Le débogage de code optimisé dans le contexte des systèmes embarqués.

Hugo Venturini

UNIVERSITÉ JOSEPH FOURIER – GRENOBLE I

THÈSE

pour l'obtention du grade de

DOCTEUR DE L'UNIVERSITÉ JOSEPH FOURIER

Spécialité : Informatique

préparée au laboratoire **Vérimag**

dans le cadre de l'**École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

présentée et soutenue publiquement par

Hugo Venturini

le 28 Mars 2008

Le débogage de code optimisé dans le contexte des systèmes embarqués.

Composition du Jury

Frédéric Pétrot *TIMA – Institut National Polytechnique de Grenoble*	Président
Susan L. Graham *University of California at Berkeley, CA, USA*	Rapporteure
Mireille Ducassé *IRISA – INSA de Rennes*	Rapporteure
Marc Pouzet *LRI – Université de Paris-Sud 11*	Examinateur
Jean-Claude Fernandez *Vérimag – Université Joseph Fourier – Grenoble*	Directeur
Miguel Santana *STMicroelectronics– Crolles*	Co-Directeur

À André Hervieu.

Remerciements

note : *Après lecture de ces remerciements, vous conservez le droit de lire le reste du manuscrit.*

Je remercie Frédéric Pétrot de m'avoir fait l'honneur de présider le jury de soutenance.
Je remercie Mireille Ducassé et Susan Graham pour le temps consacré à la relecture de ce manuscrit et pour les remarques qui en ont découlé.
Je remercie Marc Pouzet de m'avoir fait l'honneur de participer au jury de soutenance.
Ces travaux ont été menés sous la direction de Jean-Claude Fernandez et Miguel Santana. Je tiens à leur exprimer ma profonde gratitude pour leur disponibilité, leur rigueur et leur ouverture qui m'ont permis de mener ce travail à son terme. Ils ont su également m'encourager et me soutenir quand il le fallait, je les en remercie.
Je remercie tout particulièrement Frédéric Riss. Je tiens à lui exprimer ma reconnaissance pour avoir fait de ces trois années un parcours d'une si grande richesse. Son goût pour la programmation, sa culture informatique et sa façon de les transmettre restent une inspiration pour moi.
J'ai une pensée pour Bruno puis Guillaume qui ont partagé leur espace de travail avec moi ... et qui m'ont supporté.
Merci à Guillaume et Ludovic pour ces années de thèse que nous avons vécues ensemble.
Les membres du laboratoire Verimag ainsi que les ingénieurs de STMicroelectronics ont contribué au bon déroulement de ces travaux, tant par leur niveau scientifique et technique que par la bonne ambiance de travail qu'ils entretiennent chaque jour. Je remercie Laurent, Denis, Vincent, Thomas, Jean-Marc, Manuel, Valérie, Thierry, Vincent, Thomas, Laurent et Quentin du temps qu'ils ont consacré avec moi sur leurs outils à STMicroelectronics. Je remercie les *brin d'orage* de la première heure, Mathias, Ludovic, Guillaume, Jacques, Loïc et Simon, pour la qualité de leurs interventions et présentations. Merci à tous ceux qui nous ont rejoint.
Ce travail n'aurait pas été possible sans les moments de détente. Je remercie donc les gens avec qui j'ai partagé ces activités : ablok et ses after, la musique (et ses after aussi), les longues discussions cinéma, musique et autres média, le ciné-forum, les dîners, les picnics, les sorties...
J'ai une seconde petite pensée pour les gens avec qui j'ai habité, Luc, Amélie, Anna et Petter, puis Ludovic, Amélie et Lionel, merci (à vous aussi) de m'avoir supporté.
Je veux remercier les personnes que j'aurais oubliées dans ces remerciements de ne pas m'en tenir rigueur...
Merci à ma famille pour son soutien inconditionnel ... et pour la qualité du pot de thèse.
Enfin, je remercie Marie pour sa présence indéfectible.

Table des matières

Chapitre 1

Introduction

Sommaire

Le contexte global dans lequel s'inscrit cette thèse est celui du développement de programmes destinés à être exécutés sur systèmes embarqués. Ces derniers imposent souvent de lourdes contraintes telles qu'une mémoire réduite ou une puissance de calcul limitée. Le développeur qui destine sa production à de tels systèmes se doit de respecter ces contraintes et, pour ce faire, utilise divers outils et techniques. Parmi ces outils on trouve notamment des couples compilateur/débogueur. Les compilateurs dont la cible sont des systèmes contraints implémentent généralement plusieurs optimisations permettant des gains d'utilisation mémoire ou de temps de calcul. En outre, la structure habituelle d'un compilateur lui fait aussi collecter les informations de débogage au début du processus, informations qu'il ajoute au fichier binaire à la toute fin. De ce fait, si le programme est modifié, les informations de débogage présentes dans le fichier binaire sont en partie incorrectes pour ne pas dire complètement fausses dans certains cas. Or le débogueur s'appuie sur ces informations afin de répondre aux requêtes de l'utilisateur. Si elles ne correspondent à aucune réalité du programme, les réponses seront tout simplement erronées, ce qui n'est évidemment pas le comportement souhaité pour un débogueur.

Nous nous proposons donc de répondre à la question suivante : *"Est-il possible de déboguer du code hautement optimisé ?"* Question à laquelle nous répondrons par l'affirmative en présentant la conception puis la réalisation d'une solution logicielle partielle à ce problème. Elle est basée sur des outils existants et utilisés dans un contexte industriel de haute technologie. L'instrumentation de ces outils nous a ensuite permis de valider l'approche que nous avons choisie.

Dans la suite de ce chapitre, nous présentons les motivations de cette thèse ainsi que les contributions apportées. Nous terminons par le détail du plan du reste de ce manuscrit.

1.1 Le développement dans le monde des systèmes embarqués

Un langage de programmation[1] est un code de communication permettant à un être humain de dialoguer avec une machine en lui soumettant des instructions et en analysant les données matérielles fournies par le système. Le langage permet à la personne qui rédige un programme de faire abstraction de certains mécanismes internes tel que le modèle de calcul de la machine qui aboutissent au résultat désiré.

L'activité de rédaction du code source d'un programme est nommée programmation. Elle consiste en la mise en œuvre de techniques d'écriture et de résolution d'algorithmes informatiques, lesquelles sont fondées sur les mathématiques. À ce titre, un langage de programmation se distingue du langage mathématique par sa visée opérationnelle (une fonction et par extension un programme doit retourner une valeur), de sorte qu'un langage de programmation est toujours un compromis entre la puissance d'expression et la possibilité d'exécution [Dow97].

Les langages de programmation permettent de définir des ensembles d'instructions effectuées par l'ordinateur lors de l'exécution d'un programme. Il existe des milliers de langages de programmation chacun d'eux utilisant un paradigme particulier de programmation. Nous nous intéressons au langage *C* pour le développement de systèmes embarqués. Le *C* est un langage de programmation de type impératif créé par Dennis Ritchie et Ken Thompson en 1972 dans les Laboratoires Bell. Par la suite, Brian Kernighan aida à populariser le langage. En 1978, il fut notamment le principal auteur du livre *The C Programming Language* [RKT78]. La norme du langage *C* laisse la définition exacte du comportement de plusieurs opérations au choix du concepteur du compilateur. Ces comportements sont donc définis par l'implémentation. Cette propriété du *C* permet au compilateur d'utiliser directement les instructions proposées par le processeur et donc de compiler des programmes exécutables courts et efficaces. En contrepartie, c'est parfois la cause de bogues de portabilité des codes source écrits en *C*.

Un système est dit embarqué lorsqu'il est destiné à une ou plusieurs tâches précises. Il est dans certains cas une partie d'un produit ou d'un système bien plus gros[2]. Nous en utilisons de plus en plus dans la vie quotidienne : téléphones portables, lecteurs de musique digitale, logiciels de voitures, feux de signalisation routière, télécopieurs, pacemakers, pour ne citer que ceux-là. Ces systèmes ont pour caractéristique commune d'être contraints. Selon la machine sur laquelle un programme embarqué est destiné à être exécuté, il peut avoir à satisfaire différentes de ces contraintes :

- **L'utilisation mémoire :**
  La miniaturisation du matériel entraîne dans certains cas une réduction des capacités mémoires de l'appareil. Le programme embarqué doit donc tenir compte de cette contrainte et, par exemple, ne pas dépasser certaines limites d'occupation mémoire statique et d'utilisation mémoire lors de son exécution.
- **Le temps de calcul :**
  L'utilisation de programmes embarqués se fait souvent dans des contextes où le temps est un paramètre essentiel du système. Les délais d'exécution sont alors connus et bornés. De tels systèmes ont des propriétés *temps réel* dur (*safety critical*) ou mou (video, ...).
- **La consommation d'énergie :**
  Le caractère embarqué d'un système fait que ce dernier n'est pas toujours relié à une source infinie d'énergie. Le système doit alors gérer une certaine quantité d'énergie fournie par

[1] `http://fr.wikipedia.org/`
[2] Michael Barr's Embedded Systems Glossary

une batterie autonome (panneaux solaires, pile).

- **La sûreté/sécurité :**
  Certaines pannes de systèmes embarqués peuvent avoir des conséquences désastreuses tant d'un point de vue humain (train d'atterrissage d'un avion, appareillage médical) que d'un point de vue économique (système d'exploitation d'un distributeur d'argent). De tels systèmes sont dit *critiques*.

Un système embarqué peut avoir à satisfaire toutes ou parties de ces contraintes bien qu'elles puissent être contradictoires. L'exemple type est le gain de temps d'exécution d'une boucle ordonnancée par pipeline logiciel. Cette optimisation augmente la taille du code généré et donc son occupation en mémoire. Si un programme devait être optimisé de manière à augmenter sa vitesse et diminuer sa taille, le pipeline logiciel demanderait de faire des compromis. La demande d'optimisation du programme à exécuter est très forte, voire primordiale. Certaines optimisations sont réalisables par le programmeur. Ce dernier peut de lui-même chercher à réduire par exemple le nombre d'exécutions d'un corps de boucle, ou limiter le nombre de variables utilisées. L'utilisation d'allocation de mémoire dynamique peut permettre de minimiser l'espace mémoire nécessaire à l'exécution du programme. Toutes ces astuces de programmation sont d'autant plus efficaces que le programmeur connaît la plate-forme d'exécution et le compilateur utilisé pour générer le fichier binaire. Comme nous l'avons vu précédemment, la norme du langage *C* laisse au compilateur certains choix lors de la transformation du code source en code binaire. Dans certains cas très particuliers, le développeur averti prend le parti d'écrire lui-même certaines portions de code en langage assembleur. Il est alors certain d'utiliser toutes les ressources disponibles sur le système cible. Qu'il les utilise au mieux n'est pas à discuter ici, le fait est qu'il utilise une connaissance accrue de la programmation, de l'algorithmique et du matériel afin de tirer les meilleurs performances possibles du système final.

1.2 Les optimisations et le débogage

Les optimisations jouent un rôle majeur lors de la compilation de programmes embarqués. Elles interviennent à tous les niveaux, tant sur le code source que sur le programme binaire. Certaines sont considérées comme classiques car très utilisées [BGS94] et d'autres plus marginales car développées dans des contextes particuliers [DTLS04], destinées à des systèmes très spécifiques. La granularité ou niveau d'abstraction d'une optimisation dépend directement des modifications qu'elle cherche à apporter au programme. La suppression de portions de code mort a besoin d'analyser une représentation du programme contenant des groupes d'instructions, alors que l'ordonnancement d'instructions doit manipuler les instructions elles-mêmes. L'allocateur de registres a, quant à lui, besoin de manipuler une représentation du programme proposant une parfaite visibilité des registres.

Afin de modifier le programme, les optimisations n'ont pas toutes besoin des mêmes informations ni sur la syntaxe ni sur la sémantique du programme. En fonction de leurs besoins, plusieurs représentations de ce programme sont construites et manipulées. Cette analyse est appelée *analyse de flot de données*. Ses différentes composantes utiles aux optimisations sont présentées section 2.2.2.

De manière générale, les développeurs ne s'occupent pas d'automatiser l'optimisation de leur programme par le compilateur avant la livraison de leur produit. La méthode classique de débogage de programme est ainsi de développer sans optimisation de compilation durant la phase de mise au point et quand le produit est prêt à être livré, le compiler avec le maximum

d'optimisations. Cette méthode ne convient pas au cycle de développement dans le contexte des systèmes embarqués. Comme nous le verrons dans la section 2.3, cette méthode est utilisée car les outils actuels ont été développé pour une manipulation de programmes non optimisés. Ils exploites la structure des programmes sans tenir comptent des modifications apportées lors de leur optimisation. Ils ne permettent donc pas de contrevenir de manière satisfaisante aux problèmes du débogage de code optimisé. Depuis les prémisses de l'informatique et jusqu'à aujourd'hui, les développeurs n'ont jamais eu de débogueur capable de manipuler de manière satisfaisante un programme optimisé. Les ingénieurs ont toujours optimisé après la phase de développement d'un produit. Ils ont toujours corrigé les failles rapportées par les utilisateurs en travaillant sur le code non optimisé, puis ont livré la version du produit optimisé. Occasionnellement, un profileur est utilisé. Cet outil permet l'observation d'un programme lors de son exécution et est donc utile pour diagnostiquer un éventuel problème dans un programme. Le fait est qu'il ne permet pas la manipulation du programme lors de son exécution, il est donc difficile d'observer l'état exacte du programme à un point précis de son exécution. Nous sommes donc, avec un profileur, dans un registre différent d'outils proposant des mécanismes de contrôle tels que `gdb` décrit dans la section 2.6.

Le contexte des systèmes embarqués dans lequel nous nous situons n'est pas différent. L'activation des optimisations lors de l'intégration ou de la production d'un logiciel embarqué reste problématique. Le code source écrit par le programmeur est optimisé par le compilateur. Ce dernier lui apporte les modifications nécessaires à une exécution, si ce n'est optimale en tous cas améliorée. Mais en ce faisant, même s'il ne modifie pas la sémantique programme de manière visible en terme d'entrées/sorties, il modifie l'ordre des instructions. Modifier l'ordre des opérations exécutées modifie l'ordre d'exécution des instructions du code source telles que le développeur les avaient initialement écrites. Les implications de ces modifications sont présentées section 2.3.

1.3 Autres motivations

Le cycle de vie d'un programme implique une large part de support[3]. L'équipe responsable du support reçoit des rapports de bogues concernant le programme livré. À partir des descriptions d'exécutions et de traces d'exécution, le développeur doit localiser le(s) bogue(s) et le(s) corriger. Il doit donc observer les différents états du programme lors de son exécution en suivant la trace fournie. En d'autre mot, il doit faire le lien entre l'exécution de son programme optimisé et le code source initialement écrit. Un débogueur de code optimisé permettrait à l'utilisateur de faire ce travail.

Dans la phase de développement d'un programme embarqué, il arrive un moment où il faut le tester dans les conditions réelles, c'est-à-dire sur le matériel. Il est alors important d'y embarquer le programme tel qu'il sera livré, c'est-à-dire le programme optimisé. Mais si cette phase de test révèle des dysfonctionnements lors de l'exécution, le développeur doit là aussi faire un lien entre le programme exécuté et le code source. Voici une autre situation où un débogueur de code optimisé peut s'avérer utile.

[3]On appelle *support logiciel* le fait de répondre aux questions techniques et/ou fonctionnelles des utilisateurs, de prendre en compte leurs remarques ainsi que les rapports de bogues afin de produire des versions successives du logiciel.

1.4 Contributions

La solution logicielle au débogage de code optimisé ne doit pas dépendre d'une optimisation en particulier. Elle doit permettre de déboguer les optimisations implémentée dans le compilateur. La solution doit permettre d'arrêter l'exécution du programme à une adresse donnée puis de le relancer. Elle doit être capable de répondre aux questions suivante : *À quelle(s) instruction(s) du code source correspond telle adresse ? À quelle(s) adresse(s) du programme binaire correspond telle instruction ?* La solution doit être capable d'utiliser des informations sur l'exécution du programme et ainsi répondre aux questions de l'utilisateur sur l'état courant de l'exécution après chaque arrêt.

À partir d'une analyse de l'état de l'art du débogage de code optimisé (chapitre 2) et des outils fournis par STMicroelectronics (chapitre 3), nous avons cherché à développer une solution viable industriellement.

L'idée est de présenter au développeur l'exécution du programme optimisé de manière à ce qu'il comprenne aisément le lien avec le code source malgré les transformations appliquées par le compilateur. L'approche adoptée est de ne pas émuler l'exécution du programme non-optimisé à partir de l'exécution du programme optimisé. Le développeur de programmes embarqués a des connaissances que nous allons exploiter.

Notre parti est de montrer la réalité à l'utilisateur, de faire ce que P. Zellweger[Zel83] et J. Hennessy[Hen82] ont défini comme étant du *débogage non-transparent*. Il offre au développeur la possibilité de comprendre l'exécution de son programme. À notre connaissance, le seul travail effectué avec pour objectif de ne faire aucune récupération de valeur est celui sur le *Convex Debugger*, `CXdb` [BHS92], en 1992 par Brooks et Al, voir section 2.7.3.

Afin de tracer les modifications effectuées par le compilateur, nous proposons d'étiqueter chaque instruction du code source lors de la compilation. Il s'agit ensuite pour le compilateur d'ajouter ou de supprimer des étiquettes et de propager cette information tout au long de la compilation. Ajoutées au fichier binaire en tant qu'informations de débogage, elles sont ensuite utilisées par le débogueur afin de répondre sans erreurs aux interrogations de l'utilisateur.

Notons que, s'il existe des descriptions d'implémentation de système d'étiquetage, il n'existe pas à notre connaissance d'implémentation, ni même de description du principe de suivi des optimisations lors de la compilation. En terme de retour d'information à l'utilisateur, les travaux les plus avancés semblent être ceux de Tice[TG98] dont l'outil `Optview` modifie le code source en y ajoutant des commentaires ainsi qu'en y reportant certaines modifications de haut niveau apportées au code telles que la suppression de portions de code mort ou encore la propagation de constantes, voir section 2.7.6.

1.5 Organisation du manuscrit

Nous avons présenté les contraintes qu'impliquent les systèmes embarqués sur le développement de programmes. Que ce soit en terme d'utilisation d'espace mémoire, en terme de consommation CPU , en terme de consommation d'énergie ou en terme de sécurité et de sûreté, un logiciel embarqué est fortement contraint. Pour obtenir le maximum des performances, les chaînes de développement se dotent de compilateurs intégrant un nombre important d'optimisations.

Le débogage de code optimisé devient donc une étape incontournable du développement de logiciel embarqués. Cependant les développeurs évincent cette étape par manque de solution

adéquate. Une fois le programme optimisé, un débogueur classique n'est pas capable de faire correspondre le programme binaire et le code source sans erreurs. Nous proposons de modifier le compilateur afin que ce dernier maintienne des informations ciblées par optimisation tout au long du processus de compilation et d'optimisation. Le débogueur instrumenté pourra ensuite lire ces informations et ainsi permettre le contrôle de l'exécution du programme binaire.

Au **chapitre 2**, nous présentons le contexte scientifique dans lequel se situent nos travaux. Nous présentons d'abord les éléments de l'analyse de flot de données subséquemment utilisés. Nous présentons ensuite les deux grands problèmes du débogage de code optimisé et retraçons les étapes et travaux du domaine, de J. Hennessy en 1982 à nos jours. Nous concluons sur nos choix scientifiques.

Au **chapitre 3**, nous présentons le contexte industriel dans lequel se situent nos travaux. STMicroelectronics s'intéresse au développement et à la production de systèmes embarqués. Dans leurs départements R&D, sont conçus et développés les outils servant à nos expérimentations. Nous présentons le processeur MMDSP+, le compilateur `mmdspcc` et le débogueur. Nous décrivons leurs principales caractéristiques et concluons sur nos choix technologiques dépendant de ces outils.

Au **chapitre 4**, nous définissons l'environnement formel dans lequel nous décrivons ensuite l'instrumentation du compilateur et du débogueur afin de répondre à notre question initiale. Les instrumentations et portions d'algorithmes proposés sont déroulés sur trois exemples d'optimisations représentatifs chacun d'une classe d'optimisation : la suppression de sous-expressions communes, l'inversion de boucle et le pipeline logiciel.

Au **chapitre 5**, nous présentons les résultats de l'implémentation. La validation expérimentale est faite sur les trois optimisations utilisées au chapitre précédent puis sur un ensemble d'une dizaine d'optimisations pré-sélectionnées. Le programme compilé est un condensé de ce que l'on trouve dans la suite de tests utilisée pour la validation du compilateur produit par STMicroelectronics. Nous présentons également l'intégration d'une partie ces travaux aux outils de STMicroelectronics.

Nous terminerons par un résumé de la thèse et un récapitulatif des contributions. Nous verrons les perspectives envisagées et proposerons des idées de réalisations pour certaines d'entres elles.

Première partie

Le débogage dans la chaîne de développement

Chapitre 2

Contexte scientifique et état de l'art

Schrödinbug:
Bogue qui n'est pas découvert et non gênant pour les utilisateurs, mais qui apparaît après que quelqu'un a relu le code source ou utilisé le logiciel d'une façon non habituelle. À partir de ce moment-là, le programme ne fonctionne plus pour personne ... jusqu'à ce que le bogue soit corrigé.

Sommaire

2.1 Introduction

Le terme *bogue* est dérivé de l'anglais : *bug*. Thomas Edison l'utilisait déjà en 1870 pour désigner les défauts de systèmes mécaniques. En 1946, Grace Hopper a attribué une erreur dans le calculateur électromécanique Mark II à un papillon nocturne pris dans un relais, utilisant ainsi le mot *bug* pour désigner l'erreur.

En France, le terme *bogue* est recommandé par la Délégation générale à la langue française et aux langues de France (DGLF) depuis un arrêté paru au Journal officiel du 30 décembre 1983. Ce mot, qui se veut français, n'exprime pas une étymologie. C'est pourquoi peu de gens utilisent la version francisée. À cette époque le genre féminin était préconisé. Cependant à la fin de la décennie 1990, les dictionnaires tels que le *Nouveau petit Robert* et *Le Petit Larousse illustré* rapportaient l'usage de ce terme au masculin, sans doute sous l'influence québécoise où l'Office québécois de la langue française (OQLF) prônait depuis longtemps l'emploi du genre masculin. Le terme français a été popularisé avec le fameux bogue de l'an 2000. Désormais la DGLF recommande aussi le genre masculin pour ce mot.[1]

Nous pouvons distinguer quatre domaines de technologies d'optimisations directement liées aux contraintes énoncées précédemment : la recherche d'augmentation de vitesse lors de l'exécution, la recherche de diminution de taille d'utilisation mémoire, la recherche de consommation minimale d'énergie et la recherche de sécurité et de sûreté. Nombre d'entre elles sont effectuées par modification du programme lors de la compilation. Les systèmes embarqués étant fortement contraints, il n'est pas rare que le compilateur d'un tel système optimise fortement les fichiers destinés à y être exécutés. Comme nous allons le voir, une importante part de la littérature a été développée sur les optimisations quand peu d'écrits traitent du débogage de code optimisé. Le monde de l'industrie n'y échappe pas ; il n'est pas rare de lire dans les manuels d'un couple compilateur/débogueur que le premier optimise avec de notables bénéfices quand le second ne permet pas le débogage d'un programme optimisé sans crainte de réponses incohérentes. Les manuels de débogueurs tels que ceux fournis pas Microsoft, IBM ou HP, pour ne citer qu'eux, mentionnent le débogage de code optimisé plus ou moins de la même manière. Il faut déboguer le code non optimisé tant que faire ce peut et ne s'occuper du débogage de code optimisé qu'en ultime recours. Voici un exemple du manuel de Microsoft VisualC++ :

> En raison de (...) limitations, vous devez effectuer le débogage en utilisant si possible une version non optimisée de votre programme. Par défaut, l'optimisation est désactivée dans la configuration Debug d'un programme Visual C++ et activée dans la configuration Release.

