-level compiling technique based on a minimal code generator with parametric embedded sections to generate binary code at run-time. This generator called a "compilet" creates code and allocates registers using the data input. Then, it generates the needed instructions. Our measurements, performed on Itanium 2 and PowerPC platforms have shown a speed improvement of 43% on the Itanium 2 platform and 41% on the PowerPC one.

The proposed technique proves to be particularly useful in the case of intensively reused functions in graphic applications, where the advantages of dynamic compilation have not been fully taken into account yet.

1. INTRODUCTION

Many different techniques [17] improve code performance in terms of efficiency, power consumption or size. In classical static compilation, heuristics or other techniques such as loop unrolling or strength reduction are used. But, the information knowledge such as data values is missing at compile-time which would be very useful for statements generation when data values and invariants can be exploited [15]. The resulted code where less computations have to be executed, is often superior in speed to statically optimized code. As the data values can change at each run, the profile-based technique is not really convincing. The dynamic compilation is then the most suitable technique which complements static compilation taking advantage of data values and invariants for every run[11].

New methods have been introduced with the appearance of virtual machines. Java [10], for instance, has split compiling into a two-step process, consisting of two translation phases (source to bytecode and bytecode to native) and an execution phase. The first stage generates platform-independent bytecode, and the second one, at run-time, generates target code on demand. This is called the Just In Time (JIT) compiler principle [1]. Target-dependent code is generated only at run-time, using a complex piece of code linked to the Java virtual machine (JVM). Hence, it takes time to compile a method, especially when we want apply any kind of optimization, and when this compilation has to be done each time the application is run. Moreover, a good JIT is complex and takes up a considerable amount of space.

Another optimization method uses techniques to inline assembly code inside a C program. The gcc `asm` extension is an example of such. It allows to inline assembly instructions inside a source function. Another example is the `Altivec` extension for gcc which allows the use of multimedia instructions that many C compilers can not generate. But, in this case, developers must have an extended knowledge of every platform on which they program, and of their specific instructions, in order to optimize the code, as assembly is platform-dependent. This technique is frequently used in image processing as compilers do not usually have the capability to use graphical instructions without a link to a specific multimedia library.

This paper deals with applying dynamic compilation to multimedia applications on two different platforms using a toolkit called ccg [5][20]. Multimedia applications become one of the main used ones on personal computers[4]: because of the spreading use of complex processes geared towards them, mainstream users are requiring increasingly more efficient computers, at the lowest possible cost[13]. Hence, this topic represents a challenging target for us. We are working towards this goal by achieving more with a given computation power, in spite of an unavoidable overhead due to the dynamic code generation. The proposed technique determines the threshold at which reuses will off-set the overhead. In addition, it generates only the instructions that will be needed to process the data. Another advantage is that we use all the instruction set, even the graphical instructions, without relying on any multimedia libraries. However, we have to solve the problem of this code generation cost which at run-time can be high [8][7] and higher as the code complexity increases.

In section 2, we describe our experimental environment and the methodology used to create our "compilets". In section 3, we present and discuss our results on convolution filters

and geometric transformations. In section 4, we review some related work putting in prospect our contribution. Finally, in section 5, we conclude with several directions that will be explored in future research.

2. EXPERIMENTAL ENVIRONMENT

Dynamic compilation is not a new concept, but our intention is to apply it efficiently to multimedia, where some interesting restrictions exist and make interesting our algorithm of code generation. For this study, the execution of speed improvement have been monitored.

2.1 Methodology

To validate our approach, we experimented two multimedia applications, the convolution filter and a vector-matrix multiply. Convolution consists in applying a matrix to each image pixel, in order to create another image where pixels are a linear combination of their neighbors (Figure 1). We use the multimedia instructions which allow to process data values as vector or pixel and not as an integer or a float using saturated arithmetics to reduce the number of assembly statements and the size of the generated code.

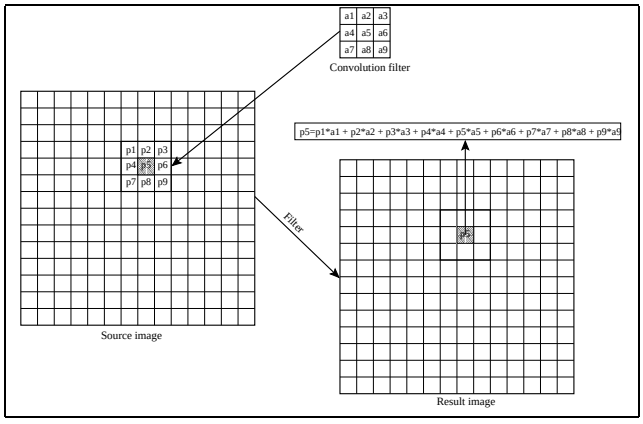


Figure 1: Example of a small image and convolution filter

In the vector-matrix multiply experiment, we consider an implementation from OpenGL Mesa-6.0.1, which contains unrolling and linearization:

```
d[0]= v[0]*m[ 0] + v[1]*m[ 1] + v[2]*m[ 2] + v[3]*m[ 3];
d[1]= v[0]*m[ 4] + v[1]*m[ 5] + v[2]*m[ 6] + v[3]*m[ 7];
d[2]= v[0]*m[ 8] + v[1]*m[ 9] + v[2]*m[10] + v[3]*m[11];
d[3]= v[0]*m[12] + v[1]*m[13] + v[2]*m[14] + v[3]*m[15];

where v represents the source vector, m the
unidimensional array which contains the matrix values
and d, the destination vector.
```

For the last experiment, we use two variants according to our algorithm: the standard form which is equivalent of the static version with some little improvements such as a better use of ILP, and the optimized form is not really that optimal, but is easy to set up. It consists in reducing loads and arithmetic instructions.