Dans ce chapitre, nous présentons notre contexte scientifique et faisons un état de l'art du débogage de code optimisé. Afin de saisir la problématique réelle, nous parlerons d'abord de la compilation et des optimisations ainsi que des outils théoriques nécessaires à leur mise en place avant d'aborder les deux grands types de problèmes auxquels nous devrons répondre. Nous verrons ensuite les approches qui ont déjà été proposées ces trente dernières années, les solutions publiées et nous verrons à quelles questions spécifiques elles répondent ainsi que leurs

[1] `http://fr.wikipedia.org/wiki/Débogage`

restrictions. Nous terminerons par une description plus techniques de six débogueurs méritant notre attention avant de conclure.

2.2 La compilation et le débogage

La compilation est un mécanisme de traduction : un programme écrit dans un langage dit source va être traduit en un langage dit cible. Un compilateur met en application cette traduction par la succession d'au moins quatre étapes :

1. L'*analyse lexicale* : cette étape est la vérification que le programme source utilise bien le vocabulaire autorisé par le langage utilisé.
2. L'*analyse syntaxique* : cette étape est la vérification que le programme source respecte bien les règles de grammaire du langage dans lequel il a été écrit.
3. L'*analyse sémantique* : cette étape est la vérification que les phrases écrites en langage source ont du sens.
4. La *génération de code* : cette étape est la traduction proprement dite ; le programme source est traduit en langage cible. Il s'agit de la phase la plus conséquente du processus de compilation.

Dans certains cas, des étapes vont être ajoutées. Par exemple, des compilateurs vont chercher à optimiser le programme tel qu'initialement écrit. Des optimisations peuvent être apportées au programme avant et/ou après la génération de code. Dans l'architecture des compilateurs modernes, une constante demeure, les trois premières étapes restent le prologue obligatoire à toute réécriture du programme initial. Dans cette thèse, nous nous intéressons à la traduction de programmes *C* en un langage machine.

Les compilateurs proposent en principe une option qui génère une description du programme qui est ajoutée au fichier binaire. Ainsi, le débogueur peut répondre aux requêtes de l'utilisateur en lisant à la demande ces informations.

Toujours de manière générale, le débogage est l'art de trouver les bogues dans un exécutable. Pour ce faire, le débogueur fournit un ensemble de mécanismes permettant la manipulation des différents états du programme lors de son exécution. Le détail des manipulations possibles dépend du débogueur utilisé, nous aborderons ce point plus en détail à la section 2.6 en étudiant un débogueur considéré comme standard : `gdb`, le *GNU Project Debugger*. Le débogueur utilisé pour nos expérimentations est décrit plus avant dans la section 3.4.

2.2.1 La chaîne de développement et les formats d'information

Afin de permettre au débogueur de faire le lien entre le code source et le fichier exécutable, le compilateur récolte un ensemble d'informations appelées *informations de débogage* ; pour les lui transmettre, ces dernières sont attachées au fichier exécutable. Il existe plusieurs formats d'informations de débogage. Tous ces formats proposent des structures de données à l'expressivité variable. Étant donné que le fichier binaire est propre à un système d'exploitation et/ou un certain processeur, les informations de débogage doivent elles-mêmes être adaptées à la plateforme cible. Un format régulièrement utilisé est le format DWARF 2 [DWA] mais il en existe d'autres, nous donnerons les plus connus dans la section 2.6

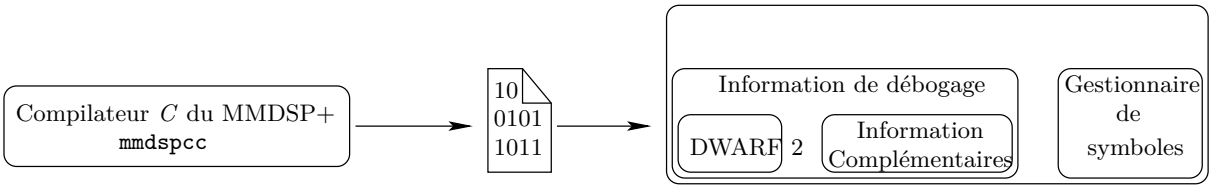


FIG. 2.1 – Le passage des informations de débogage se fait du compilateur vers le débogueur par l'intermédiaire d'une section du fichier exécutable ou d'un fichier à part.

2.2.2 Les optimisations à la compilation

Avant et/ou après la génération de code, le programme va être optimisé. Dans la suite de ce manuscrit, nous appellerons **optimiseur** le module du compilateur effectuant une certaine modification du programme. Un compilateur contient donc zéro, un ou plusieurs optimiseurs en plus des différents analyseurs (lexical, syntaxique et sémantique) et du générateur de code. L'optimisation de code consiste à modifier ce dernier afin de le rendre plus efficace sans altérer son exécution. L'altération d'exécution à laquelle nous faisons référence est celle observable à travers les sorties du programme ; en considérant le programme comme une boite noire. Les optimisations appliquées vont modifier l'intérieur de cette boite de manière à augmenter l'efficacité de certaines de ses fonctionnalités sans pour autant modifier l'état de ses sorties.. Il est primordial qu'une modification du programme n'altère pas la sémantique fonctionnelle. D. F. Bacon, S. L. Graham et O. J. Sharp [BGS94] définissent une transformation *légale*, c'est-à-dire une transformation qui ne modifie pas le programme au niveau de ses entrées/sorties :

Définition 2.1 (transformation légale) *Une transformation est dite* légale *si le programme original et le programme transformé produisent exactement les mêmes sorties pour tout couple d'exécutions ayant les mêmes données d'entrées et pour lesquelles chaque paire d'opération non déterministes donnent le même résultat.*

L'efficacité d'une optimisation est dépendante du contexte d'utilisation de l'exécutable. Il existe une catégorie de machines pour laquelle le gain en terme de consommation d'énergie est un critère important. Une importante recherche est faite notamment dans le monde des réseaux de capteurs, mais ceux-ci sont en dehors de notre sujet de recherche. La recherche en sécurité et sûreté n'est pas non plus discutée dans ce manuscrit.

L'efficacité qui nous intéresse est celle de vitesse d'exécution et d'utilisation mémoire. Il existe un certain nombre d'optimisations permettant d'obtenir de conséquentes améliorations dans ces deux cas. En voici quelques exemples :

- **Vitesse d'exécution**
  Plusieurs techniques permettent d'augmenter la vitesse d'exécution d'un programme : l'exploitation du parallélisme inhérent à une application, l'allocation de registres cherchant à diminuer le nombre d'accès mémoires en conservant certaines valeurs dans des registres, la suppression de sous-expressions communes (CSE) ou la propagation de constantes diminuent le nombre de calculs grâce à certaines formes de réécriture de certaines instructions.
- **Taille de code**
  De manière similaire, des techniques permettent de diminuer la taille utilisée en mémoire par le programme. L'allocation de registres permet, là aussi, de faire un gain substantiel. Le *cross-jumping* est une optimisation qui permet de réduire la taille du code machine en factorisant les portions de code redondantes dans un appel de fonction.

2.2.3 L'analyse des flots de données et de contrôle

Dans cette section, nous présentons les outils théoriques utiles à la compréhension de nos travaux. Le domaine qui nous intéresse est celui des optimisations lors de la compilation [AU77, ASU86, ALSU07]. Le programme y est représenté sous la forme d'un graphe orienté (définition 2.2 et figure 2.2) permettant l'exploitation de caractéristiques telles que les dépendances inter-instructions ou l'activité des variables.

Définition 2.2 (Graphe Orienté) *Un graphe orienté G est un couple $(\mathcal{S}, A)$ où*
- *$\mathcal{S}$ est un ensemble fini non vide de sommets,*
- *$A \subseteq \mathcal{S} \times \mathcal{S}$ est un ensemble d'arcs orienté entre sommets. Si $(s_i, s_j) \in A$, s_i est le prédécesseur de s_j et s_j le successeur de s_i.*

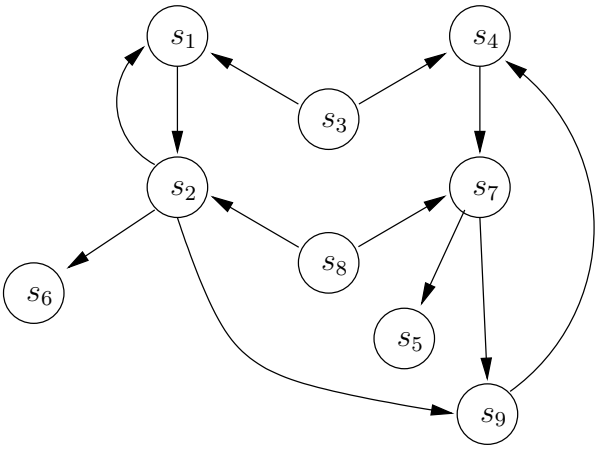


FIG. 2.2 – Exemple de graphe orienté composé de neuf sommets et douze arcs.

Dans la suite, nous utilisons plusieurs types de graphes orientés : les graphes de flot de contrôle et les graphes de dépendance de données. Il s'agit de graphes dont les sommets sont des instructions et dont les arcs expriment soit le flot de contrôle, soit des dépendances. Nous parlons de flot de contrôle entre deux instructions lorsque l'une est un branchement conditionnel vers l'autre ou quand l'une suit directement l'autre.

Définition 2.3 (Graphe de flot de contrôle) *Le graphe de flot de contrôle **CFG** d'un programme est un graphe orienté $G = (\mathcal{S}, A_{CFG}, s_e, s_s)$ où l'on distingue deux sommets particuliers s_e et s_s.*
- *$\mathcal{S}$ est l'ensemble des instructions du programme,*
- *A_{CFG} est l'ensemble des arcs (s_i, s_j) tels que s_i est un branchement vers s_j ou s_j suit directement s_i,*
- *$s_e \in \mathcal{S}$ est l'entrée du programme,*
- *$s_s \in \mathcal{S}$ est la sortie du programme.*

Définition 2.4 (Chemin) *Soit $G = (\mathcal{S}, A_{CFG}, s_e, s_s)$ un graphe de flot de contrôle. On appelle* chemin *de longueur $k > 1$ une séquence d'instructions $s_i \dots s_{i+k}$ telle que*

$$\forall j \in [i, i+k-1],\ (s_j, s_{j+1}) \in A_{CFG}$$

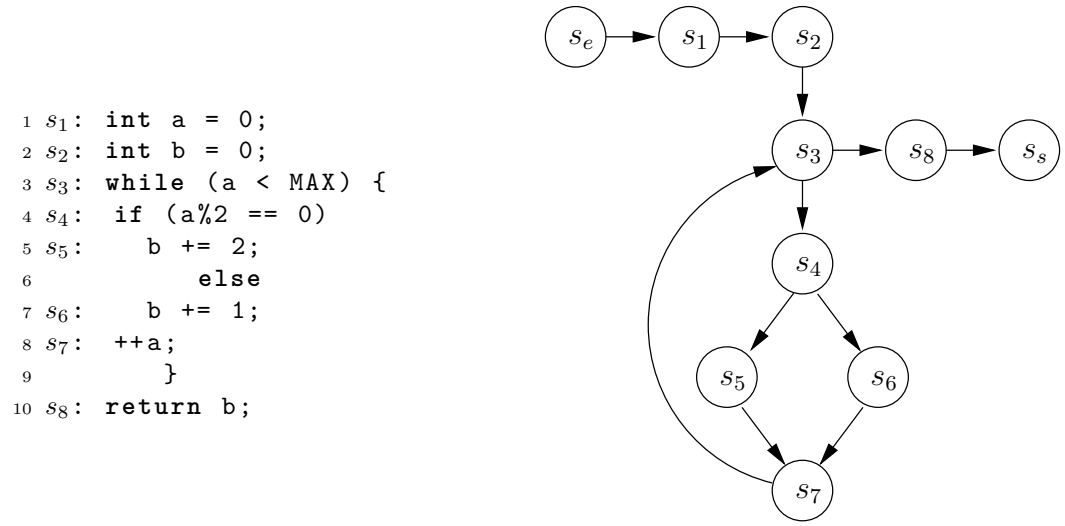


FIG. 2.3 – Exemple de programme et de son graphe de flot de contrôle.

Dans la suite, on considère un graphe de flot de contrôle $G = (\mathcal{S}, A_{CFG}, s_e, s_s)$. On note A^* la fermeture transitive de A_{CFG} telle que $A^* = \{(s_i, s_j) | s_i, s_j \in \mathcal{S} \times \mathcal{S} \wedge \exists$ un chemin de s_i à $s_j\}$.

Au cours de l'exécution d'un programme, une variable v peut avoir différentes valeurs. Diverses opérations telles que la manipulation de pointeurs ne permettent pas toujours de déterminer statiquement quelles instructions modifient la valeur de cette variable. Une définition d'une variable est une instruction qui affecte, ou peut affecter une valeur à cette variable. Soit s_i un sommet du graphe correspondant à une définition d. Cette définition est :

- *supprimée* lorsqu'une nouvelle valeur est affectée à v,
- *visible — reaching definition —* à une instruction s_k s'il existe un chemin $s_0 \dots s_k$ tel que $\exists i \in [0, k]$ où d n'est pas supprimée par l'instruction s_j, $j \in [i+1, k]$,

Une variable v est active — *live variable* — en une instruction s_i s'il existe un chemin $s_0 \dots s_k$ passant par s_i tel que :

- la valeur de v n'est pas supprimée par s_j, $j \in [i+1, k]$
- v est utilisée en s_k

Dans l'analyse des variables actives, on cherche à savoir si la valeur d'une variable v à une instruction s_i est susceptible d'être utilisée le long d'un chemin commençant par s_i.

On note $def(s_i)$, respectivement $ref(s_i)$, les ensembles de variables définies, respectivement utilisées, à l'instruction s_i. Le tableau 2.1 est une illustration de ces deux ensembles. Sur un chemin $s_e \dots s_i \dots s_j \dots s_k$ il existe quatre types de dépendances : les dépendances vraies, les anti-dépendances, les dépendances de sorties et les dépendances d'entrées.

Définition 2.5 (Dépendance vraie) *On appelle* dépendance vraie *entre deux instructions s_i et s_j le fait que s_j utilise une variable que s_i définie. On note A_{TD} l'ensemble des dépendances vraies d'un programme :*

$A_{TD} = \{(s_i, s_j) | (s_i, s_j) \in A^* \wedge def(s_i) \bigcap ref(s_j) \neq \emptyset\}$

L'exemple du tableau 2.1 a l'ensemble des dépendances vraies suivant :

$$A_{TD} = \{(s_1, s_2), (s_2, s_5), (s_2, s_7), (s_3, s_4), (s_4, s_5), (s_4, s_8)\}$$

s_i	Instruction	$def(s_i)$	$ref(s_i)$
s_1	`a = j + b`	$\{a\}$	$\{j, b\}$
s_2	`b = a + f`	$\{b\}$	$\{a, f\}$
s_3	`c = e + j`	$\{c\}$	$\{e, j\}$
s_4	`d = f + c`	$\{d\}$	$\{f, c\}$
s_5	`e = b + d`	$\{e\}$	$\{b, d\}$
s_6	`f = 42`	$\{f\}$	$\emptyset$
s_7	`g = b`	$\{g\}$	$\{b\}$
s_8	`h = d`	$\{h\}$	$\{d\}$
s_9	`j = 43`	$\{j\}$	$\emptyset$

TAB. 2.1 – Ensembles def et ref pour un programme donné.

Définition 2.6 (Anti-dépendance) *On appelle* anti-dépendance *entre deux instructions s_i et s_j le fait que s_i utilise une variable que s_j définie. On note A_{AD} l'ensemble des anti-dépendances d'un programme :*

$A_{AD} = \{(s_i, s_j) | (s_i, s_j) \in A^* \wedge def(s_j) \bigcap ref(s_i) \neq \emptyset\}$

L'exemple du tableau 2.1 a l'ensemble des anti-dépendances suivant :

$$A_{AD} = \{(s_4, s_6), (s_2, s_6), (s_1, s_2), (s_1, s_9), (s_3, s_9), (s_3, s_5)\}$$

Définition 2.7 (Dépendance de sortie) *On appelle* dépendance de sortie *entre deux instructions s_i et s_j le fait qu'elles définissent une même variable. On note A_{OD} l'ensemble des dépendances de sorties d'un programme :*

$A_{OD} = \{(s_i, s_j) | (s_i, s_j) \in A^* \wedge def(s_i) \bigcap def(s_j) \neq \emptyset\}$

L'exemple du tableau 2.1 ne contient pas de dépendances de sorties.

Définition 2.8 (Dépendance d'entrée) *On appelle* dépendance d'entrée *entre deux instructions s_i et s_j le fait qu'elles utilisent une même variable. On note A_{ID} l'ensemble des dépendances d'entrées d'un programme :*

$A_{ID} = \{(s_i, s_j) | (s_i, s_j) \in A^* \wedge ref(s_i) \bigcap ref(s_j) \neq \emptyset\}$

L'exemple du tableau 2.1 a l'ensemble des dépendances d'entrées suivant :

$$A_{ID} = \{(s_1, s_3), (s_1, s_5), (s_1, s_7), (s_2, s_4), (s_5, s_7), (s_5, s_8)\}$$

À partir de ces quatre définitions, nous pouvons définir un graphe de dépendance de données. On note V l'ensemble des variables d'un programme P et A_{DDG} l'ensemble des dépendances de ce programme : $A_{DDG} = A_{TD} \cup A_{AD} \cup A_{OD} \cup A_{ID}$

Définition 2.9 (Graphe de dépendances de données) *Le graphe de dépendances de données* ***DDG*** *est le graphe $G = (\mathcal{S}, A_{DDG}, s_e, s_s)$ d'un programme P où*

- *$\mathcal{S}$ est l'ensemble des sommets où chaque sommet $s \in \mathcal{S}$ est une instruction,*
- *$A_{DDG} = A_{TD} \cup A_{AD} \cup A_{OD} \cup A_{ID}$*

Il existe une forme nommée ***Static Single-Assignment* SSA** [CFR+91] utilisée dans certaines parties de compilateurs tels que `LLVM` [LA04] ou `gcc` [Sta99]. Dans la forme SSA, une variable ne peut être définie que par une seule instruction. Ceci présente les avantages suivant :

- Elle simplifie la mise en oeuvre des algorithmes d'analyse de flot de données et des optimisations sur la forme intermédiaire.
- La taille mémoire utilisée est moindre que dans le cas des chaînes UD/DU.
- Des utilisations non reliées d'une même variable se trouvent séparées.

Ces graphes servent aux algorithmes d'optimisations qui cherchent à réécrire des parties du programme sans introduire de bogues. Mais ces modifications ont des répercutions sur les informations relatives à ces portions de code. Dans la section suivante, nous présentons les problèmes introduits par le fait de ne pas avoir les informations de débogage appropriées.

2.3 Les problèmes relatifs au débogage de code optimisé

La structure habituelle d'un compilateur lui fait collecter la majeure partie des informations de débogage au début du processus de compilation. Ces informations font référence à certaines parties du programme (fichiers, lignes, variables, ...), mais ne sont, à ce moment précis, pas encore en liens explicite avec le programme binaire final. Le compilateur les ajoute au fichier binaire à la toute fin du processus de compilation (voir figure 2.4), la résolution des adresses est elle alors faite par l'éditeur de liens. Notons qu'il est possible de stocker ces informations dans un fichier séparé, mais là n'est pas notre propos. De ce fait, si le programme est modifié entre la collecte et le stockage, les informations de débogage présentes dans le fichier binaire sont en partie obsolètes, pour ne pas dire dans certains cas complètement erronées. Or le débogueur s'appuie sur ces informations afin de répondre aux requêtes du développeur. Si elles ne correspondent à aucune réalité du programme, alors les réponses seront tout simplement fausses, ce qui n'est pas le comportement souhaité pour un débogueur.

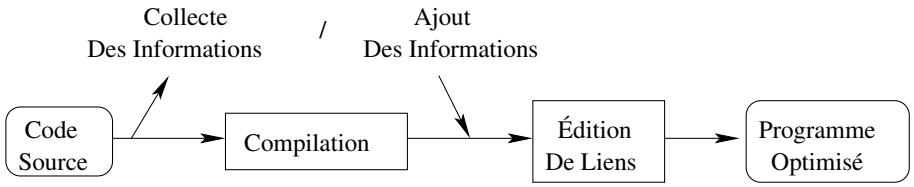


FIG. 2.4 – La collecte des informations de débogage se fait au début du processus de compilation, elles sont ensuite ajoutées au fichier binaire.

Nous allons ici exposer plus précisément les deux types de problèmes générés tels qu'ils ont été définis dans [Zel84b, CMR88]. Le premier concerne les données, et le second concerne la localité du code.

2.3.1 Les problèmes relatifs aux données

Le premier type de problème est connu sous le nom de *problèmes relatifs aux valeurs des données —data value problems*[Hen82, Zel83]. Ce sont les difficultés auxquelles il faut faire face lorsque l'on cherche à retourner la valeur d'une variable à la demande de l'utilisateur. Il peut

être décomposé en trois problèmes que nous allons expliciter : le problème de résidence des données, le problème de localisation des données et le problème de concordance des données.

2.3.1.1 Le problème de résidence des données

Afin d'économiser le nombre de registres utilisés, une optimisation consiste à détecter les variables inactives pour supprimer le code inaccessible et les instructions inutiles. Le **problème de résidence** — *residency problem* — est observable quand la valeur d'une variable n'est pas accessible. Prenons le listing de la figure 2.5 ; soit `f` et `g` deux fonctions sans effet de bord, supposons que `x` soit définie dans `f`. L'utilisateur peut réclamer la valeur de `x` à tout moment lors de l'exécution de la fonction `f`. Maintenant, disons que `x` n'est plus utilisée au-delà de la 4^{ieme} ligne. Pour économiser certaines ressources (mémoire, registres), le compilateur peut supprimer `x` au-delà de cette ligne, c'est-à-dire que `x` n'est plus du tout en mémoire, ni même dans un quelconque registre, la variable est perdue. Lors d'une session de débogage, l'utilisateur vouloir peut demander la valeur de `x` à la fin de l'exécution de `f`, mais dans le cas d'une telle optimisation, le débogueur ne sera pas capable de répondre correctement à l'utilisateur.

```
1       int f() {
2         int x;
3
4         b = g(x);
5         // x est inactive
6         return b;
7       }
```

FIG. 2.5 –
Illustration du problème de résidence de données. Ici, la valeur de la variable `x` devrait être accessible au delà de l'appel de la fonction `g`, en cas d'optimisation, `x` n'est potentiellement plus stockée nul part car inutilisée.

2.3.1.2 Le problème de localisation des données

Un autre problème, appelé **problème de localisation des données** — *data location problem* —, est observable quand une variable n'est pas à la place attendue c'est-à-dire dans le registre ou à l'adresse définie par les informations de débogage. Prenons les listings de la figure 2.6 ; la scalarisation (voir A.15 page 124) rend locale une variable globale en utilisant une variable temporaire, dans notre exemple la variable `tmp`. Nous n'avons donc pas la bonne valeur de `a` entre les lignes 5 et 10. Or, la valeur actualisée de `a` existe en mémoire ou dans un registre, il ne manque que l'information nous indiquant où.