Besides, a specific algorithm has been created to take into account the particular arithmetics associated to pixels. In image processing, pixels, generally described by 4-byte integers, are controlled by the saturated arithmetics. For this reason we break the usual compilation rules in our algorithm by eliminating either the multiply operations by zero or by one. We can consider that this removal is of little consequence. Besides, it brings two benefits: first, the operations number is reduced as all the computations which are useless (multiplication by 1 or 0) disappear; Second, this reduction of instructions number decreases the execution time.

2.2 Overview of the compilets system

Our technique operates in two steps at runtime. The first one is to call a compilet, defined at compile-time in ccg language, which generates binary code according to the target architecture, the knowledge of data and our algorithm (Figure 2 and more detailed [2]).

```
for(i=0 ;  i<4 ;  ++i)
  if(0.0 == mat[row][i])
    if(0==i) genSub(regDst(lno),regDst(lno),regDst(lno));
  else
    if(1.0 == mat[row][i])
      if(0==i) genMove(regDst(lno),regSrc(i));
      else     genAdd(regDst(lno),regSrc(i),regDst(lno));
    else
      if(-1.0 == mat[row][i])
        if(0==i) genNeg(regDst(lno),regSrc(i));
        else     genSub(regDst(lno),regSrc(i),regDst(lno));
      else
        if(0==i) genMul(regDst(lno),regMat(lno,0),
                        regSrc(0));
        else     genMulAdd(regDst(lno),regMat(lno,i),
                           regSrc(i),regDst(lno));
```

Figure 2: Part of a compilet for a vector-matrix multiply

As the optimization of the dynamic generated code is done only at the level of 3D transformations and not at the level of data retrieval, there is a waste of time and space with the loading of zeroes and ones. Hence, another "parametric embedded code" of machine code has been added:

```
tmp = m[i][0];
if ((0==opt) || ((1.0!=tmp) && (-1.0!=tmp) && (0.0!=tmp)))
    genLoad(regMat(i,j),SIZEOFFLOAT*(4*i+j),regBase(0));
```

A compilet consists in reducing loads and arithmetic instructions such as computations by zero. It chooses adequate instructions and generates correct register allocation for a given matrix and a given architecture. The generated binary code is an executable function.

The second phase consists in calling this minimal generated function to execute the multimedia application.

One of the interests of our approach is the fact that all of our codes are written in C language in which several segments of dynamic code are embedded via an ISA (Instruction Set Architecture) which gives a portability to the compilets on different architectures. Hence, a developper can only use some defined compilets without any knowledge of the architecture.

2.3 Hardware and Software setup

Our experiments are made on Itanium 2 and PowerPC architectures. The Itanium processor is a uniprocessor operating

at 900 MHz with 2 GB of memory. The cache memory hierarchy is organized in three levels, the L1D level of 16 KB, the L2 level unified of 256 KB and the L3 level of 1.5 MB. The PowerPC is a 800 MHz processor machine with 256 MB of memory. The cache memory hierarchy is organized in two levels: the L1 cache of 32KB and L2 cache of 256 KB.

The test machines were respectively running Linux Red Hat 7.1 based on 2.4.18 SMP kernel for Itanium 2 and MacOS 10.2.3 Darwin Kernel version on PowerPC.

The compiler for all platforms is the 3.3 version of gcc. We observe some constraints in the choice of the compilation options (set to *CFLAGS=-O7 -fno-inline -ffast-math*) for all platforms. Other more specific information such as *-mcpu=970 -mtune=970* for PowerPC are added. Results are given in CPU cycles and are an average of 100000 of experimental measures.

3. RESULTS

3.1 Vector-Matrix experiments

Here (Figure 3), we focused exclusively on the cost of running a function dynamically generated using both modes, standard and optimized, compared to one in a standard static C version. First, we notice the amount of time saved. In the case of the IPF architecture, in any experiment, we get, at best here, a 30% speed improvement with our dynamic version compared to the static version generated by the gcc v3.3 compiler. From that, we verify that gcc compilers do not take yet advantage of the specifics of IPF architecture (such as microSIMD), as it does for PowerPC architecture. In the last experiment, the performance of our dynamic versions, and the opportunity to use them, depends strictly on the number of zeroes in the transformation matrix. The main reason is that static compilers have schedulers able to interleave data loadings and arithmetic operations. Our optimized version, which has been made at little cost, has those instructions kept apart.

In the previous section, we compared the dynamic and static versions of our different transformations only in their use. We voluntarily omitted the cost of the dynamic code generation here. However, the overhead of this generation can not be neglected and, to redeem it, we have to reuse the dynamic version several times. Our goal is to find a satisfying compromise from which dynamic compilation becomes profitable.

In Table 1, the cost of the code generation is detailed for both versions: the optimized and the standard one. The cost is relatively important for all platforms, particularly on Itanium 2 platform, where ccg has a scheduler to choose the template fields (a tag which declares explicitly the instruction type on IPF).

Platforms	Std. code (cycles)	Opt. code (cycles)
Itanium 2	19580	16260
PowerPC	5780	6370

Table 1: Overhead in CPU cycles for dense matrices

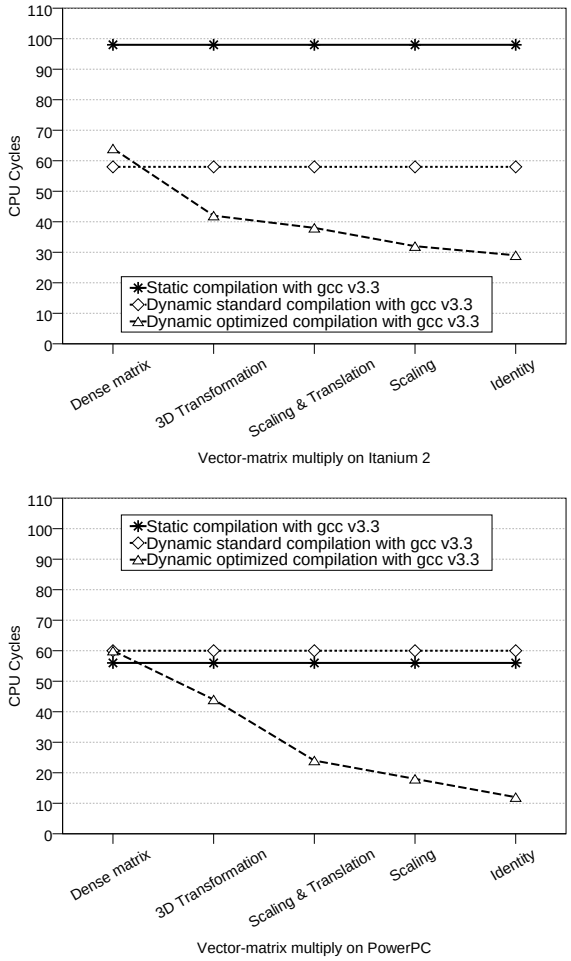
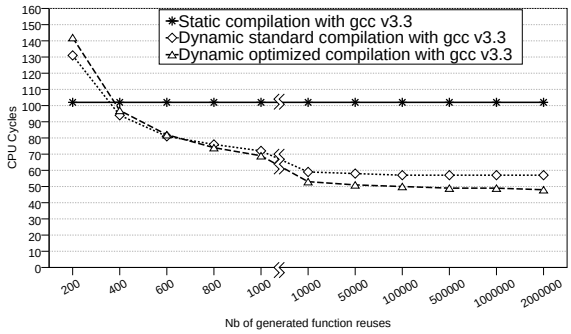


Figure 3: Comparison between the different dynamic versions and their C counterpart by architecture

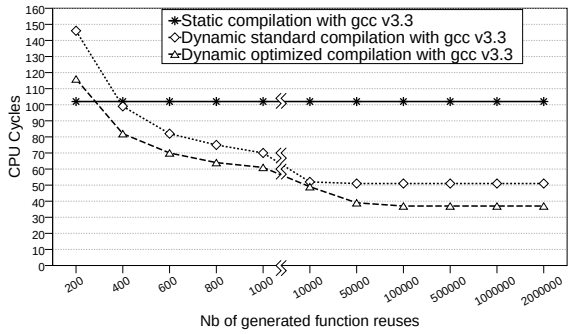
This simple experiment is suitable to show the dependency between overhead redeeming of code generation and function reuses.

On the Itanium platform (Figure 4), the overhead due to the generation of dynamic code, standard or optimized, is redeemed starting from 400 uses of the generated function. It is an acceptable number, as regular exploitation of a 3D transformation can use more than one million points. In the case of a 3D and scaling transformation respectively, the cost of the optimized generated code is covered at only 200 uses with a 16% and 28%speedup improvement.

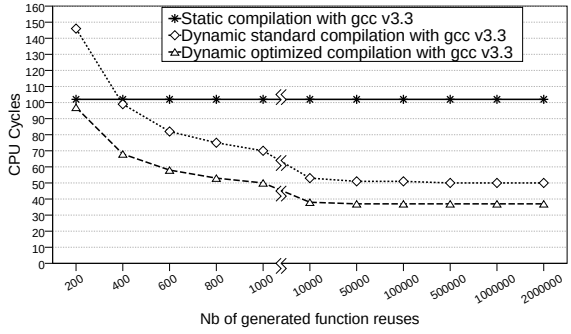
The highest speedup improvement of around 43% has been obtained for the processing of 50000 points with the dynamically generated function. This means that dynamic compilation will reach an optimal efficiency with 3D scenery which size is usually around one million points or more. However its efficiency appears since around 300 points. This gain, earned thanks to a small effort of conception and coding, largely justifies the use of dynamic compilation for image processing on the Itanium architecture.