2.3.1.3 Le problème de concordance des données

Le troisième et dernier problème est appelé le **problème de concordance** — *currency problem*. Prenons les listings de la figure 2.7 où `f` et `bar` sont deux fonctions sans effet de bord. La variable `tmp` n'est utilisée qu'une unique fois dans la boucle à la ligne 7 pour la même affectation. Le compilateur peut décider de déplacer cette affectation avant ou après la boucle (resp. *code hoisting* et *code sinking*) ; nous avons choisi le déplacement de portion de code hors

```
1  int f() {
2    int x;
3    int i;
4
5
6    a = a + x;
7    for (i=0; i<10; ++i) {
8      a = a + i;
9    }
10
11
12   return 0;
13 }
```

(a) Avant *scalarisation*

```
1  int f() {
2    int x;
3    int i;
4    int tmp;
5    tmp = a;
6    tmp = tmp + x;
7    for (i=0; i<10; ++i) {
8      tmp = tmp + i;
9    }
10   a = tmp;
11
12   return 0;
13 }
```

(b) Après *scalarisation*

FIG. 2.6 – Illustration du problème de localisation des données

```
1  int f() {
2    int i;
3    int tmp;
4
5    for (i=0; i<10; ++i) {
6
7      tmp = bar();
8    }
9
10   return 0;
11 }
```

(a) Avant *code sinking*

```
1  int f() {
2    int i;
3    int tmp;
4
5    for (i=0; i<10; ++i) {
6
7    }
8    tmp = bar();
9
10   return 0;
11 }
```

(b) Après *code sinking*

FIG. 2.7 – Illustration du problème de concordance

de la boucle vers le bas. Lors d'une session de débogage, l'utilisateur demande la valeur de `tmp` à l'intérieur de la boucle, mais comme l'instruction a été déplacée après la boucle, la valeur ne sera pas celle attendue par l'utilisateur. Dans le cas d'un déplacement avant la boucle, le problème resterait d'actualité car lors de la première itération, avant la première affectation de `tmp`, la valeur pourrait être différente de `bar()`.

Contrairement au problème de localisation des données, dans ce cas la valeur n'existe nulle part en mémoire.

2.3.2 Les problèmes relatifs à la localité du code

Le deuxième type de problème généré par l'optimisation de code est relatif à la localité du code —*code location problem*. Il est observable dans la correspondance entre les instructions du code source et les instructions du code optimisé. L'écriture de lignes de code contenant plusieurs instructions de langage source augmente la présence de problèmes relatifs à la localité du code. Pour la plupart, les instructions du code source ne sont pas atomiques quand elles sont traduites en langage machine. Pour chaque instruction du code source, un ensemble d'instructions machine sera généré par le compilateur. Il existe donc une première instruction machine corres-

pondant au début de la traduction de l'instruction source originale, et une instruction machine de fin correspondant à la fin d'exécution de cette même instruction source originale. Or les instructions machine sont dupliquées, combinées, déplacées, effacées, entrelacées avec d'autres instructions machine générées à partir d'autres instructions source. Il est donc difficile de déterminer à posteriori où dans le programme final une instruction source démarre et se termine. Par exemple, la suppression de code mort génère des relations du type un-zéro entre le code source et le programme optimisé. La suppression de sous-expressions communes et l'utilisation du parallélisme au niveau instruction (ILP) génèrent des relations du type plusieurs-un. Le fait de modifier l'ordre des instructions change leur ordre d'exécution.

Nous allons donc distinguer deux ordres, celui du le code source et celui du le programme optimisé.

Afin de manipuler le programme, le programmeur, par l'intermédiaire d'un débogueur, en suspend l'exécution. L'endroit choisi pour l'arrêt est appelé *point d'arrêt*. Dans le cas d'un programme non optimisé, l'arrêt se fait avant l'exécution de l'instruction sur laquelle est instancié le point d'arrêt.

Il existe deux concepts d'instanciation des points d'arrêt dans un programme optimisé : les points d'arrêt syntaxiques et les points d'arrêt sémantiques [Cop93, CMR88, Zel84a]. Ces points font référence, comme leur nom l'indique, à la syntaxe et à la sémantique du programme. Il n'existe pas, à notre connaissance, de définitions formelles de ces concepts.

Les points d'arrêt syntaxiques suivent le programme non optimisé, leur ordre est celui de leurs instructions source respectives. Après avoir instancié une série de points d'arrêt à partir du code source, une exécution pas à pas de l'exécutable ferait ressortir l'ordre des points d'arrêt dans le même ordre que celui des instructions source sur lesquelles ils auraient été instanciés.

Les points d'arrêt sémantiques suivent l'exécution effective du programme car leur ordre est dû à une réorganisation causée par les optimisations. Ainsi, en cas d'avancée pas à pas par l'utilisateur, il y a de fortes chances pour que l'ordre des points d'arrêt apparaisse comme anarchique aux yeux de l'utilisateur.

```
1 i = init;
2 while (i < n) {
3     m = c + x + y + n;
4     s = s + (m * (y + n));
5     i++;
6 }
```

(a) Code source, en langage *C*

```
1          ADD   r9 r7 r6      // S3: c + x
2          MOV   r10 r8        // S1
3          ADD   r12 r13 r11   // S3,S4: y + n
4          ADD   r16 r9 r12    // S3
5          MUL   r14 r16 r12   // S4: m * r12
6  TOP:    BGEQ  r10 r11 BOT   // S2
7          ADDI  r10 r10 1     // S5
8          ADD   r15 r15 r14   // S4: s + r14
9          B     TOP           // S6
10 BOT:    ...
```

(b) Code optimisé, en langage assembleur

FIG. 2.8 – Une illustration des points d'arrêt syntaxiques et sémantiques

La figure 2.8 et le tableau 2.2 illustrent ces concepts de points d'arrêt syntaxiques et sémantiques. Un programme du listing de la figure 2.8(a) écrit en *C* est compilé et optimisé, le résultat est donné en langage assembleur par le listing de la figure 2.8(b). Tout d'abord, le calcul de `m`, invariant de la boucle a été sorti de celle-ci (les lignes de langage assembleur 1, 3 et 4. qui correspondent respectivement aux calculs de `c + x`, `y + n` et enfin `m`). Supposons que le

Ligne de l'instruction source (figure 2.8(a))	Point d'arrêt syntaxique	Point d'arrêt sémantique (figure 2.8(b))
1	2	2
2	6	6
3	6	4
4	7 ou 8	8
5	7 ou 8	7
6	9	9

TAB. 2.2 – Liste des points d'arrêt relatifs au listing de la figure 2.8

programme n'entre pas dans la boucle, alors changer la valeur de `m` modifierait la sémantique du programme ; pour ce problème de sûreté, nous supposerons donc que `m` n'est pas utilisé plus loin dans le programme. La tableau 2.2 donne les instructions sur lesquelles se ferait l'instanciation des points d'arrêt syntaxiques et sémantiques. Les instructions implémentant l'instruction source de la ligne 3 sont les lignes assembleur 1, 3 et 4. Étant donné que la ligne 4 du programme optimisé calcule la valeur finale de `m` et la laisse dans le registre `r16`, il s'agit de la place logique pour le point d'arrêt sémantique correspondant. Le point d'arrêt syntaxique de cette même ligne 3 du code source ne peut être avant le point d'arrêt correspondant à la ligne 2 du code source, donc le premier endroit possible est l'entrée de boucle, la ligne 6 du programme assembleur.

Les avantages et les désavantages de ces deux concepts sont les suivants :

- Les points d'arrêt syntaxiques suivent le même ordre que les instructions du code source, mais une interrogation plus poussée de l'état du programme sur un tel point révèle en principe de grosses incohérences avec le code source.
- Les points d'arrêt sémantiques ne trahissent pas l'exécution du programme aux yeux de l'utilisateur, mais ce dernier ne peut plus considérer que deux instructions qui se suivent dans le source seront exécutées dans ce même ordre.

2.4 Les approches et méthodes de débogage

Optimiser un programme modifie la correspondance qui existe entre le code source et le programme optimisé et rend donc obsolètes les informations de débogage collectées selon l'approche classique. Si nous faisions un suivi instruction par instruction du programme en suivant le code source, nous serions perdus. La première question que nous devons alors nous poser est la suivante : *devons-nous donner à l'utilisateur l'illusion que son programme n'a pas été optimisé afin qu'il puisse en suivre l'exécution à travers le code source ?* En d'autres mots, doit-il y avoir une différence lors du débogage de deux versions d'un même programme : optimisé et non optimisé ?

Le débogueur est un outil qui se positionne entre l'utilisateur et son programme de manière à en faciliter l'observation. Il existe alors ce que l'utilisateur voit du débogueur et la manière dont le débogueur gère le programme. Dans la section suivante nous définissons ces différents comportements.

2.4.1 Approche transparente ou non-transparente

Afin de gérer les retours du débogueur aux utilisateurs, il existe deux approches radicalement différentes dénommées *transparente* et *non-transparente*. L'approche **transparente** donne l'illusion au développeur que son programme a toujours exactement le même comportement, qu'il soit optimisé ou non. Par opposition, l'approche **non-transparente**, ou exécution correcte, révèle au développeur le comportement exact de chaque version de son programme.

Ces définitions ont été publiées en 1983 par Polle T. Zellwegger [Zel83] qui a travaillé sur deux optimisations : l'*inlining*[2] et le *cross-jumping*[3]. P. T. Zellweger a développé un prototype, `Navigator`, pour le débogage de programmes écrits en *Cedar*, un langage à la *ALGOL*.

D'une manière générale, le niveau de transparence de débogage est directement lié à la *visibilité* des optimisations, et donc à la compréhension du comportement du programme. Un débogage en transparence pure dissimule l'intégralité des optimisations et donc masque le réel comportement du programme, mais permet au développeur un raisonnement bien plus facile sur le code source. D'un autre côté, un débogage purement non-transparent rend plus aisé la compréhension de l'exécution du programme mais demande au développeur un effort d'adaptation par rapport au code source qui n'est plus directement lié à l'exécutable. C. Tice représente ces relations par un schéma que nous avons reproduit en figure 2.9.

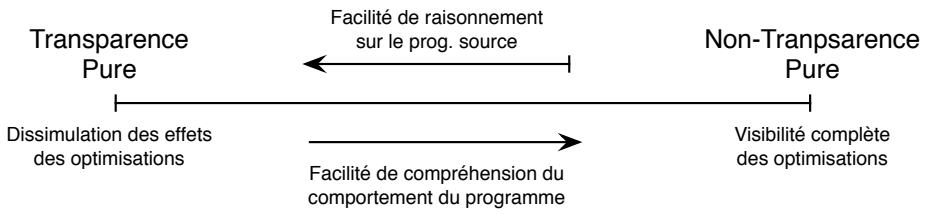


FIG. 2.9 – Continuum de transparence

2.4.2 Manipulation intrusive ou non-intrusive

Un débogueur doit permettre la manipulation du programme lors de son exécution. Afin de pouvoir inspecter l'état de l'exécutable, le débogueur peut faire appel à divers mécanismes. Ils peuvent se classer en deux catégories : intrusifs et non-intrusifs. Un débogueur est dit **intrusif** lorsqu'il modifie le comportement du programme. Par opposition, un débogueur est dit **non-intrusif** lorsqu'il ne modifie pas l'exécutable sauf pour le contrôle des points d'arrêt.

2.5 Chronologique des travaux antérieurs

2.5.1 Les années 80

J. Hennessy [Hen82] a été le premier à définir les notions de variables non-courantes et variables compromises — resp. *non-current variables* et *endangered variables*.

[2]L'inlining consiste à remplacer un appel de fonction par une copie de la fonction elle-même.

[3]Le cross-jumping consiste à remplacer une portion de code par un appel de fonction ayant le même comportement. Cette optimisation correspond à un gain en terme d'utilisation mémoire par l'intermédiaire d'un *jump*.

On appelle **variable non-courante** une variable dont la valeur, à un instant donné, n'est pas égale à celle qu'elle aurait eu au même endroit de l'exécution du même programme non optimisé, une variable dont les valeurs seront incorrectes si elles sont examinées à certains points intermédiaires d'un calcul. On appelle **variable compromise** une variable dont la valeur dépend du chemin emprunté dans le flot de contrôle. Soit v, est une variable à un point p dans le programme :

- v n'est compromise dans aucun chemin, alors sa valeur est la valeur dite courante.
- v est compromise dans tous les chemins, alors sa valeur est non-courante.
- v est compromise dans certains chemins, alors sa valeur n'est erronée que si le chemin le plus récent était compromis.

J. Hennessy considère le débogage de programmes écrits dans un sous-ensemble du langage Pascal. Il présente la première méthode permettant de retrouver les valeurs attendues de variables non-courantes par l'utilisation de techniques impliquant :

- le *déroulage* du programme débogué :
  le principe de cette technique est de continuer plus avant l'exécution du programme,
- l'*enroulage* du programme débogué :
  Le but de ce mécanisme est de revenir en arrière dans l'exécution du programme.

Les idées présentées ont servi de fondements à la recherche dans ce domaine. D. Wall, A. Srivastava et F .Templin [WST85] ou encore Copperman et McDowell [CM93] ont plus tard revu l'algorithme original de J. Hennessy afin de l'adapter à certaines technologies de compilation. Nous ne détaillerons pas ici le contenu de ce papier, mais il est important de garder en tête que ces trois articles [WST85, CM93, Hen82] sont des références pour ce qui est de la résolution des problèmes relatifs aux données dans le cas de programmes optimisés.

P. T. Zellweger [Zel83, Zel84b, Zel84a] a rédigé les premières définitions de *transparence* et de *non-transparence*. Ses travaux ont également permis l'identification des deux grands problèmes que nous avons décrit aux sections 2.3.1 et 2.3.2. Son outil, `Navigator`, est décrit dans la section 2.7.1.

D. S. Coutant, S. Meloy et M. Ruscetta [CMR88] ont proposé ce qu'ils ont appelé *"Une approche pratique pour déboguer un programme globalement optimisé"*[4]. Leur outil, DOC (voir la section 2.7.2), est un prototype du compilateur et du débogueur pour HP900 Series 800. Il aborde pratiquement exclusivement les problèmes relatifs aux données. Le compilateur effectue les optimisations au niveau assembleur. Cet outil ne permet que quatre optimisations : la promotion de variable en registre, l'assignation de registres, la suppression de variable de boucle et l'ordonnancement d'instructions.

Leur solution au problème de localité du code est d'étiqueter la première instruction de chaque ensemble d'instructions généré depuis une instruction source. Si la première instruction est déplacée ou effacée, l'étiquette est déplacée à l'instruction suivante. Cette solution permet d'avoir un ordre identique de points d'arrêt syntaxiques et sémantiques uniquement si les instructions ne sont pas déplacées au delà des limites d'autres instructions. De plus, les informations de débogage ne maintiennent pas l'intégralité des correspondances entre les instructions source et les instructions machine. Certaines de ces dernières ne sont plus présentes dans les informations de débogage.

Leur solution aux problèmes relatifs aux données est de fournir les informations de portée au débogueur. La structure des informations de portée des données est la suivante : chaque

[4] *A Practical Approach To Source-Level Debugging of Globally Optimized Code*

variable est décrite comme une paire composée d'un type et d'un lieu. Le type est un *registre* ou un *contenu mémoire*, et, dépendant de ce type, un lieu pouvant être un registre, une adresse ou une valeur. Le but annoncé des auteurs est de ne pas fourvoyer l'utilisateur ; si le débogueur n'est pas sûr d'une valeur, alors le débogueur l'affiche à l'utilisateur. Ils n'abordent aucune récupération de valeur sauf dans le cas des variables de boucle.

Selon *Compiler Technical Overview* [HP04] écrit en 2004 pour `HP-UX` :

> HP has significantly enhanced performance of code compiled for debugging by providing `+O1` level of optimization by default. Optimizations performed at `+O1` include common sub-expression elimination, constant propagation, load store elimination, copy elimination, register allocation and restricted basic block scheduling. Care has been taken to ensure that the program can still be debugged correctly ; that is, that breakpoints are at expected places and variables have expected values at breakpoints corresponding to source lines.

Les optimisations traitées n'entrelacent pas les instructions. Par conséquent leur mécanisme d'instanciation de points d'arrêt fonctionne sans tromper l'utilisateur sur leur ordre dans le programme optimisé.

G. Brooks, G. J. Hansen et S. Simmons [BHS92] ont eu pour but de donner un retour visuel lors du débogage de code optimisé. Ils ont conçu le Convex Debugger, CXdb (voir 2.7.3), un débogueur qui suit une approche non-transparente en surlignant diverses expressions dans le code source au fur et à mesure de l'exécution du programme. Aucune explication n'est fournie à l'utilisateur, ceci implique donc une demande de connaissance à l'utilisateur allant au delà de celle simple des optimisations, mais également des effets de leurs interactions.

2.5.2 Les années 90

Dans son rapport de master [Coo92], C. Lyle propose d'exposer les optimisations à l'utilisateur en affichant une version modifiée du source où les opérations auraient été réordonnées et/ou éliminées afin de coller au plus près de l'ordonnancement final des opérations tel que dans le fichier exécutable. Ce rapport focalise principalement sur l'ordonnancement d'opérations pour les processeurs VLIW. Aucune implémentation n'est proposée. Pour les transformations de source à source, cette approche a été plus tard utilisée par C. Tice [TG98] par son outil `Optview`.

R. Wismüller [Wis94] a proposé un algorithme pour la résolution d'une partie des problèmes relatifs aux données. Basée sur une analyse du flot de données, sa solution détermine si la valeur d'une variable est courante ou pas, mais aucune solution n'est fournie afin de retrouver ces valeurs.

Il a également listé les principales caractéristiques d'une solution aux problèmes des données et en a tiré des conclusions :

- **Indépendance des détails d'optimisation**, car les changements peuvent être conséquents sur les blocs de base. R. Wismüller propose une correspondance explicite entre le graphe de flot du code source et celui du code optimisé.
- **Correspondance de point d'arrêt arbitraire**, car les points d'arrêt syntaxiques ne sont ni uniques (à cause de la duplication de code) ni nécessairement les meilleurs endroits pour arrêter le programme.

- **Utilisation d'information d'exécution**, car, pour déterminer l'état d'une variable (courante ou non), le débogueur a besoin d'information sur le chemin afin de différencier deux instances d'une même instruction.
- **Manipulation d'affectations indirectes**, car les programmes font souvent utilisation de pointeurs. L'utilisation des pointeurs est en général une source de problèmes lors de l'analyse de programmes.
- **Garantie d'état de variable par le compilateur**. La plupart des compilateurs voient les affectations par les pointeurs comme une utilisation implicite de toutes les variables auxquelles le pointeur peut accéder. Ces variables doivent donc être courantes à la localisation donnée par l'affectation et ses instructions de stockage associées. Utiliser une telle information peut grandement accélérer l'analyse et affiner les résultats.

L'algorithme proposé sur ces conjectures n'essaie pas de retrouver les valeurs des variables. Il est basé sur la comparaison des deux graphes de flot de données, tous deux déroulés afin de différencier deux instances d'une même instruction.

Cette technique de déroulage de graphe est utilisée par Dhamdhere et Sankaranarayanan [DS98]. R. Wismüller est également le premier à utiliser une correspondance sur *équivalence sémantique* entre instructions, ce que C. Tice appellera plus tard les *Instructions clés* [TG00].

J. Gough, J. Ledermann et K. Elms [GLE94] ont décrit un modèle alternatif dans lequel l'information serait extraite en utilisant une version modifiée du compilateur pendant la session de débogage. Ils ne donnent aucune solution explicite ni au problème de la localité du code ni aux problèmes relatifs aux données.

Ils utilisent un compilateur pour leur propre langage interprété *DCode* qui est une représentation ad-hoc du programme. Quand un point d'arrêt est à instancier dans une fonction, le débogueur demande au compilateur de recompiler cette fonction en *DCode*. Quand l'exécution rentre dans la fonction, elle est dynamiquement redirigée vers la portion de code recompilée dans laquelle l'utilisateur peut réclamer toute sorte d'information.

Leur méthode permet donc de déboguer n'importe quel programme contenant des optimisations telles que du déplacement d'instructions tout en restant dans la même fonction. Par contre, elle est incapable de retrouver la valeur d'une variable globale ou de gérer l'inlining.

Trois ans plus tard, en 1997, une présentation d'implémentation plus détaillée a été publiée par K. Elms [Elm97]. Il décrit gpdb, un débogueur permettant la modification du programme lors de son exécution de manière à l'instrumenter, et au delà, le déboguer. Il a appelé sa technique *function interpretation*.

A. R. Adl-Tabatabai [AT96] a restreint son champ d'investigation à un sous-ensemble d'optimisations. Sa proposition de solution au problème relatif à la localité du code repose sur un double étiquetage en cas d'absence d'ordonnancement des instructions. De manière à tracer les modifications faites aux instructions, le compilateur maintient les deux tables de correspondance suivantes :

- une correspondance objet-vers-source
  Cette table de correspondance va d'une instruction du code machine vers une ou plusieurs instructions du code source de laquelle elle a été générée.
- une correspondance source-vers-objet
  Cette table de correspondance va d'une instruction du code source vers une ou plusieurs instructions du code machine.

Son sous-ensemble d'optimisations contient les optimiseurs suivants :

- Duplication de code
  déroulage de boucle, déroulage partiel — *loop peeling*, l'inlining de fonction, le clonage de fonction
- Suppression de code
  la suppression de code mort, l'élimination de redondance partielle
- Insertion de code
  l'élimination de redondance partielle, l'élimination de code partiellement mort, la réduction de force interne aux boucles
- Déplacement de code
  l'élimination de redondance partielle
- Ordonnancement de code
  il explique pourquoi, dans ce cas, l'utilisateur ne peut pas s'attendre à ce que l'ordre des points d'arrêt posés soit celui espéré.

Il considère également que l'affectation temporaire résultant d'une suppression de sous-expression commune ne provoque aucune modification observable au niveau de l'utilisateur. Donc le seul moyen d'arrêter le programme sur cette instruction est qu'il s'agisse d'un bogue du programme exécuté. Pour cette raison, et dans le cas où les instructions ne seraient pas ordonnancées (ceci inclut l'absence de pipeline logiciel), A. R. Adl-Tabatabai affirme que les ordres de points d'arrêt syntaxiques et sémantiques sont identiques.

Il propose également une solution pour détecter les variables non-résidentes dans la cas d'une allocation globale des registres [AT92]. Sa solution est basée sur une analyse du flot de données. Sa seconde proposition est de détecter les variables compromises [ATG93], il s'agit d'un ensemble d'algorithmes pour repérer les problèmes causés par un ordonnancement local des instructions et certaines optimisations locales (la fusion de registres, la suppression de code mort et le déplacement d'invariant de boucle avant la boucle — *code hoisting*). Les deux dernières, selon lui, capturent les effets d'optimisations globales qui compromettraient certaines variables. Il ajoute un algorithme pour partiellement retrouver de telles variables.

L-C. Wu et W-M. W. Hwu [WmWW98] abordent le problème de localité du code d'un point de vue utilisateur. Ils proposent une implémentation pour l'instanciation de points d'arrêt.

Ils font correspondre une instruction du code source avec un ensemble d'objets de type localité classifiés en quatre groupes :

- Les **Points d'ancrage** sont la base pour le calcul des points suivants. Ils sont maintenus par le compilateur.
- Les **Points d'interception** sont des localisations où le débogueur devrait suspendre l'exécution normale et démarrer la récupération de valeurs par l'utilisation de techniques déjà utilisées par J. Hennessy : le déroulage de l'exécution.
- Les **Points de fin** déterminent où le débogueur devrait arrêter la récupération en avant et commencer à prendre les requêtes utilisateur
- Les **Points de sortie** situent où le débogueur détermine si un point d'arrêt instancié au niveau source doit être suivi.