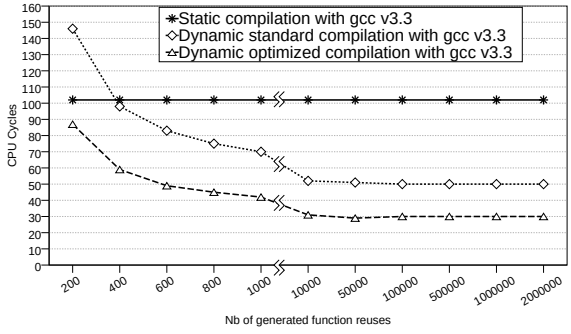
Dense Matrix



3D transformation Matrix



Scaling and Translation Matrix

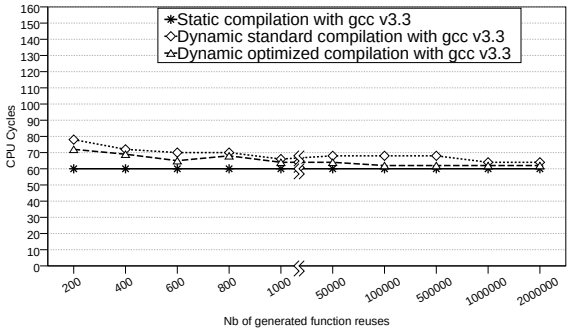


Scaling Matrix

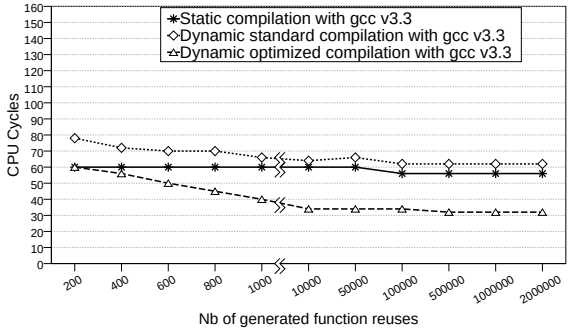
Figure 4: Itanium 2 Architecture

On the PowerPC platform (Figure 5), for a dense matrix, the overhead of compilation is still significant to be redeemed in a small number of uses and, once more, our dynamic versions are not enough efficient in comparison with the static function. We have there to go above 1000 uses of our dynamic function for the dynamic compilation to be profitable. However, in the case of image processing, once again the result is acceptable. This observation gets reversed when the matrix contains three or more zeroes. We then get a clear advantage favoring the dynamic version over the static version, as we get 8 to 25% speedup improvement compared to the latter.

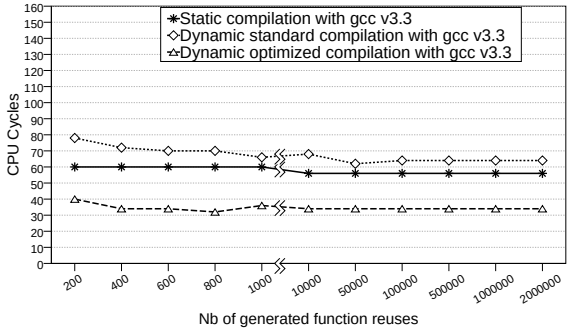
The highest speedup improvement of around 41% is reached for a 10000 points processing with the dynamically generated function. As already noticed, our dynamic version is here again completely adapted to image processing.



Dense Matrix



3D transformation Matrix



Scaling and Translation Matrix

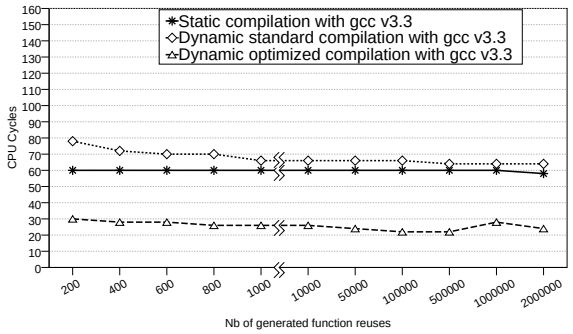


Figure 5: PowerPC Architecture

Dynamically generated code size and binary compilet size. Our technique bestows another advantage. The binary code size is smaller compared to the C version generated by static compilers. Indeed, with our algorithm, all the useless computations are eliminated. For instance, in the Itanium 2 architecture, the generated static code has a size of 658 bytes in the case of a dense matrix. For a geometric transformation matrix, this size is of 466 bytes and of 370 bytes for a scaling matrix. It is to be noted that on PowerPC, for a dense matrix, the code size is around 168 bytes. So, the code size on Itanium architecture is twice as large as its counterparts on PowerPC. In fact, on the Itanium 2 platform, all instructions are inevitably stored per three in structures named "bundles". Each bundle contains a tag named "template field" that determines which functional unit is able to evaluate the statement and what interlacing can exist between the functional units. Those template fields only give the possibility to store a unique floating-point unit per two bundles and, in order to fill those bundles, ccg adds "nop" instructions. This explains the code size on Itanium 2 in comparison with its counterparts of other platforms. However, the generated dynamic code remains smaller on Itanium 2 architecture than the static one, at worst by 22%, and can be reduced by 43% for real scaling matrices.

These results do not take into account the size of the compilet. On all architectures, the memory footprint is below 8 kbytes.

Platforms	compilet code size (bytes)
PowerPC	3260
Itanium 2	7107

The size of the compilet added to the generated code makes it bigger than its static counterpart. But a same compilet, of a size hardly reaching 8 kbytes, is intended for intensive use, can generate several different codes and, above all, able to elaborate a greatly optimized code.

3.2 Filter experiment

For the convolution filter, we have used the assembly graphical instructions which allow to process data values as vector or pixel composed by three components and not as an integer or a float using saturated arithmetics. The use of this multimedia instruction allows to reduce the number of assembly instructions and the size of the generated code.

We also took interest in the size of convolution filter. On the different platforms, the number of registers available is variable. For instance, on the IPF architecture, 96 integer registers are at the service of our convolution filter. But, on PowerPC, the number of registers is only a half of what it is on IPF. Taking into account this condition, when the size of convolution filter is not too large, the matrix can be stored in register thus allowing to earn back the data loading time. Hence, we have two possibilities during the creation of our compilet: the storing in register or the loading of matrix values. This choice depends on the size of the convolution filter and on the number of registers in the target architecture.