Les trois derniers peuvent être calculés soit par le compilateur, soit par le débogueur, toujours à partir du point d'ancrage. S'ils sont déterminés par le débogueur, le compilateur doit tout de même fournir des informations sur l'ordre des instructions [WMP+99]. Une implémentation de cette technique a été réalisée ; les auteurs ont modifié un compilateur existant et l'ont testé. Cette méthode s'est révélée inefficace lors de transformations de boucles qui superposent ou entrelacent des instances d'une même instruction (le pipeline logiciel par exemple).

2.5.3 C. Tice et C. Jaramillo

L'approche de C. Tice est basée sur le fait que le programmeur peut vouloir connaître les modifications qui ont été apportées à son code pendant les optimisations [Tic99]. Elle a développé un outil `Optview` [TG98] (voir 2.7.6) qui affiche certaines optimisations à l'utilisateur. Son outil affiche une version modifiée du code source, des commentaires y sont ajoutés quand c'est possible. Sa principale contribution est le concept d'*instruction clé* qui permet à un débogueur d'instancier des points d'arrêt depuis le code source (voir 4.2.2). De manière à rendre ce mécanisme efficace, elle définit pour chaque instruction source quelle instruction englobe au plus près la sémantique de cette instruction.

Elle aborde aussi le principe de récupération de valeurs en modifiant la proposition de D. S. Coutant, S. Meloy et M. Ruscetta. Elle propose l'instanciation de points d'arrêt cachés de manière à permettre au débogueur de récupérer les valeurs possiblement manquantes plus tard. Mais par soucis d'efficacité d'exécution, elle limite cette modification aux fonctions dans lesquelles l'utilisateur a déjà posé un point d'arrêt.

Elle donne les spécifications de son outil et explique comment elle modifie le compilateur *C* MIPS Pro 7.2 afin qu'il génère les informations de débogage dont elle a besoin.

C. Jaramillo [Jar00] propose une solution pour le débogage transparent de code optimisé. Son principal objectif est de cacher chaque optimisation à l'utilisateur en utilisant au maximum le mécanisme de points d'arrêt cachés ainsi qu'un ensemble de tables de correspondance permettant le calcul de valeurs et le calcul de portées.

Elle a développé un système de correspondance entre le code source et le code optimisé basé sur un étiquetage multiple. De là, elle propose également une méthode appelée *comparison checking* [JGS02, JGS99]. Il s'agit de comparer les valeurs calculées par le programme non optimisé et par une version du même programme optimisé pour des valeurs d'entrée identiques. C. Jaramillo montre comment cela peut aider lors de l'écriture d'optimiseurs de code au niveau source ainsi que que lors de l'écriture d'un allocateur de registres.

Elle propose un outil, `FULLDOC` (voir 2.7.5), où l'approche est dite transparente, c'est-à-dire cache les optimisations. Par exemple, lorsque l'utilisateur demande la valeur d'une variable dont l'instruction d'affectation est notée comme ayant été déplacée en aval de l'exécution du programme, c'est-à-dire sa valeur actuelle est celle précédent la valeur attendue, alors `FULLDOC` sauvegarde l'état du programme, déroule ce dernier jusqu'à atteindre le point d'affectation de la nouvelle valeur, la retourne à l'utilisateur, puis ré-enroule le programme jusqu'au point d'arrêt. Le résultat, est une augmentation drastique du temps d'exécution lors de la session de débogage.

2.6 Un débogueur standard : le GNU DeBugger `gdb`

Chez la majeure partie des débogueurs disponibles, les mêmes caractéristiques et les mêmes types de commandes permettent la manipulation du fichier exécutable. Le GNU[5] debugger [SPS+02] est un standard parmi les outils de développement. L'intérêt de le présenter ici est qu'il propose les principales caractéristiques des débogueurs que l'on trouve sur le marché :

- démarrer le programme,
- faire s'arrêter le programme en spécifiant les conditions
- examiner ce qui s'est passé une fois que le programme s'est arrêté

[5] `http://www.gnu.org/`

· changer des choses dans le programme de manière à pouvoir expérimenter en corrigeant un bogue et ainsi pouvoir s'occuper du bogue suivant

Le programme peut être écrit en *Ada*, *C*, *C++*, *Objective-C*, *Pascal* et encore d'autres langages. Ces programmes peuvent être exécutés sur la même machine que `gdb` (exécution dite *native*) ou sur une autre machine (exécution dite *distante*). `gdb` peut s'exécuter sur la plupart des systèmes d'exploitation comme UNIX et MS-Windows.

`gdb` utilise le concept d'**étage de la pile d'appels** — *frame* — pour garder une trace des appels de fonctions. L'implémentation a été motivée par le besoin de support des informations d'appels de fonctions du format DWARF [DWA].

L'instanciation des **points d'arrêt** est faite de deux manières. Les deux s'apparentent au remplacement d'une instruction par une autre qui provoquera la levée d'une exception. Cette dernière sera alors capturée, l'instruction originale sera remise en place et la main rendue à l'utilisateur afin qu'il puisse interroger l'état courant du programme. Les deux mécanismes diffèrent sur le remplacement et la capture de l'exception. Certains processeurs sont capables de tels arrêts de programme et gèrent eux-mêmes le remplacement et la capture d'exceptions. Quand le processeur a atteint son nombre maximum de points d'arrêt instanciables ou qu'il ne propose pas de tel mécanisme, c'est le débogueur qui va remplacer l'instruction d'instanciation par une autre qui lèvera une exception. Leurs noms sont le point d'arrêt matériel et le point d'arrêt logiciel — *hardware and software breakpoints*.

Il existe une troisième possibilité qui est que le matériel cible fournisse une capacité quelconque d'instanciation de points d'arrêt. Par exemple, un *moniteur ROM* (ou *bootstrap program*) peut avoir son propre mécanisme de point d'arrêt logiciel. Bien que n'étant pas à proprement parler des points d'arrêt matériels, du point de vue de `gdb`, le fonctionnement est identique ; il instancie le point d'arrêt et attend un retour.

2.7 Six débogueurs de code optimisé

Il a existé à notre connaissance six débogueurs permettant le débogage de code optimisé. Industriels ou académiques, ces outils abordent notre problème sous certaines conditions que nous détaillons ici.

2.7.1 `Navigator`

Le premier outil connu dans le domaine du débogage de code optimisé est celui développé par Polle T. Zellweger [Zel84a] pour les besoins de son doctorat. `Navigator` propose le débogage de programme écrits en *Cedar*, un langage à la *ALGOL*. Comme nous l'avons déjà vu, les deux optimisations proposées sont l'inlining et le cross-jumping. Une solution est apportée au problème de l'horodatage des instructions, à savoir déterminer le chemin d'exécution le plus récent.

Il utilise deux tables de correspondance, l'une lui permettant un parcours du code source vers le code machine, l'autre lui permettant le parcours inverse, du code machine vers le code source. Ces deux tables permettent des correspondances du type un-plusieurs, plusieurs-un, plusieurs-plusieurs et un-un entre le code source et le code machine et sont toutes deux maintenues par le compilateur. Pendant la session de débogage, le débogueur utilise un mécanisme d'horodatage permettant la différenciation dynamique des chemins dans le flot de contrôle car les deux optimisations influençant la structure de ce dernier rendent l'instanciation de points

d'arrêt dépendant du chemin. Elle suggère l'utilisation de points d'arrêt cachés de manière à améliorer l'algorithme de J. Hennessy [Hen82] servant au calcul des valeurs de variables. Quant à l'instanciation de points d'arrêt, les ordres syntaxiques et sémantiques sont restés les mêmes car aucun entrelacement d'instructions machine n'a lieu lors de ces deux optimisations. Le mécanisme de point d'arrêt est resté celui qui consiste à faire s'arrêter le programme sur la première instruction machine correspondant à l'instruction source sur laquelle le point d'arrêt a été demandé.

2.7.2 `DOC` : Debug of Optimized Code

Développé par D. S. Coutant, S. Meloy et M. Ruscetta [CM90], `DOC` est un prototype du compilateur et du débogueur pour HP900 Series 800 développés dans le but de démontrer la faisabilité du débogage de code optimisé. Les optimisations qu'autorise `DOC` sont la promotion de variable en registre, l'assignation de registres, la suppression de variables de boucles et l'ordonnancement d'instructions. `DOC` peut détecter si une variable est non-résidente ou corrompue et avertir l'utilisateur le cas échéant. Par contre il ne tente aucune récupération de valeur. Afin de simplifier le problème du débogage de code optimisé, `DOC` fait deux simplifications :

- aucune optimisation globale telle que le *code hoisting*,
- l'instanciation de points d'arrêt ne peut se faire qu'à des points de contrôle prédéterminés.

Permettre une manipulation simple de l'état (demande de valeurs de variables, et arrêt sur des points de contrôle) peut simplifier l'implémentation du débogueur car le compilateur et le débogueur savent a priori quels sont les points sur lesquels un point d'arrêt pourra être instancié, et surtout, ils savent, encore a priori, grâce à un pré-calcul possible, quels sont les ensembles de variables corrompues ou non-résidentes à chaque point de contrôle. En l'occurrence, si le programme s'arrête, à cause d'une erreur interne par exemple, alors `DOC` n'est pas capable de donner l'ensemble des variables corrompues avec certitude. Le compilateur de `DOC` marque l'ensemble des instructions pouvant recevoir un point d'arrêt avant l'ordonnancement, puis calcule l'ensemble des variables corrompues pour chacun de ces points en repérant les instructions `store` déplacées par l'ordonnanceur au delà des bornes des instructions. Cette information est fournie au débogueur.

2.7.3 `CXdb`

Le Convex Debugger, `CXdb`, a été développé par G. Brooks, G. J. Hansen et S. Simmons [BS94] dans le cadre des laboratoires de recherche de Convex Computer Corporation. Il a été écrit afin de déboguer des programmes écrits en *C* et *FORTRAN*.

Le Convex Debugger, CXdb, est un débogueur qui suit une approche non-transparente en surlignant diverses expressions dans le code source au fur et à mesure de l'exécution du programme. Aucune explication n'est fournie à l'utilisateur, ceci implique une demande de connaissances à l'utilisateur allant au delà de celles simples des optimisations, mais également des effets de leurs interactions.

Les auteurs conviennent clairement de la grande difficulté pour un utilisateur de comprendre les retours donnés pour cet outil. Ils proposent d'investiguer plus avant les possibilités de retours qui amélioreraient la communication `CXdb`-développeur mais à notre connaissance, il n'existe pas de travaux relatifs à cette proposition.

2.7.4 COCA : la sémantique plutôt que la syntaxe

Développé par M. Ducassé [Duc99], COCA est un débogueur de programmes écrits en *C*. Il propose le débogage au niveau source non plus en suivant la syntaxe mais en suivant la sémantique du programme. Un interpréteur du langage *prolog* lui est attaché, ce qui permet de contrôler l'exécution du programme avec une interface scriptable s'inspirant de GNU/Emacs. Le mécanisme des points d'arrêt est basé sur des événements relatifs à des constructions du langage. Ces événements ont une sémantique, ce qui constitue la principale différence entre une approche de débogage traditionnelle, au niveau syntaxique, et COCA.

Sur le même concept, E. Jahier [JD99] a développé Opium-M, un débogueur pour programme écrit en langage *Mercury*.

2.7.5 FULLDOC : Full Debugging of Optimized Code

FULLDOC [JGS00] est un outil dédié au débogage transparent de code optimisé. Il a été développé par C. Jaramillo dans le cadre de sa thèse [Jar00]. FULLDOC utilise trois sources d'informations de débogage. La première est la correspondance entre les instructions du code source et les instructions du code optimisé. La seconde provient d'une analyse statique visant à rassembler l'information sur la reportabilité des valeurs. La troisième et dernière source est l'analyse dynamique effectuée pendant l'exécution.

FULLDOC est censé fournir une interface de debogage complètement transparente.

2.7.6 Optview

C. Tice a développé dans le cadre de sa thèse co-encadrée à SGI un outil Optview [TG98] qui affiche certaines optimisations à l'utilisateur. Son outil affiche une version modifiée du code source, des commentaires y sont ajoutés quand c'est possible. Il s'agit en fait d'un couple, Optview et Optdbx. Le premier est la partie ajoutée à un compilateur commercial déjà existant, MIPS Pro 7.2 C Compiler, tandis que Optdbx est le débogueur capable de lire les informations de débogage fournies par Optview.

Optview commence par lire le fichier contenant le code source afin de stocker ce dernier ligne par ligne dans un tableau, chaque ligne étant étiquetée espace, commentaire, déclaration directive de pré-processeur ou code exécutable. Il va ensuite réécrire toutes les constructions "multifonctionnelles" (entête de boucle for, expressions conditionnelles, opérateurs d'incrémentation et de décrémentation imbriqués dans d'autres instructions, etc ...). La troisième étape est de déterminer le nouvel ordre dans le code source optimisé. Ceci est fait de la manière suivante : pour chaque instruction du code source qui a une instruction correspondante dans le programme optimisé, une instruction clé est identifiée. Quand toutes les instructions clés sont identifiées, l'ordre d'exécution de ces instructions clés devient l'ordre des instructions dans le code optimisé. Optview parcourt alors le code source et met à jour les instructions dont la sémantique a été altérée par le réordonnancement. Un outil interne à SGI est utilisé pour effectuer la réécriture. Optview commente chaque ligne du code source optimisé qui est considéré comme du code mort, à savoir toute ligne du code source qui n'a pas d'instruction correspondante dans le programme optimisé. Le dernier parcours du code effectué par Optview sert à ajouter des commentaires pour expliquer la propagation de copies, l'élimination de sous-expressions communes et la propagation de constantes.

2.8 Conclusion et propositions

La littérature nous fournit pléthore de propositions pour déboguer les optimisations de haut-niveau. Toutes ces solutions restent incomplètes dans un contexte industriel car elles travaillent sur un nombre restreint d'optimisations. De plus, elles s'abstraient de toutes optimisations de bas-niveau.

Deux courants semblent avoir émergé depuis les premiers écrits de Hennessy [Hen82] sur le débogage de code optimisé : l'utilisation du graphe de flot de contrôle du programme optimisé, étiqueté ou non, et l'utilisation de ce dernier couplé au graphe de flot de contrôle du programme non optimisé. Dans les deux cas, le soucis est double : définir le plus certainement possible l'aspect courant ou non d'une valeur, et retrouver la valeur espérée par l'utilisateur. Sous jacent au problème de concordance des données, nous trouvons le problème de localisation des données, qui consiste, comme nous l'avons vu section 2.3.1, à déterminer avec précision dans quel registre ou espace mémoire se situe la valeur d'une variable donnée. Nous pouvons résumer la résolution du problème des données par une séquence de questions auxquelles il faut répondre : pour une variable donnée, la valeur de cette variable est-elle quelque part ? Si oui, où ? Cette valeur est-elle celle attendue ? Si non, peut-on retrouver la valeur espérée ? Nous noterons que la réponse à la seconde question, celle sur la localisation de la valeur d'une variable, toute parfaite qu'elle puisse être, ne nous permettrait pas à elle seule de proposer la modification des valeurs des variables dynamiquement à un niveau autre que technique lors du débogage. En effet, modifier la valeur d'une variable n'a de sens que si l'on connaît le contexte d'utilisation de cette variable. Or optimiser un programme revient à modifier le contexte dans lequel se trouve chaque variable, et ce à chaque instant. En d'autres termes, si à un quelconque instant l'utilisateur veut modifier la valeur d'une variable, c'est en fonction de ce qu'il observe, de ce qu'il sait, de ce qui a déjà été exécuté, il est donc important de fournir plus d'informations que simplement la valeur courante d'une variable. Prenons un exemple pour mieux saisir le "contexte" dont nous parlons. Soit une boucle dont le nombre d'itérations est impair, disons 9, et admettons que l'une des optimisations apportée à cette boucle est loop reversal, c'est-à-dire le compteur de boucle va de 9 à 1, et non plus de 1 à 9, le corps de la boucle est bien entendu modifié en conséquence. À la cinquième itération, l'utilisateur observe que le compteur de boucle est bien égal à 5, l'optimisation est donc transparente à ses yeux s'il se cantonne à cette valeur. Si nous lui permettons de changer la valeur de cette variable sans lui indiquer que la boucle a été inversée, il augmentera cette valeur pensant sortir plus rapidement de la boucle, ce qui est évidement faux. De la même manière, il mettra cette valeur à 1 pensant réinitialiser la boucle. Le contexte est non pas seulement les valeurs des variables, mais également le pourquoi de ces valeurs, quelles soient celles espérées ou non.

Le recoupement des différents travaux de Adl-Tabatabai [AT96], ou encore Brooks [BHS92] nous montre sans ambiguïté l'efficacité de l'utilisation d'un principe d'étiquetage simple des instructions lors de la compilation. Une implémentation d'un tel système existe, `CXdb`. En ajoutant à cela le concept d'instruction clé de Tice [TG00] qui permet l'instanciation de points d'arrêt, nous avons un débogueur capable de montrer à l'utilisateur la correspondance réelle qui existe entre le code source et le programme optimisé.

Ainsi, bien que différenciables, les deux grands types de problèmes du débogage de code optimisé, à savoir, le problème relatif à la localité du code et les problèmes relatifs aux données, sont intimement liés dans le contexte industriel dans lequel nous nous trouvons et que nous décrivons au chapitre suivant. La suite de nos travaux est le développement d'une solu-

tion non-transparente de débogage dans laquelle nous révélons au développeur le contexte de modification de son programme sans surcoût en terme d'exécution de la session de débogage. Les expérimentations du chapitre 5 seront faites sur des optimisations de haut comme de bas niveaux.

Chapitre 3

Contexte industriel

Mandelbug:
Bogue dont les causes sont si complexes que son comportement apparaît chaotique

Sommaire

3.1 Au sein de STMicroelectronics

Issue de la fusion de SGS Microelettronica (IT) et de Thomson Semiconducteurs (FR) en 1987, STMicroelectronics est parmi les leaders mondiaux des fabriquants de semi-conducteurs. En 2007, le nombre d'employés est approximativement de 50 000 répartis dans 16 unités de recherche et développement, 39 centres de design et applications, 15 sites de fabrication et 78 bureaux de vente, le tout dans 36 pays.

STMicroelectronics réalise notamment des processeurs destinés à des systèmes embarqués. Les outils logiciels permettant d'exploiter ces systèmes sont en partie développés sur les sites de Crolles et Grenoble. Pour la création de programmes s'exécutant sur de tels systèmes, l'équipe

CEC, *Compilation Expertise Center*, fournit les compilateurs. L'équipe IDTEC dans laquelle je suis intégré fournit le débogueur.

Notre expérimentation a été faite avec la chaîne d'outils Flexware [PS02] qui comprend un compilateur *C* pour le MMDSP+, un débogueur, `IDbug`, et un ensemble d'applications. Le débogueur a besoin d'informations telles que le nom des variables et des fonctions utilisées, ou encore la correspondance entre les lignes du code source et les instructions machine résultantes de la compilation. Ces informations de débogage sont générées par le compilateur pendant la compilation d'un programme, et ajoutées au programme exécutable final. L'ensemble d'applications de tests comprend plusieurs codecs (CODeur/DECodeur) de formats audio tels que le codec Enhanced Full Rate (EFR) du GSM wireless communication standard, g723.1, mp3 et AMR. L'équipe CEC développe et maintient un compilateur dont les programmes sont exécutables sur ce processeur. Nous le décrirons dans la section 3.3. L'équipe IDTEC développe et maintient un débogueur de programmes *C* et *C++* exécutables sur ce processeur. Nous le décrirons à la section 3.4.

3.2 Le processeur MMDSP+

Il existe plusieurs types de processeurs embarqués. Certains se différencient par le fait qu'ils fournissent des mécanismes de parallélisme au niveau instruction, ILP —*Instruction Level Parallelism*. Le type d'architecture dont nous parlerons par la suite est appelé ASP-VLIW — *Application Specific Processor - Very Long Instruction Word*. Un ASP est généralement conçu dans un but précis. Son jeu d'instructions contient un ensemble d'opérations visant à faciliter les tâches pour lesquelles il sera le plus souvent utilisé. Une instruction de processeur VLIW encode plusieurs opérations en parallèles ; plus précisément, une instruction encode au plus une opération pour chaque unité d'exécution du processeur. Par exemple, si le VLIW a cinq unités d'exécution, alors une instruction VLIW sera composée d'au maximum cinq opérations. Notons que ce maximum est théorique. La taille du VLIW est un choix de design, elle est indépendante du nombre d'unités fonctionnelles.

Cette capacité à exécuter plusieurs opérations simultanément va pousser le compilateur à profiter du parallélisme inhérent à une application pour accélérer son exécution en faisant usage au maximum de son VLIW.

Le MMDSP+ est un processeur de type DSP dont l'architecture est de type Harvard : la mémoire de données et la mémoire programme sont physiquement séparées. L'accès à chacune des deux mémoires s'effectue via deux bus distincts ce qui permet de paralléliser les accès au programme et les accès à la mémoire. C'est un ASP-VLIW composé d'une unité de contrôle et de trois unités d'exécution. Elles gèrent trois formes arithmétiques (linéaire, saturée sur 16 ou 24 bits, et étendue sur 40 ou 56 bits) et de quatre types d'adressage (*post increment/decrement, modulo, bit-reverse, index, linear*) :

- **L'unité de contrôle** (CU)
  La CU est un décodeur pipeliné sur deux niveaux permettant les branchements classiques (*jump, direct/indirect,, call, return*), les interruptions classiques (*goto/return from interrupt*) et les boucles matérielles sur trois niveaux d'imbrication.
- **L'unité de calcul des données** (DCU)
  La DCU est dédiée aux calculs d'entiers.
- **L'unité de calcul à virgule flottante** (FPU)
  La FPU est dédiée aux calculs à virgule flottante sur 32 bits.

· **L'unité de calcul d'adresses** (ACU)
L'ACU supporte divers modes d'adressage (*post increment/decrement, modulo, bit-reverse, index, linear*).

Les instructions du MMDSP+ ont une taille de 64 bits. Elles sont composées d'un ensemble de champs. Chacun d'eux correspond à une opération qui sera associée à une unité (table 3.1) :

Nom du champ	**FPDCU**	**XY**	**XMV**		**DCU or FPU**	**ACU**	**YMV or IMM**	**CU**
			XSRC	**XDST**				
Taille	1	1	6	6	19	7	16	8

TAB. 3.1 – Description d'une instruction du MMDSP+

· **FPDCU**
Il indique s'il s'agit d'une opération entière ou à virgule flottante. Selon, le champ FPDCU représente une opération de la FPU ou de la DCU.
· **XY**
S'il est à 0, il n'y a qu'un seul accès mémoire (Xbus). YMV est alors une valeur immédiate représentée sur 16 bits. S'il est à 1, il y deux accès mémoires (Xbus et Ybus).
· **XMV**
Ce champ représente le transfert sur le bus **Xbus**. La source XSRC est copiée dans XDST. Ce transfert peut être effectué entre tous les registres du DSP.
· **DCU** ou **FPU**
Ce champ décrit l'opération effectuée par l'unité arithmétique.
· **ACU**
Ce champ décrit le calcul effectué par l'unité d'adressage.
· **YMV**
Ce champ représente le transfert sur le bus **Ybus**. La donnée est lue en mémoire Y et transférée à la DCU ou à la FPU.
· **IMM**
Quand il n'y a pas de transfert sur le bus Ybus, le champs représente une valeur immédiate, sinon il représente le transfert sur le bus **Ybus**.
· **CU**
Il s'agit de l'opération effectuée par le contrôleur.

Tous ces champs ne sont pas indépendants. Par exemple quand le DSP exécute les instructions d'une sous-routine, il doit sauvegarder l'adresse de retour de la pile. Le contrôleur exécute un branchement à une adresse immédiate alors que les champs ACU et XMV indiquent que l'adresse de retour est écrite en mémoire.

3.3 Le compilateur *C* du MMDSP+

Le compilateur *C* du MMDSP+ [PS02] est basé sur la technologie FlexCC2 [BDG+02]. Il est composé d'un haut niveau (*frontend*) basé sur la technologie CoSy [ACE03] et d'un bas niveau (*backend*) dont la technologie appelée EliXir a été développée à STMicroelectronics [BDG+02, DTLS04].

3.3.1 L'architecture

CoSy est un environnement de construction de compilateurs. Il fournit un haut-niveau ANSI *C*, un ensemble d'optimisations et un générateur de générateurs de code. Son architecture est basée sur le concept de *moteur — engine*. Un moteur est un module qui optimise, transforme ou analyse une représentation intermédiaire appelée CCMIR— *Common CoSy Medium Intermediate Representation*. Le compilateur est ainsi une séquence de moteurs. L'accès à la re-

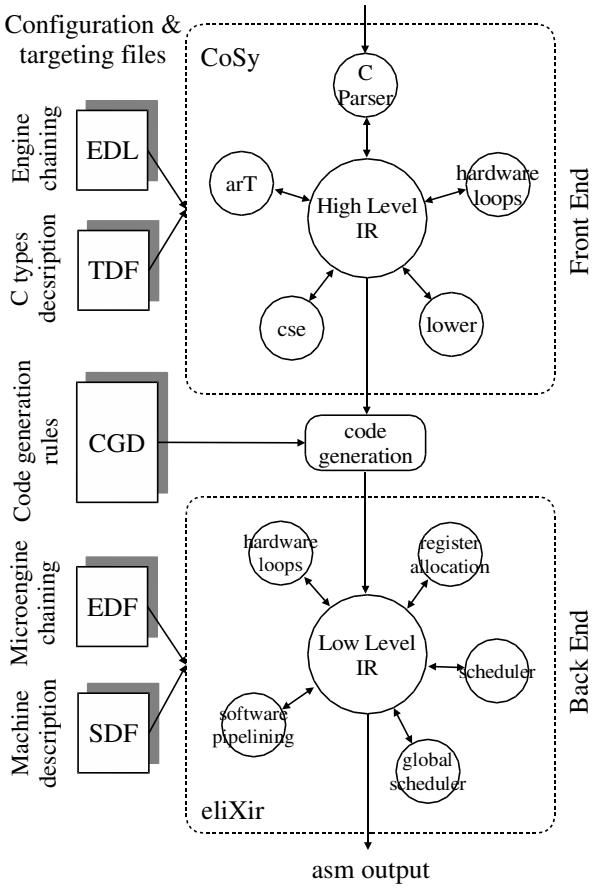


FIG. 3.1 – Architecture du compilateur *C* du MMDSP+ basé sur FlexCC2

présentation intermédiaire est géré par la *fSDL— Full Structure Definition Language*. Il s'agit d'une représentation extensible permettant la création de structures de données. Les fonctions de manipulation de ces structures sont ensuite générées automatiquement.

Parmi les optimisations, on trouvent :

- suppression de sous-expressions communes, propagation de copie, propagation de constantes (*common subexpression elimination*, *copy propagation*, *constant folding*).
- réduction de force, déroulage de boucle (*strength reduction*, *loop unrolling*).
- (*tail recursion*, *switch optimization*).

CoSy fournit également un ensemble de moteurs permettant d'abaisser[1] le code (*lowering the*

[1]Le verbe *abaisser* est emprunté à la subjectivité du niveau d'abstraction. Abaisser le code signifie baisser d'un ou plusieurs niveaux d'abstractions

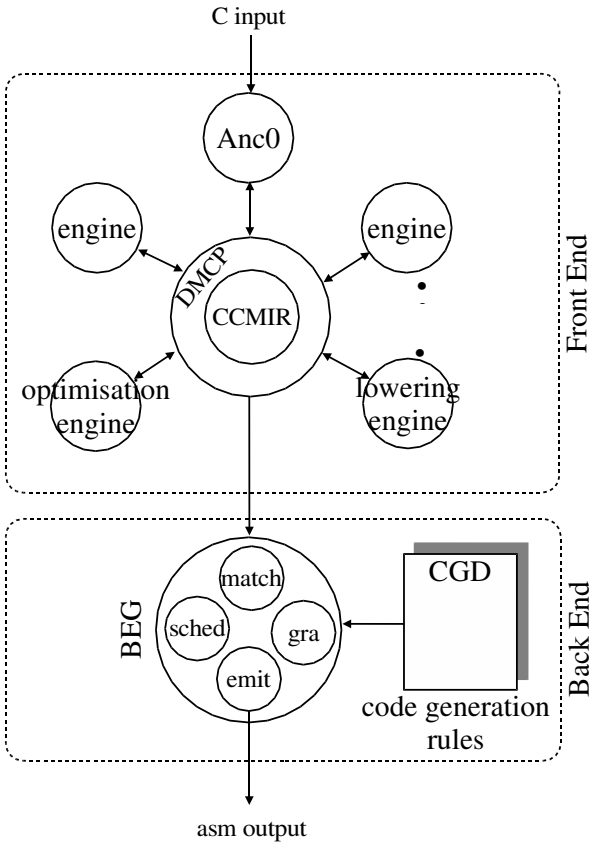


FIG. 3.2 – Architecture d'un compilateur CoSy

code) afin d'adapter la représentation intermédiaire aux spécificités du processeur cible.

Le compilateur *C* du MMDSP+ propose différents niveaux d'optimisation. Lors de la compilation du compilateur, un fichier de configuration est utilisé afin de fixer les différents niveaux d'optimisations que le compilateur pourra proposer et les optimisations correspondantes. La séquence des moteurs détermine l'ordre des optimisations. Chaque optimisation manipule le programme et renvoie ensuite une information caractérisant sa réussite ou son échec. Ce système d'entrée/sortie permet de créer un flux d'optimisation complexe car utilisant des mécanismes de branchements conditionnels.

Dans CoSy, à partir d'une description de générateur de code (CGD) décrivant les transformations partant de la CCMIR, BEG (le *back-end generator*) génère l'ensemble des moteurs effectuant la génération de code, l'ordonnancement et l'allocation de registres. BEG permet de faire le lien entre CoSy et EliXir. Dans le `mmdspcc`, seul le générateur de code est généré par BEG. L'ordonnanceur et l'allocateur de registres ont été réécrits et intégrés à EliXir.

EliXir est un bas-niveau propriétaire de STMicroelectronics développé pour pallier les déficiences de BEG en terme d'optimisation de bas-niveau.. Sa structure est donnée figure 3.3. Il

permet la manipulation d'une représentation intermédiaire proche du langage machine. EliXir repose sur un concept similaire aux moteurs de CoSy : les micro-moteurs — *microengines*. Le

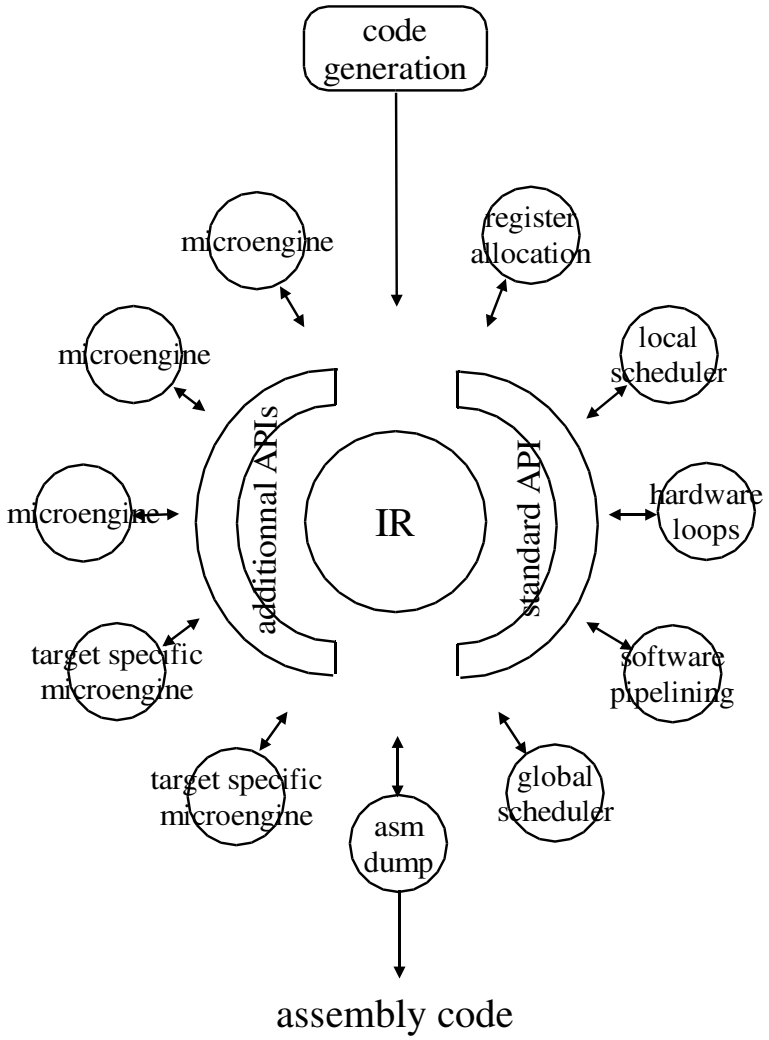


FIG. 3.3 – EliXir structure

but d'EliXir est de fournir des analyses et des optimisations basées sur des informations détaillées sur le processeur cible. Les principales optimisations à ce niveau sont l'allocation de registres, l'ordonnancement et des optimisations à lucarne — *peephole optimization*.

Le reciblage se fait grâce à un fichier de configuration du processeur cible contenant les informations suivantes :

- structure des registres,
- banques de mémoires,
- ressources matérielles de calcul ,
- la syntaxe de l'assembleur,
- la sémantique de contrôle,
- le flot de données.

Le modèle de processeur manipulé par EliXir a été étendu afin de proposer un meilleur support des architectures AS-DSPs et ASIPs. Les fichiers de registres composites, les boucles matérielles

et les espaces mémoire multiples en sont un exemple.

Les ordonnanceurs utilisent le modèle de parallélisme proposé par le processeur afin de réorganiser et compresser le code. Les deux principales contraintes sont alors l'encodage des instructions et l'utilisation des ressources matérielles. Les infrastructures de bas-niveau sont généralement basées sur des tableaux de réservation d'ordonnancement — *scheduling reservation tables* — peu flexibles augmentant fortement le temps de compilation. EliXir a été développé dans les deux directions suivantes :

- Dans le but d'utiliser la notion de *slot* VLIW proposé par C Eisenbeis, Z. Chamski et E. Rohou [ECR99]. Cette notion traite de l'encodage contraint des VLIW non symétriques : une opération ne peut pas aller dans n'importe quel unité du processeur.
- Dans le but d'utiliser le concept de *ressources stimulées*. Des stimuli sont utilisés afin de permettre des accès en parallèles aux unités fonctionnelles. Deux accès concurrents à une même ressource avec le même stimulus ne sont pas en conflit. Les ressources stimulées permettent de manipuler des opérandes inter-dépendantes lors de l'ordonnancement. Ce concept est particulièrement intéressant pour la gestion de modes d'adressages complexes lors de l'ordonnancement.

3.3.2 Son architecture et sa chaîne d'optimisations

L'approche de STMicroelectronics a été d'écrire des modules de haut-niveau par dessus CoSy et de compléter l'ensemble avec une infrastructure bas-niveau nommée EliXir. Un classement a été fait en fonction de trois critères, chaque optimisation faisant partie d'un des groupes suivants :

- le groupe des optimisations **supprimant des instructions**, la suppression de code mort en est un exemple.
- le groupe des optimisations **déplaçant des instructions**, le déplacement d'instructions invariantes hors de corps de boucle en est une illustration.
- le groupe des optimisations **réécrivant des instructions**, le remplacement d'accès tableau en arithmétique de pointeur en est un exemple.

3.3.3 Les optimisations de haut-niveau

Le compilateur *C* du MMDSP+ propose un ensemble complet d'optimisations [BDG+02]. Les optimisations standards sont fournies par CoSy. Les optimisations avancées ont été développées par STMicroelectronics.L'approche adoptée a été de s'appuyer sur l'infrastructure de CoSy autant que faire ce peut. Son architecture modulaire et l'extensibilité de sa représentation intermédiaire permettent une adaptation précise aux systèmes embarqués. Le développement des optimisations spécifiques à STMicroelectronics pour ses DSP a eu trois orientations :

- les transformations de tableaux en pointeurs (ArT et GarT),
- une utilisation efficace des ressources et du mode d'adressage du DSP,
- le support des boucles matérielles.

3.3.3.1 ArT

ArT [Gui99] — *ARray Transformation* — est un moteur qui cherche à remplacer les accès tableaux par des manipulations de pointeurs des modes d'adressage "pre/pos" incrémentés dans

les boucles. Le but annoncé est de mieux exploiter le type d'adressage et ainsi obtenir un gain de vitesse d'exécution. La transformation de tableaux en pointeurs améliore les performances du code initial, contenant boucles et tableaux, de 20% à 30% en moyenne, suivant le processeur cible. L'une des raisons de cette amélioration, outre la réduction des opérations dans le corps de boucles, est l'augmentation du taux de parallélisme au niveau instruction. Le flot de transformation de ArT pour une boucle est donné par la figure 3.4. La première étape est l'extraction les variables d'induction et l'analyse les expressions d'induction de la boucle. Une expression d'induction est une expression de la forme $base + \sum_k \alpha_k \times i_k + cte$ où *base* est une adresse de base, i_k sont les variables d'induction, et α_k et *cte* sont les invariants de boucle. Un ensemble de connivence — *Connivance sets* — regroupe les expressions d'induction qui sont associées à la même adresse de base et qui appartiennent à la même famille de variables d'induction. Ces derniers sont alors construits et utilisés afin d'affecter les expressions d'induction aux pointeurs.

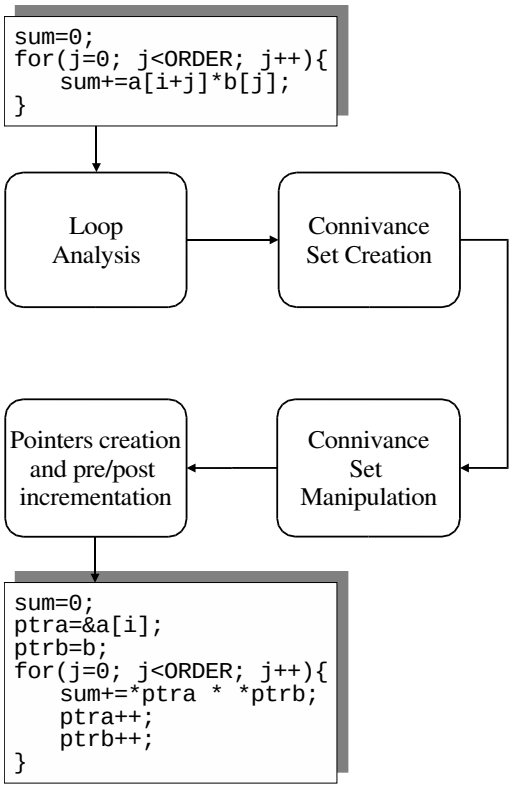


FIG. 3.4 – Flux de transformation ArT

Quand les ensembles de connivence sont formés, un unique pointeur est associé à chaque ensemble. Chaque expression d'induction peut alors être obtenue en utilisant un déplacement d'invariant à l'intérieur de la boucle en utilisant au mieux des schémas de pré- ou de post-modification. Une pré-modification initialise le pointeur à l'adresse de base et utilise un déplacement pour accéder à la donnée. Une post-modification initialise le pointeur sur l'adresse de la première donnée et utilise un déplacement de post-incrémentation/décrémentation afin de mettre à jour le pointeur au fur et à mesure du corps de boucle. ArT peut manipuler tout type de corps de boucle ainsi que les tableaux à plusieurs dimensions bien que les accès multi-dimensionnels ne soient pas toujours optimaux.

3.3.3.2 GarT

GarT est une extension de ArT qui permet une manipulation plus efficace de tableaux à plusieurs dimensions dans des boucles imbriquées. Alors que ArT manipule chaque niveau de boucle indépendamment, GarT pallie au principal défaut de ArT qui est que les tableaux à plusieurs dimensions dans une boucle imbriquée sont transformés en utilisant plusieurs pointeurs alors qu'il est souvent possible de n'en utiliser qu'un seul.

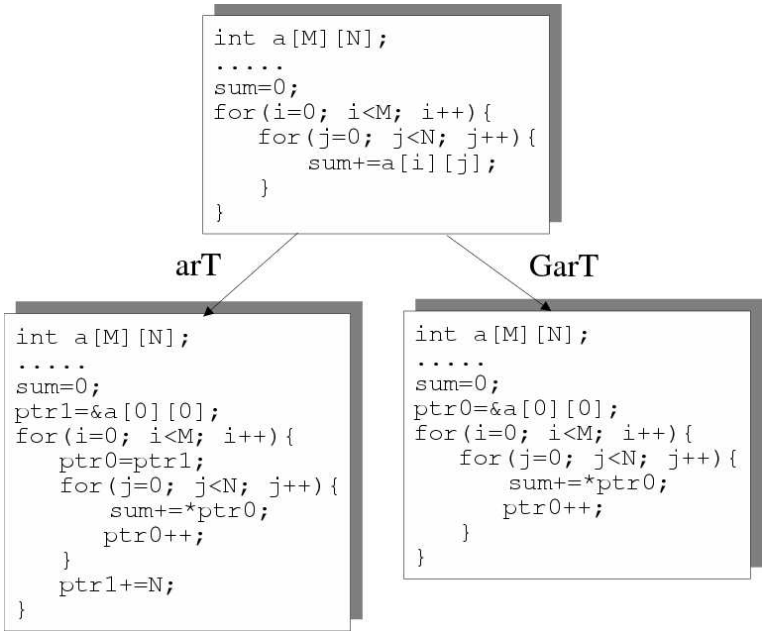


FIG. 3.5 – Optimisations effectuées par les modules ArT et GarT

La portée des transformations de tableaux, restreinte à un niveau de boucle par ArT, est étendue ici aux boucles imbriquées. L'analyse d'expression d'induction et le remplacement de références de tableaux par des pointeurs sont appliqués par GarT globalement aux boucles imbriquées. En particulier, un unique pointeur est utilisé afin d'accéder aux éléments de tableaux à plusieurs dimensions. La valeur de ce pointeur est ajustée à la fin de chaque boucle imbriquée afin de commencer l'itération suivante de la boucle contenante avec la bonne valeur. La figure 3.5 illustre les transformations faites par GarT.

3.3.3.3 Les boucles matérielles

L'utilisation de boucles matérielles est une caractéristique que l'on retrouve sur beaucoup de DSPs. Dans ces boucles, le compteur et le saut sont gérés par le matériel. Ainsi le gain de performance à l'exécution est significatif car il n'y a pas de surcoût. Pour être réécrite en boucle matérielle, une boucle doit avoir un nombre fixe d'itérations et une taille de corps de boucle tous deux connus à la compilation. Dans le compilateur *C* du MMDSP+, les boucles candidates sont détectées à haut-niveau. Elles sont réécrites afin d'utiliser l'instruction spéciale définissant une boucle matérielle. Le listing de la figure 3.6 présente un exemple de boucle matérielle. L'instruction `repi L2,#128` indique que le corps de boucle va jusqu'à l'étiquette `L3` et qu'il doit être exécuté 128 fois. Le compteur de boucle est dans le registre `r5`. Dans le

cas d'un mécanisme classique de boucle, figure 3.6(a), il est initialisé à 0 en début de boucle, puis incrémenté de 1 et comparé à 128 en fin de boucle. Le saut conditionnel se fait alors vers l'étiquette `L3` afin de ré-exécuter le corps de boucle. Dans le cas d'une boucle matérielle, ces quatre instructions (initialisation, incrémentation, comparaison et saut) sont remplacées par une simple opération `repi`.

```
    move #0,r5
L3:
    load  [P0++], r0
    load  [P1++], r1
    mul   r0, r1, r2
    add   r2, r3, r3
    add   r5, #1, r5
    cmp   r5, #128
    blt   L3
L2:
```

(a) Avant réécriture

```
    repi L2, #128
L3:
    load   [P0++], r0
    load   [P1++], r1
    mul    r0, r1, r2
    add    r2, r3, r3
L2:
```

(b) Après réécriture

FIG. 3.6 – Exemple de boucle matérielle.

L'intérêt de cette approche est qu'elle ne nécessite aucun mécanisme de récupération au cas où la boucle ne satisferait pas les critères d'éligibilité : la transformation finale se fait à bas-niveau.

3.3.4 Les optimisations de bas-niveau

Les optimisations de bas-niveau sont implémentées en tant que micro-moteurs d'EliXir. Parmi ces optimisations, on peut trouver :

- l'allocation de registres,
- le support pour les boucles matérielles,
- le pipeline logiciel,
- les ordonnancements globaux et locaux,
- des optimisations à lucarne — *peephole optimization*.

Le pipeline logiciel et l'ordonnancement global sont fait en *post-pass*, après l'allocation de registres. En effet, les AS-DSPs sont souvent contraints par un petit nombre de registres. La pression registre maximum autorisée est rapidement atteinte, même en utilisant des heuristiques de détection de cette pression lors de l'ordonnancement. De plus, les algorithmes de DSP passent énormément de temps dans des boucles. Le code d'éviction — *spill code* — est le code ajouté afin de déplacer la valeur d'une variable en mémoire et ainsi libérer un registre. Ainsi le code d'éviction peut engendrer une perte de performance non négligeable qu'il faut éviter à tout prix, même au prix d'un ordonnancement n'engendrant pas les performances optimales.

3.3.4.1 L'allocation de registres

L'allocation de registres a un impact très important sur les performances de l'application compilée, non seulement par la quantité de code d'éviction introduit mais aussi par la possibilité d'appliquer ensuite certaines optimisations telles que le pipeline logiciel par exemple. Dans le contexte des DSP, le code d'éviction est à éviter à tout prix. La réduction de ces évictions est majoritairement faite à l'aide d'analyses de portées de variables et de hiérarchies d'allocation.

Les coupures de temps de vie des variables sont faites à partir d'une représentation de type SSA (*Static Single Assignment*). Les nœuds intermédiaires appelés *Phi-nodes* introduits par SSA sont manipulés par l'allocateur de registres. Ce dernier essaye de les fusionner, sinon il les insère simplement.

L'allocation de registres est fournie avec une API se greffant par dessus EliXir (figure 3.7). Les heuristiques de coloration de Briggs par fusion itérative [GA96] — *iterative coalescing* — sont utilisées ainsi que des optimisations sur les évictions.