In the case of the mean filter (Figure 6), the speedup we obtain using our compilets for code generation is of 4 at worst on the Itanium 2 architecture, and of 17 at best. In fact, we use the micro-SIMD instructions parallelism in our compilets, which is adapted to graphic applications, whereas the gcc compiler can only use a classical instruction set. On the PowerPC platform, our generated code is compared to the Altivec version, and it notices a speedup of 1.4. Moreover, with a sparse filter, our compilets obtain, on PowerPC platform a speedup of 3. Finally, we have tested a $5*5$ Gaussian filter and we have obtained on both PowerPC and Itanium 2 platforms, a speedup of 1.5 and 7.6 respectively.

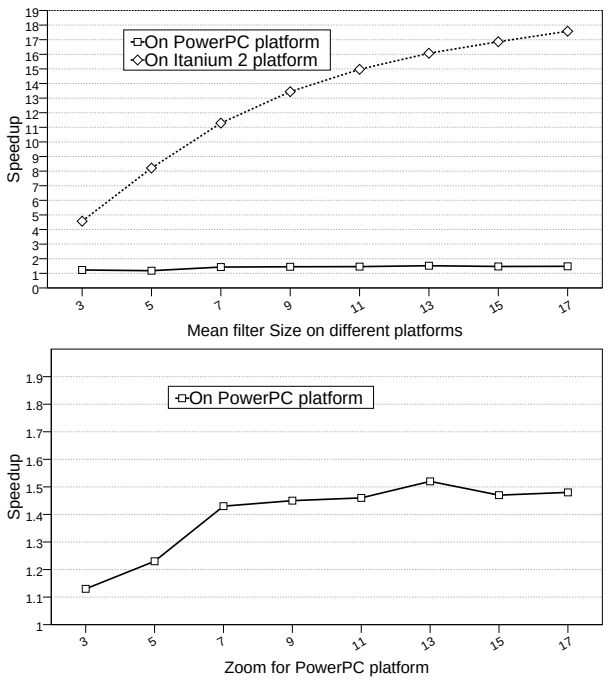


Figure 6: Mean filters: comparison between dynamic and C versions for a 512*512 image

4. RELATED WORK

Among all researchs on dynamic compilation built over the last ten years, only a small part is close to our concerns here. For the majority of them, the dynamic code generator is often a large and complex program which allows the pro-

grammers to express algorithms in high-level languages such as Fabius [12], Calpa [16], Dynamo [14] or HotSpot [19]. On the opposite, our compilet, which uses low level compiler techniques, is a minimal dynamic code generator which creates only the needed code with a small cost in terms of size and coding time.

Some researchs on staged dynamic compilers, which postpone a portion of compilation until runtime when code can be specialized based on runtime values [3][9], focus on spending as little time as possible in the dynamic compiler performing extensive offline pre-computations. This technique, which is also used in our compilets negates the need for any intermediate representation at runtime. However, we choose to return at low level compiler techniques to take into account the graphical instruction set for each platform, if it exists, and obtain better performance.

In the dynamic compilation for multimedia, a group which has conceived the FFTW [6] does a static optimized code generation with dynamic scheduling. Our compilet prefers using a code generator which creates the optimized adapted code at runtime.

5. CONCLUSION

In this paper, we confirmed that it is possible to use the data values input as parameters for an effective run-time code generator in multimedia applications. Results highlight that the produced binary code can be of satisfying quality, tailored for the data input, while needing only a slight programming effort.

It is important to note that the "compilet" is of very small size and is platform-independent. With a small programming effort, we get better performance than a static compiler consisting of thousands of code lines.

This technique gives direct access to specific instructions of the target processor. This is an elegant way to deal with arithmetics such as saturated arithmetics which are seldom supported by compilers. The C dialtec Cg [18] allows to generate code for graphical processors but can not mix C and assembly code nor interact with general purpose processors.

In future works, we will implement those "compilets" in mainstream applications such as 3D visualization softwares, and we will include them in more complex run-time processes, able to detect when and where this technology will be useful.

6. REFERENCES

[1] Aycock, J. A brief history of just-in-time. *ACM Comput. Surv. Volume 35*, Issue 2 (2003), 97–113.

[2] Brifault, K., and Charles, H.-P. Effective and economical run-time code generation driven by data for multimedia applications. Technical Report 2004/62, PRiSM Laboratory, University of Versailles, July 2004.

[3] Consel, C., and Nol, F. A general approach for run-time specialization and its application to c. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (January 1996), ACM Press, pp. 145–156.

[4] Diefendorff, K., and Dubey, P. R. How multimedia workloads will change processor design. *IEEE Computer Volume 30*, No. 9 (September 1997), 43–45.

[5] Folliot, B., Piumarta, I., Seinturier, L., Baillarguet, C., Khoury, C., Leger, A., and A1, F. O. Beyond flexibility and reflection: The virtual virtual machine approach. In *International Workshop on Cluster Computing* (September 2001), vol. 2326, Springer-Verlag Heidelberg, p. 16.

[6] Frigo, M., and Johnson, S. G. The fastest Fourier transform in the west. Technical Report MIT-LCS-TR-728, Massachusetts Institute of Technology, September 1997.

[7] Grant, B., Mock, M., Philipose, M., Chambers, C., and Eggers, S. J. The benefits and costs of dyc's run-time optimizations. *ACM Trans. Program. Lang. Syst. Volume 22*, Issue 5 (September 2000), 932–972.

[8] Grant, B., Mock, M., Philipose, M., Chambers, C., and Eggers, S. J. Dyc: An expressive annotation-directed dynamic compiler for c. *Theoretical Computer Science Volume 248*, Issue 1-2 (October 2000).

[9] Grant, B., Philipose, M., Mock, M., Chambers, C., and Eggers, S. J. An evaluation of staged run-time optimizations in dyc. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation* (May 1999), ACM Press, pp. 293–304.

[10] Joy, B., Steele, G., Gosling, J., and Bracha, G. *Java(TM) Language Specification (First Edition).* Addison-Wesley Pub Co, September 1996.

[11] Kennedy, K., and Allen, R. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann, October 2001.

[12] Lee, P., and Leone, M. Optimizing ml with run-time code generation. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation* (1996), ACM Press, pp. 137–148.

[13] Lee, R. B., and Smith, M. D. Media processing: a new design target. *IEEE Micro Volume 16* (January 1997), 43–45.

[14] Leone, M., and Dybvig, R. K. Dynamo: A staged compiler architecture for dynamic program optimization. Technical Report 490, Department of Computer Science, Indiana University, September 1997.

[15] Leone, M., and Lee, P. A declarative approach to run-time code generation. In *Workshop Record of WCSSS 1996: The Inaugural Workshop on Compiler Support for System Software* (February 1996), ACM Press, pp. 8–17.

[16] Mock, M., Chambers, C., and Eggers, S. J. Calpa: a tool for automating selective dynamic compilation. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture* (December 2000), ACM Press, pp. 291–302.

[17] Muchnick, S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, August 1997.

[18] NVIDIA Corporate Office. *NVIDIA Cg Toolkit: Cg Language Specification*, nvidia corporation ed. 2701 San Tomas Expressway, Santa Clara, CA 95050, August 2002.

[19] Paleczny, M., Vick, C., and Click, C. The java hotspot server compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium* (April 2001), vol. JVM 2001, USENIX, the Advanced Computing Systems Association, pp. 1–13.

[20] Piumarta, I. Ccg: a tool for writing dynamic code generators. In *Workshop on simplicity, performance and portability in virtual machine design held in conjunction with OOPSLA 1999 Conference* (November 1999).

Chapitre 5

Perspectives et conclusions

J'ai tenté dans ce document de souligner les problèmes liés à la compilation pour les applications hautes performances et j'ai montré les activités que j'ai mené dans ce domaine dans 3 différentes directions.

Dans ce chapitre je tenterai de tracer quelques perspectives que l'on peut imagine pour ce domaine de recherche.

5.1 Des compilateurs partout

J'espère avoir convaincu le lecteur que la génération et l'optimisation dynamique de code apporte un intérêt pour les applications haute performances.

De nombreuses applications utilisent déjà des générateurs de code dynamiques :

- Les GPU (Graphical Process Unit) sont rapidement devenus populaires car ils fournissent un ratio $\frac{puissance\ de\ calcul}{prix}$ impressionnant. Mais ce sont des processeurs dédiés, qui ne contiennent pas de système d'exploitation et qui ont une mémoire locale différente de celle de l'ordinateur central.
  Les GPU sont généralement programmés grâce à un langage de haut niveau, mais pour des raisons de portabilité sur les différents modèles de cartes graphiques les phases d'assemblage et d'édition de liens se passent lors de l'initialisation du programme.
  Il n'y a pas de phase d'optimisation et les programmes sont chargés une fois pour toute.
  Malheureusement les constructeurs de cartes graphiques ne fournissent pas les jeux d'instruction des cartes graphiques, il est donc impossible de construire un générateur binaire pour ces architectures
- Les systèmes d'exploitation [16] utilisent pour faire du filtrage de paquets IP un compilateurs capable de transformer les requêtes de filtrage en langage d'assemblage spécialisé pour le filtrage réseau.

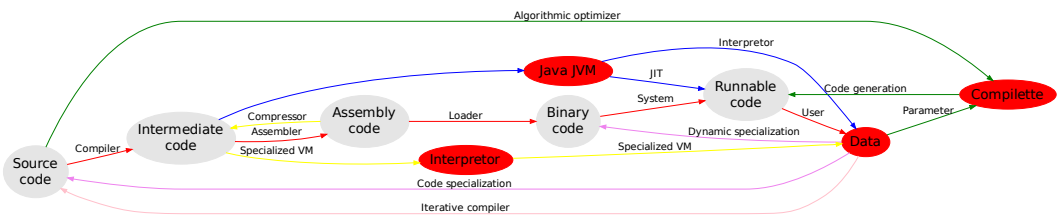


FIG. 5.1 – Les méthodes d'exécution et de compilations entrelacées

- Les systèmes de base de données contiennent tous un optimiseur qui permet de compiler la requête à la base de donnée vers une représentation intermédiaire. Ce n'est pas à proprement parler de la génération de code binaire mais les techniques utilisées sont assez similaires à celles utilisées dans les compilateurs à l'exception de l'objectif qui est de minimiser les mouvements de données et non pas de générer du code.

De nombreux outils de génération de code existent déjà, et l'on pourrait imaginer développer ce concept car les processeurs seront toujours de plus en plus difficiles à programmer.