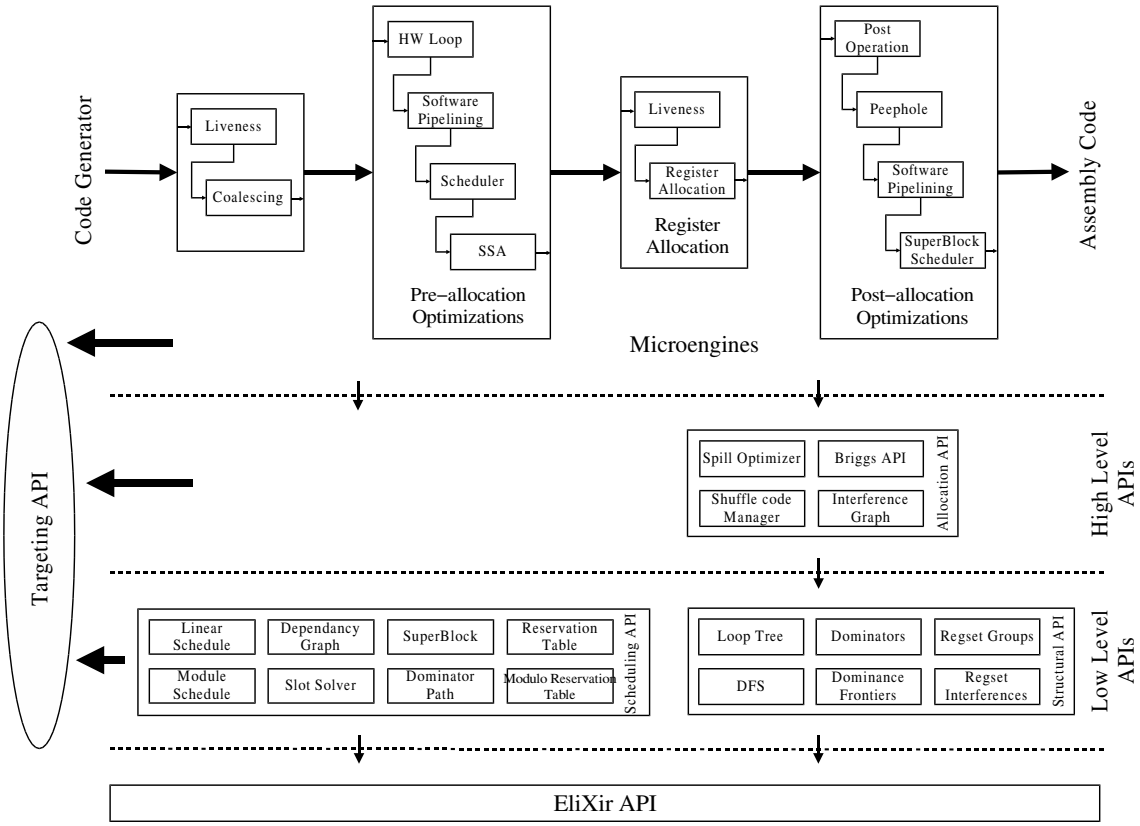


FIG. 3.7 – Micro-moteurs et APIs EliXir

3.3.4.2 Le pipelining logiciel

Le pipeline logiciel est un ordonnancement des instructions d'une boucle dont le but est de construire une boucle équivalente de taille minimum en superposant des opérations provenant d'itérations différentes de la boucle originale. Bien entendu, cette réécriture soulève des problèmes de correspondance entre le code source et le programme optimisé. Nous donnons ici une description détaillée du principe car une partie de nos expérimentations a été réalisée sur cette optimisation.

L'algorithme de pipeline logiciel sur lequel nous nous appuyons est celui implémenté par le compilateur *C* du MMDSP+ [BDG+02]. Il s'agit d'un algorithme du type **modulo software-pipelining** inspiré de B. Rau [Rau94]. L'optimiseur qui l'implémente sélectionne les boucles pouvant être pipelinées en fonction des critères suivant :

Définition 3.1 *Une boucle est éligible si elle respecte les trois propriétés suivantes :*
- *elle ne contient aucune boucle imbriquée*
- *son incrément est de* 1*,*
- *elle ne contient aucune instruction conditionnelle.*

La dernière restriction signifie que le corps de boucle ne doit pas contenir d'instruction `if` autre que le test de sortie. Généralement, il s'agit d'une contrainte forte : les développeurs ont souvent besoin d'utiliser de tels sauts conditionnels dans les corps de boucles ; `switch`, `if` et autres opérateurs conditionnels sont courants. Dans le contexte des systèmes embarqués, les applications sont très spécifiques : le processeur MMDSP+ est dédié à l'encodage et au décodage de formats audio et vidéo comme par exemple l'AMR ou le MP3. Ce genre d'algorithme utilise de manière très intensive les boucles afin de manipuler de gros tableaux, et ce, sans instructions de saut conditionnel tel que le `if`. De plus, notons que dans le cas le plus général il existe une transformation appelée *if-conversion* qui peut être appliquée sur des VLIW à instructions prédiquées et donc diminue grandement l'impact d'une telle restriction.

Intuitivement, plus il y a d'instructions d'un programme exécutées simultanément, plus ce programme s'exécutera rapidement. Nous pourrions donc obtenir l'exécution la plus rapide en exécutant toutes les instructions en parallèle, ce qui est impossible à cause des contraintes suivantes :
- **La dépendance de données :** Si une instruction `A` calcule un résultat qui est utilisé par un opérande d'une instruction `B`, alors `B` ne peut s'exécuter avant que `A` ne soit terminée.
- **L'unité fonctionnelle :** S'il y a x opérateurs (multiplicateurs, additionneurs, etc) sur le circuit, alors au plus x opérations (multiplications, additions, etc) pourront s'exécuter en même temps.
- **Le décodeur d'instructions :** Il ne peut y avoir plus de y instructions fournies à la fois : la taille du mot contenant les instructions du VLIW est limitée.
- **Les registres :** Au plus z registres peuvent être utilisés en même temps.

Les trois dernières contraintes sont souvent regroupées sous le terme de *contraintes de ressources*. C'est pourquoi, afin d'être au plus près du matériel et de ses spécificités, le pipeline logiciel est effectué au niveau assembleur, c'est-à-dire après la génération de code. La première contrainte, celle des données, est la seule à avoir un impact sur nos travaux dans ce domaine. Les trois autres n'ont pas d'impact sur nos travaux car elles sont résolues en respectant les différentes dépendances inter-instructions. Une dépendance entre deux instructions existe si le fait d'interchanger leur ordre change le résultat. Un graphe de dépendances des données — DDG *Data Dependency Graph* — est utilisé afin de décrire les dépendances entre instructions : les nœuds sont les opérations et l'ensemble des arcs est l'ensemble des dépendances. *Anti-dependencies* et *output dependencies* sont également respectées pendant l'ordonnancement du corps de boucle (voir section 2.2.2).

Superposer les différentes itérations d'un corps de boucle implique dans bien des cas la génération d'un prologue et d'un épilogue à ce corps de la boucle. Les instructions sont superposées à travers les itérations de la boucle originale, certaines instances d'instructions peuvent être exécutées en avance dans le prologue. De la même manière, les dernières instances de certaines instructions peuvent être déplacées en dehors du corps de boucle, dans l'épilogue.

80% du code de l'ensemble des tests mis à disposition pas STMicroelectronics dans sa suite de tests est fait de boucles. Le compilateur `mmdspcc` en pipeline entre 30% et 50%, selon le codec. Le tableau 3.2 détaille le pourcentage de boucles pipelinées par codec.

Librement inspiré de A. Appel [App98], une boucle du code source est donnée dans le listing

Codec (CODeurs/DÉCodeurs)	**Nombre de boucles analysées**	**Nombre de boucles pipelinées**	**ratio**
Enhanced Full Rate	179	83	46%
AMR	354	172	49%
g723.1	306	131	43%
prologic24	54	6	11%
sbcwork	119	39	33%

TAB. 3.2 – Ratio de boucles pipelinées par codec de l'ensemble d'applications tests.

de la figure 3.8. Dans sa version originale, cette boucle contenait des accès tableaux qui ne compliquaient que trop la lecture de l'exemple sans incidence sur le résultat. Nous avons enlevé cette syntaxe superflue car elle ne modifiait pas les effets du modulo-pipeline logiciel.

Il s'agit d'une boucle `for` de N itérations, N étant inconnue à la compilation.

```
1    for (i = 1; i ≤ N; i++) {
2          a = j + b
3          b = a + f
4          c = e + j
5          d = f + c
6          e = b + d
7          f = 42
8          g = b
9          h = d
10         j = 43
11   }
```

FIG. 3.8 – Une boucle `for` qui va être pipelinée au niveau logiciel

Cet exemple est relativement complet car il montre trois sortes de dépendances de données inter-itération. Nous les résumons dans le tableau 3.3, où les indices représentent les numéros d'itération. En voici le descriptif :

- **Il n'y a aucune dépendance inter-itération.**
  Deux cas sont présents ici. Certaines variables telles que `f` et `j` n'ont aucune dépendance avec les autres variables. Le second cas de dépendance est donné par `e`, `g` et `h` qui dépendent d'instances de la même itération que la leur, respectivement `b` et `d`, `b`, et `d`.
- **Il y a dépendance avec l'itération précédente.**
  Les variables `a` et `c` dépendent d'instances d'affectation de l'itération précédente, respectivement `j` et `b`, et `e` et `j`.
- **Il y a dépendances avec différentes itération antérieures.**
  Le dernier cas de dépendance est celui illustré par les variables `b` et `d` qui dépendent toutes les deux d'instances d'affectation de différentes itérations précédentes, respectivement `a` et `f`, et `f` et `c`.

Notre exemple part du principe que les instructions contiennent cinq opérations. Afin de ne pas fourvoyer le lecteur, nous omettons les instructions *load* et *store* car elles ne changent pas les principes de l'algorithme. Le résultat du pipeline logiciel est donné par la figure 3.9. Une fois

		Itérations					
	Addr	1	2	3 (N-2)	4 (N-1)	5 (N)	numéro de ligne du code source
	0x01	acfj					(2,4,7,10)
	0x02	bd	fj				(3,5,7,10)
	0x03	egh	a				(6,8,9,2)
Prologue	0x04		bc	fj			(3,4,7,10)
	0x05		dg	a			(5,8,2)
	0x06		eh	b	fj		(6,9,3,7,10)
	0x07			cg	a		(4,8,2)
			for i=3 to N-2 {				(1)
	0x09			d	b		(5,3)
Corps de boucle	0x0A			eh	g	fj	(6,9,8,7,10)
	0x0B				c	a	(4,2)
			}				(11)
	0x0D				d	b	(5,3)
	0x0E				eh	g	(6,9,8)
Épilogue	0x0F					c	(4)
	0x10					d	(5)
	0x11					eh	(6,9)

FIG. 3.9 – Détail d'ordonnancement de la boucle du listing de la figure 3.8.

Variable	Dépend de	
a_i	j_{i-1}	b_{i-1}
b_i	a_i	f_{i-1}
c_i	e_{i-1}	j_{i-1}
d_i	f_{i-1}	c_i
e_i	b_i	d_i
f_i	42	
g_i	b_i	
h_i	d_i	
j_i	43	

TAB. 3.3 – Les dépendances entre variables

de plus, afin de faciliter la lecture, nous considérons que chaque instruction fait la taille d'une unité adressable. Le ciblage vers un processeur spécifique reste trivial.

Au chapitre 4, nous analyserons le pipeline logiciel. Nous énoncerons la propriété 4.5 sur l'ordonnancement des instructions sur laquelle reposent les algorithmes correspondant à nos travaux, nous les détaillerons section 4.3.2.

3.3.4.3 L'ordonnancement

Des techniques d'ordonnancement global ont été conçues pour exploiter au mieux les ASP-VLIWs. Ainsi, du parallélisme non exploité par les ordonnanceurs locaux est révélé. De plus, les processeurs embarqués ne fournissent pas toujours de mécanismes d'exécution spéculative et passent le plus clair de leur temps d'exécution dans des boucles déjà pipelinées. Ces deux points

réduisent les effets de l'ordonnancement global sur les programmes embarqués. Néanmoins l'implémentation d'un ordonnanceur global dans FlexCC2 a été réalisée avec des résultats considérés comme bons [BDG+02].

La densité du code est un critère fondamental des processeurs embarqués où la mémoire est limitée. Les techniques générales telles que le *trace scheduling* n'ont pas été développées dans FlexCC2. La technique implémentée est celle de W. Hwu et al.[HMC+93], aidée par l'ordonnancement de type *dominator path scheduling* [Swe92, SB92].

3.3.4.4 Les optimisations à lucarne — *Peephole optimisations*

Les optimisations à lucarne améliorent de langage assembleur en remplaçant des séquences d'instructions consécutives par d'autres séquences équivalentes mais plus efficaces. Ces optimisations sont basées sur un moteur de reconnaissance de motifs — *pattern-matching*. Ce dernier parcours le programme à la recherche de séquences à remplacer définies par l'utilisateur. Cela permet de nettoyer le code final simplement et à peu de frais.

Dans FlexCC2, le concept de lucarne a été étendu à des motifs basés sur des relations dans le flot de données (par exemple def, use, kill). Néanmoins, le coût de recalcul de *def-use* oblige à limiter l'espace de description de motifs ainsi que la portée des formes dans la phase de reconnaissance.

3.3.5 Reciblage de FlexCC2

Le *reciblage* est le fait d'adapter la sortie d'un compilateur à un processeur particulier. Lors de la conception d'un nouveau processeur, le développement en parallèle de l'architecture et du compilateur *C* est essentiel [BDG+02]. Le fait d'avoir un compilateur tôt dans la chaîne de développement du processeur permet le design de l'ISA — *Instruction Set Architecture* — avec la perspective du compilateur. Ainsi, la suppression d'instructions inexploitables par le compilateur est possible pour une meilleure performance compilateur/processeur.

FlexCC2 permet le reciblage rapide du compilateur en automatisant au maximum la phase d'abaissement et le reciblage à l'aide de fichiers de configurations. Le reciblage de FlexCC2 est divisé en trois parties :

- Abaissement — *lowering*. La représentation intermédiaire est transformée afin de correspondre aux possibilités du processeur. CoSy fournit un ensemble de moteurs pour abaisser le code.
- Les règles de génération de code et la description de la machine. Ces fichiers sont utilisés comme entrées respectives de BEG et EliXir. Le premier doit être écrit par un expert de reciblage car il demande une certaine connaissance de la CCMIR ainsi que des instructions du processeur cible.
- Les fichiers de description des chaînes de moteurs et micro-moteurs.

Contrairement à d'autres compilateurs, le générateur de code de FlexCC2 ne produit pas le code machine cible. Il produit la forme intermédiaire d'EliXir. Pour ce faire, une API est utilisée. Cette dernière est automatiquement générée par EliXir à partir du fichier de description machine. La figure 3.10 décrit le reciblage de FlexCC2, les fichiers de ciblage sont ombrés. Un exemple de description machine est donnée figure 3.11.

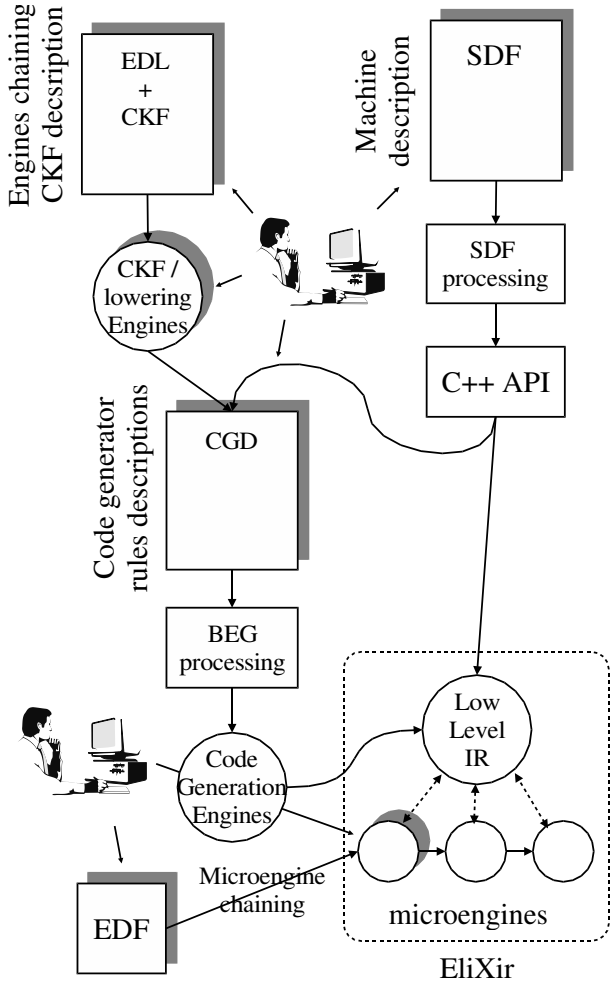


FIG. 3.10 – Le reciblage de FlexCC2

3.3.5.1 Les instructions spécifiques au processeur

Les instructions spécifiques au processeur sont prises en compte par les moteurs qui abaissent la représentation intermédiaire. Une reconnaissance de motifs — *pattern matching* — est faite sur l'arbre de représentation du programme à l'aide du CGD — *code generator description* — préalablement complété. La génération de code génère des instructions virtuelles qui seront transformées en vraies instructions par des micro-moteurs spécifiques (*non-orthogonal post-operations* par exemple) ou génériques (ordonnanceur exploitant la notion de *slot*).

Les opérations ne prennent pas toutes les mêmes types de paramètres. Les opérandes peuvent être un accès mémoire, un registre ou une valeur immédiate.

3.3.5.2 Les instructions virtuelles

Les post-opérations non-orthogonales[2] sur les modes d'adressages sont des mécanismes fréquemment exploitées par les ASPs, en général à cause des contraintes d'encodage. Par exemple :

[2]Une opération est dite orthogonale lorsqu'elle permet les trois types en paramètre (registre, adresse mémoire ou entier) non-orthogonale sinon.

```
// General purpose data registers
regset rl[7] syntax "r##l";
regset rh[7] syntax "r##h";
regclass rlh = rl,rh;               // short
regset r = rl:rh;                   // long
regset ext[7];                      // extensions
regset xr = rl:rh:ext syntax "r##"; // long long

// Multiplier register
regclass ml = rl[0], rh[0];
regclass mr = rh[0], rh[1], rl[1];

// Slots (XBUS and ALU "move")
slot SALU, SXBUS;

operation mv (register rs, register rd)
   syntax = "dmv\t{rs}, {rd}";
   slot SALU = "dmv\t{rs}, {rd}";
   slot SXBUS = "xmv\t{rs}, {rd}";
   reserv_table = [
      read rs(1),
      write rd(1)
   ];
   semantic = {
      type = MOVE;
   }
}

operation lea_ACU(register base, int offset,
                                 register ax) {
   syntax="lea\t#{offset}+{base},{ax}";
   reserv_table = [
      read base (1),
      write ax (1),
      use CST_BUS,
      use ACU,
      use ACU_SAR[ax]
   ];
}
```

FIG. 3.11 – Description de processeur dans EliXir (SDF)

- Le transfert d'une valeur entre deux jeux de registres en tant que post-opération, mais en utilisant le même numéro de registre : $A_i = R_i$.
- La post-décrémentation d'un registre d'adresse avec la valeur d'un index de registre, mais en utilisant le même numéro de registre : A_i -= I_i.

Ces opérations sont essentielles aux performances, mais difficiles à exploiter au moment de la génération de code. L'approche de FlexCC2 est de générer des instructions virtuelles qui seront détectées et remplacées plus tard par les micro-moteurs de EliXir. Ceci est également appliqué aux post-opérations d'adressage orthogonal. La figure 3.12 donne un exemple de ce type d'optimisation. Une post-opération non-orthogonale est introduite sous la forme d'une instruction *xmv_ACU* par un moteur d'optimisation de modes d'adressage.

3.3.5.3 Les idiomes de la machine : *intrinsics*

Le MMDSP+ propose des instructions matérielles qui réalisent efficacement certaines opérations spécifiques. Elles sont utilisées pour permettre l'utilisation de caractéristiques propres au processeur directement au niveau du code source *C*. Dans CoSy, ces fonctions sont décrites dans un fichier et automatiquement détectées par le haut-niveau du compilateur. Le `mmdspcc` propose entre autres les fonctions `min` et `max` sous forme d'intrinsics.

```
(a) Assembly code before addressing mode optimization

...
ldx_f   ax1,r4l
add     r0l,#1,r2l
xmv     axx1,ax1
...

(b) Optimized code using non-orthogonal post-operation

...
ldx_f   ax1,r4l || xmv_ACU  axx1,ax1 || add   r0l,#1,r2l
...
```

FIG. 3.12 – Optimisation d'instruction spécifique

3.4 Le débogueur

L'équipe IDTEC de STMicroelectronics développe et maintient une infrastructure de construction de débogueur appelée `IDbug` [GPR+05]. Cet environnement fournit tous les composants nécessaires à la construction de débogueurs au niveau source — *source level debugger*.

`IDbug` est une technologie modulaire. Elle permet le débogage de programme dans le contexte des systèmes embarqués. Un ensemble de composants réutilisables fournit les différents services nécessaires. Ces composants peuvent être intégrés à d'autres outils ou projets, ou utilisés pour la conception d'un débogueur *C* ou *C++* au niveau source.

`IDbug` est composé d'un ensemble de bibliothèques permettant la lecture de fichiers binaires et d'informations de débogage. Depuis leur création, ces bibliothèques ont été intégrées à divers outils. `IDbug` fournit à présent les options suivantes :

- La lecture de formats de fichiers binaires tels que COFF, ELF32 et ELF64.
- La lecture de formats standards d'informations de débogage tels que COFF, ArchelonCOFF, DWARF 2 et DWARF 3.
- Les mécanismes permettant le débogage de programmes écrits en *C* et en *C++* tels que l'instanciation de points d'arrêt.
- Le débogage à distance avec `R2Sserver` (section 3.4.2).

Les équipes d'STMicroelectronics utilisent `IDbug` de deux manières différentes :

- d'un côté la technologie permet à d'autres outils logiciels d'avoir accès aux informations de débogage, de gérer les informations au niveau source ou de contrôler le programme en cours de débogage — *debuggee*,
- d'un autre côté, `IDbug` permet l'implémentation de débogueurs au niveau source de programmes écrits en *C* ou en *C++* ainsi que des solutions ad-hoc pour système ou débogueurs pour systèmes multiprocesseurs.

3.4.1 Architecture générale

La figure 3.13 donne les principaux composants de l'architecture d'`IDbug`. Ces composants sont identifiés et distincts de manière à fournir une grande compatibilité avec d'autres outils. Notons par exemple que `IDbug` propose les mêmes interfaces CLI et MI que `gdb` (resp. Command Line Interface et Machine Interface). Les bibliothèques permettent la lecture de fichiers binaires contenants de multiples formats tels que COFF, ArchelonCOFF, ELF32 et ELF64. Du point

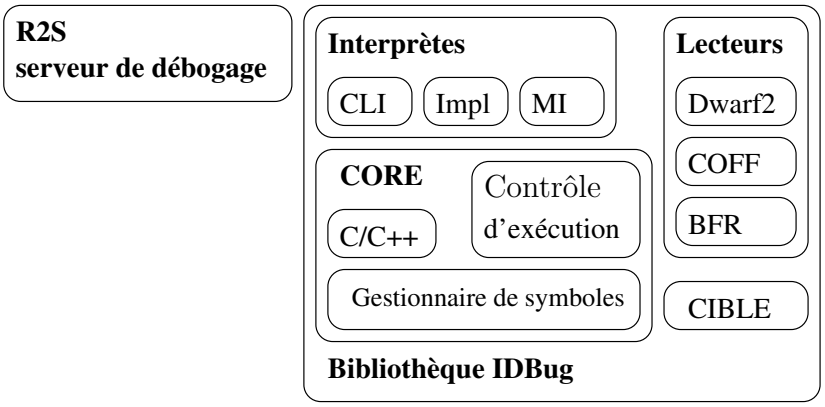


FIG. 3.13 – Les différents composants d'`IDbug`

de vue du développeur d'outils (tels que débogueurs, compilateurs, simulateurs ou profileurs), une grande partie du travail n'est pas à refaire. Les bibliothèques sont disponibles sur plusieurs plates-formes (Sun-Solaris 8 et au-delà, Linux RHEL 3.0, Windows NT/2000/XP : compilation native avec MSVC++ .Net).