- Le LARABEE, prochain processeur d'Intel [25], aura une multitude de core, des capacités vectorielles et des instructions spécialisées pour réaliser des traitements graphiques. Mais l'article indique :

  > All Larrabee SIMD vector units are fully programmable by Larrabee Native application programmers. Larrabee Native's C/C++ compiler includes a Larrabee version of Intel's auto- vectorization compiler technology. Developers who need to program Larrabee vector units directly may do so with C++ vector intrinsics or inline Larrabee assembly code.

  c'est à dire que pour exploiter au mieux ses capacités, il faudra programmer en assembleur vectoriel.
- Les applications des téléphones portables sont écrites en Java pour des raisons de facilité de déploiement. Le futur système Android réalisé par Google pour les téléphones portables n'y fait pas exception. Mais il utilise une machine virtuelle spéciale "Dalvik"[1]. C'est une machine virtuelle à registre.
  Aucune machine virtuelle pour téléphone portable ne contient de compilateur au vol alors que les processeurs qu'ils contiennent ont généralement peu de ressources en calcul.
- Le calcul intensif pour des applications dynamique. Les applications de calcul intensif peuvent être classées en plusieurs catégories :
  - Celles qui ne nécessitent aucune communication et peuvent se paralléliser de façon massive, sans qu'aucune communication entre les clients ne soit nécessaire. Les projets `Seti@Homme` ou `Bovine` sont de ce type[2].
  - Celles qui ne nécessitent qu'un défi pour la parallélisation et/ou la vectorisation : il faut trouver la bonne topologie et la bonne implémentations qui permettra de paralléliser au mieux les calculs. Ces codes fonctionne sur de grosses machines parallèles fortement couplées.
  - Celles qui nécessitent une adaptation dynamique car l'élément le plus variable l'évolution des données : exemple : la compression vidéo, le filtrage d'images, etc.
    Ces dernières bénéficieraient d'une adaptation dynamique du code à l'exécution.

5.2 Des compilateurs pour les données

L'augmentation de puissance des processeurs s'est accompagnée d'une augmentation des contraintes à respecter pour bénéficier de cette puissance.

La valeur des données, leur position en mémoire, leur position relative à une autre donnée sont quelques exemples de ces contraintes dont l'impact est devenu plus important qu'une mauvaise planification d'instructions.

[1] `http://en.wikipedia.org/wiki/Dalvik_virtual_machine`
[2] `http://setiathome.berkeley.edu/` et `http://www.distributed.net/`

La seule façon de prendre en compte ces paramètres est de générer le code final ou de le spécialiser au moment de l'exécution. Pour cela je compte continuer à développer le nouveau générateur de code binaire HPBCG qui permet de générer du code binaire :

- pour plusieurs jeux d'instructions (le CELL et l'ARM par exemple)
- en prenant en compte la valeur des données à l'exécution
- pour plusieurs processeurs. Les processeurs ARM, CELL et Itanium sont actuellement supportées.

5.3 Direction vers les applications

La complexité des applications ne fait que s'accroître et les opportunités de générer un code binaire spécialisé pour une application augmente également.

5.3.1 Les systèmes d'exploitation

Les systèmes d'exploitation utilisent des code binaires à l'exécution. Par exemple BPF [16] transforme à l'exécution des requêtes d'analyse de trame réseau en instructions pour une machine virtuelle spécialisée.

On peut imaginer qu'un tel mécanisme profite de la génération de code dynamique car l'analyse de trame pourrait s'y effectuer beaucoup plus rapidement.

L'arrivée d'interface réseau 10Gb/s nécessite de traiter de grande quantités de données rapidement et IPV6 nécessite de traiter des données de 128 bits de long. Ces 2 changements montrent qu'il y aura bientôt un problème de performance pour l'analyse de trames.

5.3.2 Les gestionnaires de base de donnée

Historiquement les gestionnaires de base de données traitaient les données uniquement à partir de disque dur. L'augmentation des tailles mémoire ont permit d'accélérer les traitements en ne travaillant qu'en mémoire.

L'utilisation de type de données binaire (images, points, lignes, polygones, cercle) ont permit d'utiliser les bases de données dans des domaines plus éloignés de leurs sujets habituels.

Ils utilisent pour cela des opérateurs spécialisés sous optimisés, car programmés en C par exemple pour postgresql[3] ou en Java pour oracle.

L'utilisation de compilateurs dynamique permettrai dans ce cas

- de réduire les accès à la mémoire en accédant, par exemple, aux 2 valeurs X et Y en un seul load plutôt qu'en 2
- de réduire le nombre d'instructions de calcul en utilisant des opérateurs vectoriels
- d'accélérer les requêtes en générant du code binaire lors de l'optimisation de la requête plutôt qu'en optimisant uniquement sur le volume d'entrée / sortie.

5.3.3 Les code de calcul scientifique

Les codes de calcul scientifiques bénéficient depuis longtemps de l'attention de la communauté des chercheurs en compilation et en architecture. La plupart des codes sont bien compris et les compilateurs permettent d'obtenir de bonne performances.

Pourtant des codes comme QCD[4] ont du être optimisés "à la main" car les compilateurs les plus performant n'arrivent pas à l'optimiser car :

[3] Fichier `./src/backend/utils/adt/geo_ops.c` de la distribution du serveur postgresql par exemple

[4] `http://en.wikipedia.org/wiki/Quantum_chromodynamics`

- leur code est trop complexe (en taille et en nombre d'opérations) et les optimiseurs sont limités en taille de code.
- ils ne permettent pas d'utiliser les opérateurs sur les nombres complexes qui sont la base de ces codes
- les opérateurs flottants nécessaires pour ces calculs sont restreints et ne peuvent pas être pleinement exploités comme l'explique la documentation du compilateur pour la machine spécialisée la plus performante actuellement pour ce code [13] :

  > «Our long term goal is to ensure that using the dual FPU will be no slower than single FPU code. This may not be achievable, due to the extra versioning necessary for alignment or aliasing checks, but the overhead should be minimized. »

Là encore on peut espérer que la génération dynamique sera capable de prendre en compte "the extra versioning necessary for alignment or aliasing checks".

5.3.4 Les applications multimédia

Nous avons vus dans la première partie que la performance des applications multimédia est variable en fonction des données qu'elles traitent.

Les compresseurs vidéos [6] les plus utilisés implémentent un certain nombre de version codées manuellement en assembleur.

Nous avons montré lors du projet PARA qu'il est possible de prendre en compte ces paramètres en utilisant un générateur dynamique qui permet de prendre en compte les problèmes d'alignements et les opérateurs multimédia.

5.3.5 La virtualisation

Les centres de calculs utilisent la virtualisation soit

- pour faire tourner plusieurs systèmes d'exploitation en même temps sur une seule machine
- soit pour émuler le code binaire d'une autre architecture

Ces procédés sont essentiels pour assurer l'exploitation de gros centres d'hébergement.

La traduction binaire de code implémentée par exemple dans [3] n'exploite que les compilateurs statiques et par la n'exploitent pas tout le potentiel de la machine cible.

Bibliographie

[1] Openoffice.org - the free and open productivity suite.

[2] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilateurs Principes, Techniques et Outils*. Interédition, 1989.

[3] Fabrice Bellard. Qemu, a fast and portable dynamic translator. *USENIX 2005 Annual Technical Conference*, pages 41–46, 2005.

[4] A. Bik, M. Girkar, and M. Haghighat. Incorporating intel mmx technology into a java jit compiler, 1999.

[5] Karine Brifault. *Contribution à la compilation dynamique pour des jeux d'instructions multimedia*. PhD thesis, Laboratoire PRiSM, Université de Versailles Saint-Quentin en Yvelines, Juin 2005.

[6] Ecole centrale de Paris. x264 - a free h264/avc encoder. http ://www.videolan.org/developers/x264.html.

[7] Henri-Pierre Charles. Loop unrolling revisited for super-scalar processors. In P. Quinton and als editors, editors, *Algorithms and parallel VLSI Architecture conference*, pages 311–316, 1991.

[8] Henri-Pierre Charles and Pierre Fraigniaud. Scheduling a scattering-gathering sequence on hypercubes. *Parallel Processing Letter*, 1993(92-133), 1993.

[9] Standard Performance Evaluation Corporation. http ://www.spec.org/.

[10] Liza Fireman, Erez Petrank, and Ayal Zaks. New algorithms for simd alignment, 2007.

[11] Matteo Frigo and Steven G. Johnson. FFTW : An adaptive software architecture for the FFT. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, WA, May 1998.

[12] GNU fundation. Gnu c compiler project.

[13] IBM. Exploiting the dual fpu in blue gene/l. Technical report, IBM, 2006.

[14] Minhaj Ahmad Khan. *Techniques de spécialisation de code pour architectures hautes performances*. PhD thesis, Laboratoire PRiSM, Université de Versailles Saint-Quentin en Yvelines, Juin 2008.

[15] Minhaj Ahmad Khan, H.-P.Charles, and Denis Barthou. An effective automated approach to specialization of code. In *Proceeding of the 20th International Workshop on Languages and Compilers for Parallel Computing, October 11-13, 2007, Urbana, Illinois*, October 2007.

[16] Steven McCanne and Van Jacobson. The bsd packet filter : A new architecture for user-level packet capture. In *1993 Winter USENIX conference, January 2529, 1993, San Diego, CA.*, 1993.

[17] Sun Microsystems. The source for java developers.

[18] Sun Microsystems. The java hotspot virtual machine. White paper, Sun Microsystems, 2001.

[19] Antoine Monsifrot, François Bodin, and René Quiniou. A machine learning approach to automatic production of compiler heuristics. In *In Artificial Intelligence : Methodology, Systems, Applications*, pages 41–50. Springer-Verlag, 2002.

[20] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics magazine*, 38, 1965.

[21] Moving Picture Experts Group (MPEG).

[22] Francois Noe, Charles Consel, and Projet Lande. A general approach for runtime specialization and its application to c. In *In 23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 145–156. ACM Press, 1996.

[23] Ecole Centrale Paris. Videolan project.

[24] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. SPIRAL : Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2) :232–275, 2005.

[25] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa andEd Grochowski, Toni Juan, and Pat Hanrahan3. Larrabee : A many-core x86 architecture for visual computing. In *ACM Transactions on Graphics, Vol. 27, No. 3, Article 18, Publication date : August 2008.*, 2008.

[26] vnc. Php : Hypertext preprocessor.

[27] R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software. Technical Report UT-CS-97-366, University of Tennessee, December 1997. URL : `http://www.netlib.org/lapack/lawns/lawn131.ps`.

[28] Niklaus Wirth. A plea for lean software. *Computer*, 28(2) :64–68, 1995.