Afin de minimiser les dépendances et l'interaction entre les composants, l'architecture est radiale, telle que schématisée par la figure 3.14. Afin de satisfaire le plus grand nombre de caractéristiques des systèmes d'applications distribuées sur systèmes multiprocesseurs, `IDbug` propose les fonctionnalités de débogage à distance, multi-contexte et esclave.

3.4.2 Le débogage à distance — *remote debugging*

Le débogage à distance, ou *remote debugging*, est le débogage d'un système embarqué sur une plate-forme cible tout en contrôlant l'exécution depuis une station de travail hôte. Le débogueur en est capable s'il sépare le moteur de contrôle d'exécution du reste afin de l'embarquer sur la machine cible tout en exécutant les autres moteurs sur un autre système (le plus souvent la machine hôte).

Un des composants d'`IDbug` est le serveur `R2Sserver` qui utilise un protocole de communication asynchrone afin de contrôler à distance l'exécution du programme. Le serveur `R2Sserver` est similaire au `gdbserver`. Il est utilisé pour la connexion du débogueur du MMDSP+ sur un hôte de type UNIX avec un système Windows sur lequel une plate-forme est connectée. Les principales caractéristiques du `R2Sserver` sont :

- une communication basée sur des évènements,
- Une communication asynchrone,
- une communication bidirectionnelle — *full duplex*,
- un serveur *multi-threadé*.

Le serveur est également utilisé entre le débogueur MMDSP+ et une plate-forme TLM ou pour déboguer une application s'exécutant sur un simulateur intégré à une plate-forme de simulation de niveau RTL. Dans ce cas, le débogueur observe une cible embarquée dans un autre processus. L'utilisation de thread permet de cacher `R2Sserver` dans un processus.

L'approche client/serveur est le meilleur moyen de déboguer une application exécutée à distance ou embarquée dans un autre processus. De plus, le débogage à distance est régulièrement nécessaire pour le débogage multi-processeurs, afin de connecter un unique processus de débo-

gage à tous les processeurs pour gérer l'*OCE debugging — On-Chip Emulation Debugging* — avec un accès unifié.

3.4.3 Le débogage multi-contexte — *multi-context debugging*

Le débogage multi-contexte, ou *multi-context debugging*, est une des caractéristiques les plus notables d'`IDbug`. Elle est basée sur l'instanciabilité de cette technologie. Un débogueur basé sur `IDbug` peut utiliser différents contextes de débogage pour chaque processeur d'une plate-forme donnée. L'architecture sous-jacente permet le partage de composants entre les différents contextes tels que la ligne de commande ou le gestionnaire d'entrées/sorties.

Le débogage de systèmes multi-processeurs avec un unique processus de débogage mais fournissant plusieurs contextes offre les avantages suivants :

- le développeur n'utilise qu'un seul débogueur,
- il y a un unique environnement d'affichage et de contrôle des processeurs,
- la synchronisation d'exécution des commandes pour la simultanéité parmi les processeurs,
- un échange inter-contexte des informations,
- une unique connexion entre le débogueur et le système complet.

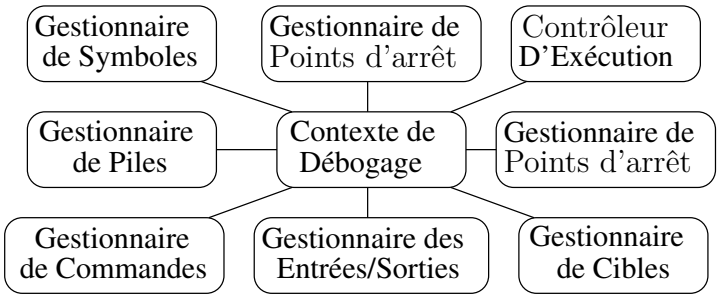


FIG. 3.14 – Architecture multi-contexte d'`IDbug`

3.4.4 Le débogage esclave — *slave debugging*

La partie la plus délicate du débogage multi-cœur d'un système est la définition des règles de synchronisation. Par exemple, sur un système multi-processeurs dont chaque cœur peut s'exécuter indépendamment des autres, il est inutile d'arrêter le système entier quand le cœur débogué s'arrête. D'un autre côté, dans le cas de systèmes distribués partageant la même horloge, il est plus efficace d'arrêter le système complet quand l'exécution atteint un point d'arrêt. `IDbug` propose une approche permettant la résolution de ces problème : le débogage esclave — *slave debugging*.

Le modèle général de communication utilisé est celui de l'esclave/maître par l'utilisation de signaux. Cette approche rend `IDbug` indépendant de toute règle qui serait en rapport avec le contrôle du système global et de la synchronisation. `IDbug` est capable d'accepter des notifications externes suivant son status d'exécution, et ainsi permettre la réaction à des évènements sur lesquels le débogueur n'a aucune visibilité comme par exemple examiner l'état du système sur un bus particulier.

Le modèle de notifications permet au débogage multi-contexte de régler des problèmes de synchronisation. L'écoute concurrente de notifications permet à chaque contexte de connaître le status des autres contextes.

3.5 Conclusion

STMicroelectronics est un fabricant de semi-conducteurs dont l'un des produits est le processeur embarqué MMDSP+ — *multimedia digital signal processor*. Ce processeur est un ASP — *application specific processor* — de type VLIW — *very long instruction word*.

Afin de permettre le développement d'applications pouvant s'exécuter sur ce processeur, `mmdspcc` est un compilateur hautement optimisant. Basé sur FlexCC2, ce compilateur fournit un ensemble d'optimisations allant des plus standards comme la suppression de sous-expressions communes aux plus spécifiques telles que l'utilisation d'intrinsics en passant par différents ordonnancements comme le pipeline logiciel. En tout, plus d'une centaine de moteurs et micro-moteurs composent ce compilateur.

L'équipe au sein de laquelle a été faite cette thèse se nomme IDTEC. Elle propose une infrastructure de construction de débogueurs : `IDbug`. Ce dernier fournit les bibliothèques permettant le débogage à distance, le débogage multi-contexte et le débogage esclave. D'architecture modulaire radiale, il propose les mêmes interfaces de communication que des outils standards tels que `gdb`.

`mmdspcc` et `IDbug` ont servi de support aux expérimentations présentées au chapitre 5.

Deuxième partie

Principes et implémentation

Chapitre 4

Principes généraux, définitions et algorithmes

Heisenbug:
Bogue basé sur le principe d'incertitude d'Heisenberg défini en physique quantique : observer une structure modifie son état. Par exemple le programme tourne sous le débogueur, mais pas sur la ligne de commande.

Sommaire

4.1 Vue d'ensemble

Un compilateur va commencer par analyser le code source et le traduire en une représentation dite intermédiaire [ASU86, Muc97, App98]. Cette analyse aide à la validation de cohérence sémantique et syntaxique du code source. La représentation intermédiaire résultant de cette traduction est dite de haut-niveau. Elle préserve les structures de contrôle complexes que l'on connaît dans les langages de programmation tels que le *C*. Par suite de réécritures, la représentation du programme va s'approcher du langage cible : le langage machine du processeur

cible. Cette traduction est faite en grande partie par le générateur de code. Ce dernier traduit la représentation intermédiaire de haut-niveau en code proche du langage machine où les structures de contrôle se réduisent à des branchements vers des étiquettes, les opérations complexes initiales sont éclatées en plusieurs opérations simples. Certaines optimisations sont appliquées sur la représentation de haut-niveau, d'autres sur la représentation de bas-niveau. Le fait est que ces deux représentations ne présentent pas les mêmes avantages. À haut-niveau, certaines réécritures comme celles de boucles par exemple sont plus aisées ; à bas-niveau, la prise en compte des spécificités du processeur cible est plus facile.

L'objectif de ce chapitre est d'expliciter notre contribution. De manière générale, après optimisation, nous savons que les informations de débogage ne correspondent plus à la réalité. Nous ne pouvons alors pas déboguer le programme, ni de manière transparente, ni de manière non-transparente. Afin de maintenir ces informations à jour lors de la compilation, nous proposons une instrumentation du compilateur et du débogueur. Ce dernier utilise des tables de correspondances entre les adresses du fichier exécutable et les lignes des fichiers sources. Dans le cadre du débogage de code optimisé, ces tables ont à nos yeux deux lacunes :

1. n'étant pas maintenues à jour lors des optimisations, elles sont obsolètes avant même l'édition de liens,
2. elle ne contiennent aucune information sur les optimisations qui leur ont été appliquées.

Le premier point a été relevé par l'ensemble des chercheurs qui se sont attelés à ce problème. Le second point a d'abord été relevé par les auteurs de `CXdb` [SBB+91]. Une solution a ensuite été proposée pour les optimisations de haut-niveau par C. Tice [TG98] à travers l'outil `Optview`.

Lors de la phase de réécriture du code source en une représentation intermédiaire, nous proposons d'annoter chaque instruction avec trois champs :

1. un couple ligne/fichier relatif au code source,
2. un booléen établissant si l'instruction est clé ou non (définition 4.2 donnée plus avant),
3. une trace initialement vide.

Ces annotations se retrouvent parmi celles normalement utilisées par le compilateur. Chaque modification apportée au programme implique leur modification. Par exemple, lors de la réécriture d'une instruction, une notification de cette modification est ajoutée à sa trace. Toutes les optimisations tiennent à jour ces informations tout comme elles le font déjà avec les autres annotations. Ainsi, au fur et à mesure des optimisations, les instructions portent avec elles une forme d'historique des différentes optimisations apportées au programme. Lors de la génération du code exécutable, ces informations sont rassemblées et ajoutées de la même manière que le sont les informations plus classiques telles que DWARF 2 ou COFF. Le débogueur les utilise alors afin de répondre de manière précise aux requêtes de l'utilisateur.

Nous allons d'abord définir la notation des concepts essentiels à notre approche : les représentations intermédiaires, les instructions clés, les ordonnancements, les traces d'optimisations et les tables de correspondance. Nous commençons par en donner les définitions qui nous permettrons, en fin de chapitre, de décrire notre contribution [VRFS07, VRFS08], nos instrumentations d'algorithmes dans le compilateur et nos algorithmes à implémenter dans le débogueur :

- Les instrumentations d'algorithmes que le compilateur implémente permettent de maintenir les informations de débogage en correspondance avec l'état réel du programme après optimisation.
- Les algorithmes que le débogueur implémente permettent de rendre compte à l'utilisateur de l'exécution réelle du programme.

La syntaxe simple que nous proposons nous permet de dégager les similarités qui existent entre les différentes représentations intermédiaires utilisées par les compilateurs. Ainsi, l'écriture d'algorithmes implémentables à haut comme à bas niveau du compilateur est facilitée. Des exemples illustrant les définitions et concepts donnés ici seront déroulés aux sections 4.3.1.1 et 4.3.1.2. Nous proposons une méthode afin de déboguer une boucle qui aurait été pipelinée au niveau logiciel [VRFS07], l'algorithme et un exemple sont donnés en section 4.3.2.

4.2 Définitions et principes de base

Afin de manipuler le programme, un compilateur utilise une ou plusieurs représentations intermédiaires, notées *RI*, de ce programme. Leur nombre dépend des propriétés qu'elles offrent en fonction des attentes et besoins du compilateur. Par exemple, pour allouer les registres aux variables, il est important que la RI permette explicitement la manipulation des ressources du processeur. Lors d'optimisation de boucles, une RI faisant abstraction de tels détails va permettre une manipulation plus facile du graphe de flot de contrôle. Une représentation intermédiaire de programme utilisée à bas niveau peut être, par exemple, celle décrite par A. Aho, R. Sethi et J. D. Ullman [ASU86] appelée le code trois adresses. Chaque instruction est un quadruplet composé de trois adresses (deux opérandes et une adresse résultat) et d'un opérateur. Steven S. Muchnick [Muc97] utilise trois représentations de haut, moyen et bas niveau dont les acronymes respectifs sont HIR, MIR et LIR —***H**igh / **M**edium / **L**ow level **I**ntermediate **R**epresentation*. `LLVM` [LA04], tout comme `gcc` [Sta99], utilise une représentation intermédiaire de type SSA. Le compilateur *C* du MMDSP+ en utilise deux : la première, appelée CCMIR, est dite de haut niveau, puis, après la génération de code, la seconde, appelée EliXir, est dite de bas niveau.

4.2.1 Représentation de programme et instructions

Un programme est un ensemble d'instructions que nous notons $\mathcal{S}$. Cet ensemble ordonné est une représentation du programme. Le compilateur va modifier cette représentation, la réécrire sous une autre forme. Nous allons utiliser une notation qui nous permettra de manipuler simplement ces transformations au niveau d'abstraction qui nous convient. Les unes après les autres, les optimisations modifient une représentation que nous notons $\mathcal{S}^{i-1}$ en une représentation que nous notons $\mathcal{S}^i$. Nous notons f^i la fonction d'optimisation de $\mathcal{S}^{i-1}$ vers $\mathcal{S}^i$. Nous définissons la chaîne d'optimisation comme étant la composition de x optimiseurs : $f \stackrel{def}{=} f^x \circ \ldots \circ f^1$. Le processus complet de compilation est noté $f(\mathcal{S}^0) = \mathcal{S}^x$. Il en ressort deux représentations particulières : le code source original $\mathcal{S}^0$, et le programme binaire optimisé S^x.

$$\mathcal{S}^0 \xrightarrow{f^1} \mathcal{S}^1 \xrightarrow{f^2} \ldots \xrightarrow{f^{x-1}} \mathcal{S}^{x-1} \xrightarrow{f^x} \mathcal{S}^x$$

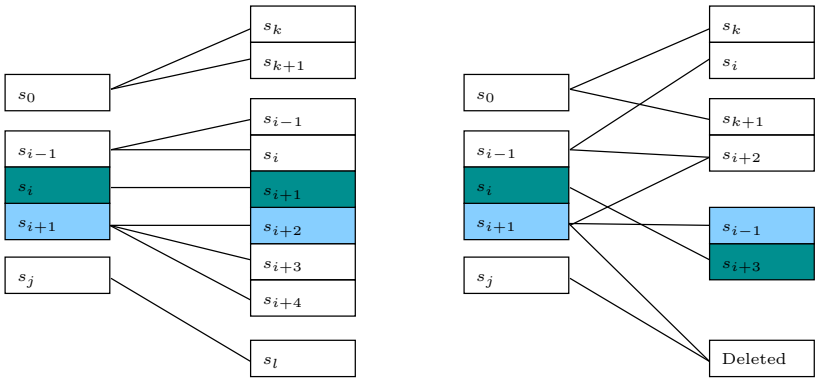
Nous avons vu à la section 2.2.3 la définition d'un graphe de flot de données $G = (\mathcal{S}, A_{CFG}, s_e, s_s)$. La relation A_{CFG} définit un ordre partiel sur $\mathcal{S}$, nous notons $\prec$ cet ordre. Cet ordre sera utilisé sur les différents éléments $\mathcal{S}$ du flot de compilation.

Lors de l'exécution du programme, les instructions du code source ont pour la plupart vocation à être exécutées plusieurs fois. Celles d'un corps de boucle, par exemple, seront exécutées autant de fois que la boucle comporte d'itérations. À chacune de ces itérations, les valeurs des données peuvent changer, le compteur de boucle et son incrémentation en sont un exemple caractéristique. Nous parlons alors d'**instance d'instruction**. Une exécution d'instruction s

est appelée une **instance** de s. De manière similaire, une **instance de point d'arrêt** est l'interruption de l'exécution du programme à ce point d'arrêt.

À chaque optimisation, chaque instruction est transformée en un ensemble d'instructions. Nous notons r la relation qui existe entre les instructions de $\mathcal{S}^0$ et les instructions de $\mathcal{S}^x$. Cette relation est la composition de relations $r^i \in \mathcal{S}^{i-1} \times \mathcal{S}^i$ telles que $r \stackrel{def}{=} r^1; \ldots; r^x$, et $r^i = \{(s_a, s_b) | s_a \in \mathcal{S}^{i-1} \wedge s_b \in \mathcal{S}^i \wedge s_b$ a été générés à partir de $s_a\}$. À chaque étape, i.e. optimisation f^i, de la compilation, les informations de débogage contiennent la relation $r^c \in \mathcal{S}^0 \times \mathcal{S}^i$ qui est calculée à partir de la table de correspondance de l'étape précédente : $r^c = r^1; \ldots; r^i$. Nous verrons plus concrètement à quoi cela correspond grâce aux exemples des sections 4.3.1.1 et 4.3.1.2 de ce même chapitre.

Nous parlons d'entrelacement de deux instructions source s_a et s_b lorsqu'une partie des instructions machine correspondantes de s_b est exécutée avant l'exécution de toutes les instructions machine de s_a. Par exemple, les instructions s_0 et s_{i-1} de la figure 4.1(b) sont entrelacées.



(a) Optimisations désactivées (b) Optimisations activées

FIG. 4.1 – La suppression de sous-expressions communes est désactivée/activée

Définition 4.1 (Entrelacement) *Soit deux représentations d'un programme partiellement ordonnées $(\mathcal{S}^0, \prec)$ et $(\mathcal{S}^x, \prec)$ et la relation r^x définissant les instructions de $\mathcal{S}^x$ ayant été générées à partir des instructions de $\mathcal{S}^0$. Deux instructions $s_a, s_b \in \mathcal{S}^0 \times \mathcal{S}^0$ telles que $(s_a \prec s_b)$ sont dites* entrelacées *s'il existe deux instructions $s_c \in \mathcal{S}^x$ et $s_d \in \mathcal{S}^x$ telles que $(s_a, s_c), (s_b, s_d) \in r^x \times r^x$ et $(s_d \prec s_c)$.*

Écrit autrement, deux instructions s_a et s_b du code source sont dites entrelacées si et seulement si :

$$(s_a \prec s_b) \wedge (\exists s_c, s_d \in \mathcal{S}^x \times \mathcal{S}^x, (s_a, s_c), (s_b, s_d) \in r^x \times r^x) \wedge (s_c \prec s_d)$$

Habituellement, afin de déboguer un programme, les optimisations sont désactivées. Dans ce cas, les instructions du code source ne sont pas entrelacées. Ceci implique que les instructions s^x de la représentation finale $\mathcal{S}^x$ sont contiguës si elles sont générées à partir de la même instruction source s^0. Si deux instructions s_i et s_j sont générées à partir de s_a, alors les instructions s_k de $\mathcal{S}^x$ telles que $k \in [i..j]$ le sont également :

$$(s_a, s_i) \in r^x \wedge (s_a, s_j) \in r^x \wedge (s_i \prec s_j) \Rightarrow \forall s_k, s_i \prec s_k \prec s_j, (s_a, s_k) \in r^x$$

C'est pour cette raison que l'instanciation de points d'arrêt dans du code non optimisé est faite sur la première instruction de cet ensemble $\{s_k | s_k \in \mathcal{S}^x \wedge k \in [i..j]\}$. Au moment de l'arrêt de l'exécution du programme, nous pouvons être sûrs qu'aucune partie de s_a n'aura été exécutée.

Néanmoins, dans le cas d'un programme optimisé, puisque ces instructions sont entrelacées avec d'autres générées à partir d'autres instructions, ce mécanisme ne fonctionne plus. Les instructions source ont rarement une exécution atomique à bas niveau, elles sont en fait composées de plusieurs instructions machine.

4.2.2 Instructions clés

Afin d'instancier un point d'arrêt, l'utilisateur sélectionne une instruction du programme. L'idée est d'exécuter le programme jusqu'à cette instruction. L'exécution y est interrompue. Le débogueur rend la main à l'utilisateur. Ce dernier peut alors inspecter l'état courant du programme. Nous avons vu en section 2.3.2, lors de notre explication des problèmes relatifs à la localité du code, qu'il existait deux manières de considérer ces points d'arrêt. Ils peuvent être syntaxiques ou sémantiques. S'ils sont syntaxiques, leur ordre va suivre l'exécution telle qu'elle a été originalement voulue par le développeur. S'ils sont sémantiques, leur ordre va suivre l'exécution réelle telle que le compilateur a réécrit le programme. Nous avons donné au chapitre 1 les raisons qui nous ont poussés à choisir un modèle qui suit la sémantique du programme.

Une instruction du code source va très souvent être composée de plusieurs instructions machine. Par exemple, une large majorité de processeurs utilise des opérations à deux opérandes, donc une affectation telle que `a = b + c + d ;` sera composée d'au moins deux instructions machine. La première calculera une sous-expression de l'instruction, `tmp = c + d` par exemple, pour enfin calculer la valeur de `a` par l'addition de cette variable temporaire au reste de l'expression initiale : `a = b + tmp`. Il ne s'agit pas à proprement parler de langage machine, mais le remplacement des variables par des registres nous y ramène facilement. Dans le cas présent, le point d'arrêt sémantique serait instancié sur l'instruction qui affecte `a`. Or, le mécanisme classique d'instanciation de points d'arrêt, tel que décrit dans en section 2.6, fait qu'il serait instancié par le débogueur à la première instruction de la paire générée, celle calculant la variable temporaire.

Dans la mesure où nous cherchons à révéler le comportement réel du programme au développeur, notre table de correspondance doit permettre l'instanciation de points d'arrêt sémantiques. Pour ce faire, nous basons nos travaux sur le concept d'*instructions clés* développé par C. Tice [TG00]. Une instruction clé peut être n'importe quelle instruction changeant l'état du programme **au niveau source**, c'est-à-dire, soit la valeur d'une variable source, soit le flot de contrôle du programme.

Le concept d'*instruction clé* est directement lié au concept de point d'arrêt sémantique. Il s'agit de parler de l'instruction machine qui effectue l'opération attendue par l'utilisateur, en ne tenant pas compte des autres. Une fois cette instruction spécifiée, elle sera celle où le point d'arrêt sera instancié. Il s'agit en d'autres mots de décaler l'instanciation du point d'arrêt le plus tard possible sans pour autant exécuter totalement l'instruction source correspondante.

C. Tice [Tic99] a défini les instructions clés comme sa principale contribution au débogage de code optimisé. Ce concept est utilisé ici afin de permettre l'instanciation de points d'arrêt. La définition suivante est adaptée de [TG00, page 9].

Définition 4.2 (instruction clé) *Soit $s_i \in \mathcal{S}^0$ une instruction dans le code source. Soit la relation r^x définissant les instructions de $\mathcal{S}^x$ ayant été générées à partir des instructions de $\mathcal{S}^0$ sans duplication de code. L'instruction clé de s_i est l'instruction de $\mathcal{S}^x$ qui peut déclencher un changement d'état du programme visible au niveau utilisateur.*

Comme indiqué ci-dessus, cette définition considère qu'il n'y a pas de duplication de code. Afin de couvrir le cas où les optimisations effectuent de la duplication de code, les instructions de chaque instance de l'instruction source dupliquée doivent être considérées indépendamment. Les instructions clés séparées doivent être identifiées pour chaque instance. Par exemple, l'instruction clé de l'affectation d'une variable du code source sera l'instruction qui écrit la valeur finale de la partie droite en mémoire (partie gauche) ; ou, si l'écriture en mémoire a été supprimée par une optimisation, ce sera l'instruction qui calcule la valeur finale de la partie droite et la laisse dans un registre.

Afin de pouvoir manipuler la relation entre les instructions sources et leurs instructions clés, nous définissons la relation K^i suivante :

$$K^i = \{(s_a, s_b) | ((s_a, s_b) \in \mathcal{S}^{i-1} \times \mathcal{S}^i) \wedge (s_b \text{ est instruction clé de } s_a)\}$$

Pour illustrer ce concept d'instruction clé, prenons le langage défini figure 4.2. Pour chacune de ces instructions source il existe une instruction clé. L'affectation a pour instruction clé la dernière instruction machine qui met dans la ressource affectée à x sa valeur finale. Le saut conditionnel `if` a pour instruction clé l'instruction machine provoquant le saut. La boucle `while` a également pour instruction clé l'instruction machine provoquant le saut.

$S ::$	$x := e$	affectation
	$\mid$ if e then S else S	saut conditionnel
	$\mid$ while e do S	boucle

FIG. 4.2 – Un langage simple afin d'illustrer le concept d'instruction clé.

Il est important de noter que les points d'arrêt s'instancient par ligne du code source. Or les instructions clés se calculent par point d'arrêt, donc par ligne du code source. C'est pourquoi une boucle écrite sur une ligne, corps de boucle compris, n'aura qu'une instruction clé alors que la même écrite sur plusieurs lignes en aura plusieurs. De manière générale, un point d'arrêt instancié sur une ligne composée de plusieurs instructions sera sur la première instruction clé rencontrée. Il s'agit d'une technique d'implémentation qui nous paraît judicieuse.

Nous pouvons faire l'hypothèse de ne pas avoir de lignes contenant plusieurs instructions. En *C* cela revient à dire qu'il y a un retour à la ligne après chaque point virgule et après chaque accolade. Dans le cas où cela se produit, les outils comme `gdb` instancient le point d'arrêt sur le premier S de $S; S$. En d'autres mots, l'arrêt se fait avant une quelconque partielle exécution de la ligne.

4.2.3 Graphe de dépendance et Ordonnancement

Le processeur sur lequel nous travaillons est de type ASP-VLIW. Comme nous l'avons vu en section 3.2, une des caractéristiques d'un tel processeur est le nombre d'unités qu'il propose (unité arithmétique et logique, unité de virgule flottante, unité de chargement/rangement en mémoire, unité de branchement). Nous notons u_j^k sa j^{eme} unité de type k dont le total est k_j

et u^k le nombre total d'unités de ce même type. Quand un processeur propose l types d'unités, son nombre total d'unités est alors :

$$u = \sum_{i=1}^{l} u^i = \sum_{i=1}^{l} \sum_{j=1}^{k_j} u_j^k$$

Chaque instruction $s_i \in \mathcal{S}^x$ a un temps d'exécution que nous notons $exec(s_i)$. Il s'agit d'un entier correspondant au nombre de cycles d'horloge nécessaires pour terminer l'exécution de s.

Un problème d'ordonnancement d'un programme — *instruction scheduling* — consiste à organiser dans le temps l'exécution des instructions de ce programme. Il est particulièrement présent dans les cas où les processeurs proposent des mécanismes d'ILP —*Intruction Level Parallelism*. On appelle fonction d'ordonnancement une fonction qui va associer une instruction à un cycle d'horloge du processeur. Ainsi, un ordre sera établi entre les différentes instructions du programme. S'il n'y a pas de parallélisme au niveau des instructions, au plus une instruction peut être exécutée par cycle d'horloge et donc l'ordre est strict. L'ordonnancement des instructions d'un programme peut être calculé à partir d'un graphe de dépendance annoté. Ce graphe est composé d'un ensemble d'instructions, d'un ensemble de dépendances entre ces instructions, et de deux fonctions, l'une associant une instruction avec un type d'unité, et l'autre associant une instruction avec un temps d'exécution. L'unité de ce temps peut être arbitraire même si le plus souvent on parle en terme de cycles d'horloge.

Définition 4.3 (Graphe de dépendance annoté) *Un graphe G de dépendance annoté est un quadruplet $(\mathcal{S}, D, type, exec)$ où*

- *$\mathcal{S}$ est l'ensemble des instructions,*
- *D est l'ensemble des dépendances,*
- *$type$ est une fonction qui renvoie pour chaque instruction son type,*
- *$exec$ est une fonction qui renvoie pour chaque instruction son temps d'exécution.*

Le listing de la figure 4.3 reprend l'exemple du chapitre précédent page 53. Son graphe de dépendance est donné figure 4.4. Afin de simplifier l'écriture de ce graphe, nous nommons chaque

```
1   for (i = 1; i ≤ N; i++) {
2     a = j + b
3     b = a + f
4     c = e + j
5     d = f + c
6     e = b + d
7     f = 42
8     g = b
9     h = d
10    j = 43
11  }
```

FIG. 4.3 – Une boucle `for` composée de 9 instructions.

instruction par la variable qu'elle modifie. Ainsi nous parlerons de l'instruction `a` pour parler de l'instruction de la ligne 2. `a` dépend de `b` et de `j` de l'itération précédente. `e` dépend de `b` et de `d` de la même itération. `f` et `j` constituent des cas particuliers car ce sont des constantes, elles

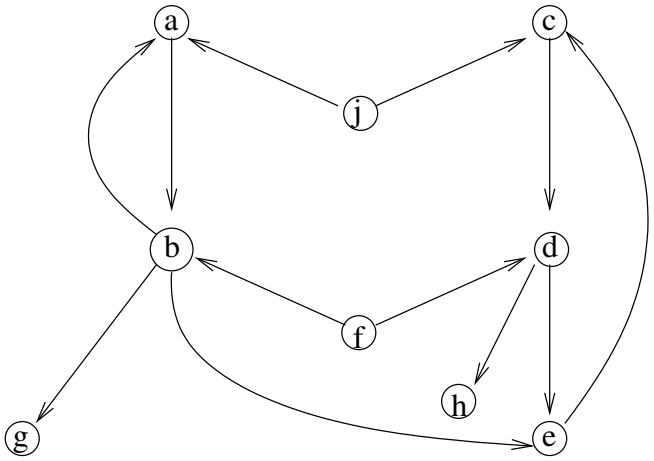


FIG. 4.4 – Graphe de dépendance du listing de la figure 4.3 : les flèches pleines représentent les dépendances inter-instructions d'une même itération de boucle.

ne dépendent donc d'aucune instruction du corps de boucle. Nous supposons que chacune des instructions de la boucle de notre exemple nécessite un unique cycle d'horloge pour s'exécuter : $\forall s_i \in \mathcal{S}^x, exec(s) = 1$.

Un problème connu est celui de l'ordonnancement optimal des instructions d'un programme. Il consiste à trouver un ordre d'exécution des instructions tel que la longueur totale de cet ordonnancement est minimale, donc qu'il prend le nombre minimum de cycles d'horloge. Cet ordre doit respecter les dépendances entre les instructions. À partir du graphe de dépendance annoté, un algorithme d'ordonnancement déduit la fonction qui associe chaque instruction à un cycle d'horloge.

Définition 4.4 (Ordonnancement) *Soit $G = (\mathcal{S}, A_{CFG}, s_e, s_s)$ un graphe de flot de contrôle. Soit D le graphe de dépendances correspondant. Un ordonnancement est une fonction C de $\mathcal{S}$ dans $\mathbb{N}$ qui vérifie les propriétés suivantes :*

1. $\forall s \in \mathcal{S}, C(s) \geq 0$
2. $(s_i, s_j) \in D \Rightarrow C(s_i) + exec(s_j) \leq C(s_j)$
3. $\forall t \in type, \forall i \in \mathbb{N}, |\{s \in \mathcal{S} | (C(s) = i) \wedge (type(s) = t)\}| \leq u^t$

La condition (1) garantit que toutes les instructions sont exécutées. La condition (2) garantit que les dépendances sont respectées. La condition (3) dit que pour tout type t, pas plus de u^t unités de type t sont en correspondance avec un entier donné. Elle garantit donc que toute opération prévue à un certain cycle aura une unité lui permettant de s'exécuter (elle garantit également que seules les ressources accessibles seront utilisées pour chaque cycle). u_t est une borne supérieure, 0 une borne inférieure car il est possible qu'aucune instruction ne soit ordonnancée à un certain cycle. La longueur d'un ordonnancement issu d'une telle fonction C est notée :

$$L(C) = \max_{s \in \mathcal{S}}(C(s) + exec(s))$$

Intéressons nous au cas d'une boucle contenant n instructions $\mathcal{S} = \{s_1, \ldots, s_n\}$. Ces instructions vont être ordonnancées. Dans le cas d'un ordonnancement strictement séquentiel, la

longueur totale est donc la somme des temps d'exécution de ces instructions :

$$L(C) = \sum_{i=1}^{n} exec(s_i)$$

Dans le cas du **pipeline logiciel**, des itérations successives sont superposées. L'intention est de minimiser les dépendances inter- et intra-itérations en entrelaçant les instructions et ainsi d'utiliser au mieux les mécanismes de parallélismes VLIW du processeur. L'algorithme de pipeline logiciel que nous traitons est divisé en quatre étapes que nous illustrons à partir du listing de la figure 4.3 :

1. dérouler la boucle et ordonnancer les instructions le plus tôt possible
2. assembler les groupes d'instructions ainsi mis à jour
3. calculer la largeur de la boucle finale
4. en déduire la construction du prologue, de l'épilogue et du corps de boucle.

Cet algorithme prend une boucle $\{s_1, \ldots, s_n\}$ de n instructions en entrée et renvoie une boucle $\{\mathcal{S}_{Prologue}, \mathcal{S}_{Kernel}, \mathcal{S}_{Epilogue}\}$ de m instructions. La boucle va alors être reconstruite de la façon suivante :

$$\{s_1, \ldots, s_n\} \longrightarrow \{\mathcal{S}_{Prologue}, \mathcal{S}_{Kernel}, \mathcal{S}_{Epilogue}\}$$

avec

· un prologue $\mathcal{S}_{Prologue}$ de longueur $L_{Prologue}(C)$,
· un corps de boucle $\mathcal{S}_{Kernel}$ de longueur $L_{Kernel}(C)$ et
· un épilogue $\mathcal{S}_{Epilogue}$ de longueur $L_{Epilogue}(C)$.

tels que

$$L_{Prologue}(C) + L_{Kernel}(C) + L_{Epilogue}(C) = L(C)$$
$$\mathcal{S}_{Prologue} \cup \mathcal{S}_{Kernel} \cup \mathcal{S}_{Epilogue} = \mathcal{S}$$

Les instructions sont, de plus, regroupées par *mot*. Un mot est un ensemble d'instructions dont l'exécution démarre au même cycle d'horloge. Le premier mot de l'exemple de la figure 4.5(b) est donc $acfj$, le second mot est $bdfj$, et ainsi de suite. L'exemple au complet se note donc :

$$\begin{cases} \mathcal{S}_{Prologue} & = & \{acfj, bdfj, egha, bcfj, dga, ehbfj, cga\} \\ & avec & acfj \prec bdfj \prec egha \prec bcfj \prec dga \prec ehbfj \prec cga \\ \mathcal{S}_{Kernel} & = & \{db, ehgfj, ca\} \\ & avec & db \prec ehgfj \prec ca \\ \mathcal{S}_{Epilogue} & = & \{db, ehg, c, d, eh\} \\ & avec & db \prec ehg \prec c \prec d \prec eh \end{cases}$$

Pipeliner une boucle revient à chercher à remonter l'exécution des opérations au plus tôt dans l'ordonnancement, soit minimiser la valeur de $C(s)$ pour toute instruction $s \in \mathcal{S}$ en tenant compte des contraintes matérielles. Pour ce faire, la boucle va être d'abord déroulée [RG81, Rau94]. Une opération s_j^{i-1} est alors répliquée. Elle est remplacée par plusieurs opérations $s_a^i, \ldots, s_{a+b}^i$. Nous avons alors des ensembles des variables lues et écrites identiques :

$$ref(s_j^{i-1}) = ref(s_a^i) = \ldots = ref(s_{a+b}^i)$$

$\wedge$

$$def(s_j^{i-1}) = def(s_a^i) = \ldots = def(s_{a+b}^i)$$

Le pipeline logiciel tel qu'implémenté dans le bas niveau du compilateur `mmdspcc` utilise la réplication de certaines opérations tout en respectant les dépendances inter-instructions notamment celles entre les différentes instances d'une même instruction. Ainsi, deux instances d'instructions ne seront jamais interverties, la propriété suivante est vérifiée.

Propriété 4.5 *L'ordre des instances d'une instruction est identique entre la boucle non-pipelinée et la boucle pipelinée.*

1. Dérouler et ordonnancer la boucle L'algorithme va dérouler un nombre arbitraire de fois la boucle et chercher un ordonnancement de l'ensemble d'instructions ainsi généré. La boucle est déroulée de manière à en extraire ce que l'on pourrait appeler un *pré-ordonnancement* du corps de boucle. Ce *pré-ordonnancement* est illustré par le tableau 4.5(a). À partir du graphe de la figure 4.4, cet ordonnancement ne tenant pas compte des contraintes matérielles peut être déduit. Le graphe de dépendance nous indique que les instructions calculant les valeurs de `a`, `c`, `f` et `j` de la première itération peuvent être exécutées lors du premier cycle d'exécution du corps de boucle. `b` et `d` dépendent respectivement de `a` et de `c`, leurs calculs sont donc ordonnancés au temps suivant. Sont ensuite ordonnancés `e`, `g` et `h`. `f` et `j` ne dépendent d'aucune variable modifiée dans le corps de boucle, leurs calculs par itérations peuvent donc être prévus en début d'exécution. Au troisième temps, l'instruction calculant la valeur de `a` est également prévue car elle ne dépend pas des instructions prévues au même temps. Le tableau est ainsi complété en déroulant la boucle au fur et à mesure de son ordonnancement.

2. Assembler les groupes d'instructions Notre exemple de déroulage de la figure 4.3 fait ressortir trois parties que nous collons ensemble : `fj ;`, `a ;b ;g ;` et `c ;d ;eh ;`. Nous allons maintenant coller côte à côte un certain nombre de fois ce nouveau corps de boucle.

3. Calculer la largeur de la boucle finale Pour déterminer le nombre de fois que la boucle est répliquée, l'algorithme évalue la largeur minimum de la boucle finale. Un mot de l'ASP-VLIW est composé de u unités. Le corps de boucle $\mathcal{S}_{Kernel}$ contient donc au mieux $u \times L_{Kernel}(C)$ instructions, ce qui signifie que toutes les unités sont utilisées à tous les cycles de la boucle. C'est aussi une borne supérieure de la somme totale de tous les temps d'exécution du corps de boucle. Nous déduisons alors la taille minimum du corps de boucle :

$$L_{Kernel}(C) \geq \frac{\sum_{s \in \mathcal{S}_{Kernel}} exec(s)}{u}$$

Afin de créer ce corps de boucle, l'optimiseur résout incrémentalement certaines contraintes jusqu'à trouver un ordonnancement correspondant à un motif se répétant. En déroulant et en ordonnançant la boucle itérativement, un motif correspondant à $\mathcal{S}_{Kernel}$ se retrouvera. Un exemple de motif est donné dans notre exemple : `db ; ehgfj ; ca`. Ordonnancer la boucle déroulée revient à résoudre le système suivant :

$$\forall (s_i, s_j) \in D \Rightarrow C(s_j) \geq C(s_i) + exec(s_i) - \lambda . L_{Kernel}(C).(offset(s_j) - offset(s_i))$$

λ est un entier que nous cherchons à déterminer. Si aucun motif n'est trouvé, cette variable, initialisée à 1, est incrémentée et la recherche recommencée. Le résultat est un ensemble d'entiers,

un par instruction, dont le plus grand correspond à la largeur du corps de la boucle finale. Dans notre exemple, le résultat est *trois* et donc il nous faut répliquer trois fois notre nouveau corps de boucle afin d'en extraire un motif. La figure 4.5(b) présente le résultat sur notre exemple. Le prologue est constitué des instructions dont le cycle d'exécution est inférieur à 8, et l'épilogue est constitué des instructions dont le cycle d'exécution est supérieur ou égal à 11.

	Itérations					
Temps	1	2	3	4	5	...
1	acfj	fj	fj	fj	fj	
2	bd					
3	egh	a				
4		bc				
5		dg	a			
6		eh	b			
7			cg	a		
8			d	b		
9			eh	g	a	
10				c	b	
11				d	g	
12				eh		
13					c	
14					d	
15					eh	

(a) Détail d'ordonnancement de la boucle du listing de la figure 4.3 dans le cas où l'on ne prendrait pas compte des contraintes matérielles.

	Itérations					
Temps	1	2	3	4	5	...
1	acfj					
2	bd	fj				
3	egh	a				
4		bc	fj			
5		dg	a			
6		eh	b	fj		
7			cg	a		
8			d	b		
9			eh	g	fj	
10				c	a	
11				d	b	
12				eh	g	
13					c	
14					d	
15					eh	

(b) Découverte du motif lors du déroulage du corps de boucle.

FIG. 4.5 – Ordonnancement de la boucle.

4. Construire les trois parties de la boucle finale De manière plus générale, supposons que le motif se retrouve des cycles x à y inclus, il s'agira alors du corps de boucle $\mathcal{S}_{Kernel} = \{s \in \mathcal{S} | x \leq C(s) \leq y\}$. Alors le **prologue** correspondra à l'ensemble des instructions s telles que $C(s) < x$. De manière analogue, l'**épilogue** sera l'ensemble des instructions telles que $C(s) > y$:

$$\mathcal{S}_{Prologue} = \{s \in \mathcal{S} | C(s) < x\}$$

$$\mathcal{S}_{Kernel} = \{s \in \mathcal{S} | x \leq C(s) \leq y\}$$

$$\mathcal{S}_{Epilogue} = \{s \in \mathcal{S} | y < C(s)\}$$

Le fait de dérouler une boucle met à jour une partie des différentes instances d'une même instruction source. Nous notons s_i^j la j^{eme} instance de la i^{eme} instruction. De la même manière, nous notons en exposant le numéro d'itération du corps de boucle[1] : $\mathcal{S}_{Kernel}^j$. Le but est de rendre explicite les dépendances inter-itérations de la boucle source. Étant donné que des itérations sont superposées, nous allons avoir, pour un certain nombre d'instructions, l'appartenance suivante :

[1]Il s'agit d'une notation particulière au pipeline logiciel. En dehors de ce contexte, l'exposant indique toujours l'ensemble des instructions après une optimisation.

$s_i^k \in \mathcal{S}_{Kernel}^l$ où $k \neq l$. Il existe alors une différence entre le numéro d'itération et le numéro d'instance que nous appelons *offset* (definition 4.6).

Définition 4.6 (Offset) *La différence entre le numéro d'une instance d'instruction et le numéro d'itération dans laquelle elle s'est exécutée est appelée* ***offset*** *de l'opération correspondante. Quand la k^{ieme} exécution de l'instruction s_j est à la l^{ieme} exécution du corps de boucle $\mathcal{S}$ alors $offset(s_j) = k - l$.*

Remarquons que lorsque la boucle n'a pas encore été déroulée, l'ensemble des dépendances D de la définition 4.3 de graphe de dépendance annoté est donc composé de couples (s_i^k, s_j^l) où $k = l$, soit un *offset* nul.

Le réordonnancement d'une boucle afin de la pipeliner au niveau logiciel est une forme d'optimisation car la fonction d'ordonnancement C va être modifiée.

4.2.4 Trace d'optimisation

Informellement, nous appelons trace d'optimisation associée à une instruction une liste d'étiquettes. Chacune d'elles représente une modification faite à une instruction. Par exemple, une instruction s^i peut avoir la trace suivante : *strength reduction ; common subexpression elimination ; code hoisting*. Cela signifie que s^i vient d'une s^0 qui a été modifiée par réduction de force, puis par CSE, puis finalement par déplacement vers le haut du programme.

Nous notons L l'ensemble de ces étiquettes présentant une modification apportée à une instruction $s \in \mathcal{S}$. Les étiquettes sont attachées à chaque instruction durant la compilation en fonction de règles détaillées dans les sections 4.3.1 et 4.3.2.

Définition 4.7 (trace d'optimisation associée à une instruction) *Une* trace d'optimisation *T est une séquence d'étiquettes associée à une instruction présentant l'ordre des optimiseurs lors de la compilation : $T \in L^*$.*

4.2.5 Relation de correspondance

Chaque s_j^i est associée à un ensemble d'instructions source. Pour chacune d'elles, elle peut être une instruction clé ou pas, et peut avoir été modifiée par un ou plusieurs optimiseurs. Nous utilisons donc une relation m que nous étiquetons.

Définition 4.8 (correspondance d'instruction) *Une* correspondance d'instruction *m^i est une relation étiquetée entre les instructions de $\mathcal{S}^i$ et $\mathcal{S}^0$ dont les étiquettes sont des traces d'optimisation $T \in L^*$.*

$$m^i = \{(s_a, s_b, T) | (s_a, s_b) \in r^i \wedge T \in L*\}$$

Cette information fait référence à une instruction de la représentation $\mathcal{S}^i$, alors que la représentation de programme est une séquence d'instructions. Par conséquence, nous définissons un couple de relations dont, après compilation, une représentation sera l'information de débogage pouvant être utilisée par un débogueur.

Définition 4.9 (information de débogage) *Soit $m^i \in \mathcal{S}^0 \times \mathcal{S}^i$ une relation de correspondance d'instruction et $K^i \in \mathcal{S}^0 \times \mathcal{S}^i$ une relation d'instructions clés. On note $M^i = (m^i, K^i)$ l'information de débogage.*

M^x, ou sa représentation en tant que structure de données, est l'information de débogage que le compilateur fournit pour la phase de débogage : la **table de correspondance**.

4.3 Instrumentations d'algorithmes et algorithmes

Dans cette section nous utilisons les éléments définis précédemment afin de décrire les instrumentations d'algorithmes à implémenter dans le compilateur et dans le débogueur. Nous traitons d'abord le cas général ne dépendant pas d'une optimisation particulière mais plutôt du type de modification que l'optimiseur effectue sur la représentation du programme. Nous verrons ensuite le cas particulier du pipeline logiciel.

4.3.1 Plus généralement : en fonction du type de modification

Toutes les définitions données précédemment vont servir ici à écrire trois catégories d'instrumentation d'algorithmes simples en apparence mais efficaces, comme nous le verrons au chapitre 5, pour maintenir à jour les informations indispensables au débogage de code optimisé. Chaque optimiseur ajouté à la compilation doit maintenir la table de correspondance à jour en fonction des modifications apportées au programme. Informellement, quand une instruction est réécrite en un ensemble d'instructions, chacune de ces instructions contient la même information et une étiquette est ajoutée à cette information de manière à tracer cette duplication d'information. Quand un ensemble d'instructions est réécrit en un nouvel ensemble d'instructions, nous manipulons des unions d'ensembles d'informations.

Les trois catégories d'instrumentation d'algorithmes permettant une propagation des informations sont les suivantes :

- **La suppression** d'une instruction s_j^{i-1} entraîne la suppression de sa relation de correspondance.
- **Le déplacement** d'une instruction s_a^{i-1} ne modifie en rien les instructions de la table de correspondance car cela n'affecte pas la référence possible à une instruction du code source. En revanche, ses traces d'optimisations T sont augmentées d'une étiquette arbitraire.
- **La réécriture** d'un ensemble d'instructions $\{s_a^{i-1} | s_a^{i-1} \in \mathcal{S}^{i-1}\}$ en un autre ensemble d'instructions $\{s_b^i | s_a^{i-1} \in \mathcal{S}^i\}$ se fait d'une manière sensiblement similaire. Chaque instruction de l'ensemble d'arrivée est mise en relation avec chacune des instructions de l'ensemble de départ. Les traces étiquetant ces relations sont augmentées.

La troisième catégorie inclut la réécriture d'une instruction en un ensemble d'instructions. Cette modification est présente lors de la génération de code car le générateur de code traduit le programme en une représentation au niveau machine. Cette traduction en langage pseudo-machine génère des relations du type un-plusieurs entre les instructions de haut-niveau et la représentation intermédiaire de bas niveau.

Ces trois catégories d'instrumentation d'algorithmes ont été implémentées dans le compilateur C du MMDSP+ (voir le chapitre 3). Nous verrons plus tard que les étiquettes doivent être choisies avec circonspection, un message inapproprié fourvoierait l'utilisateur. Ainsi les étiquettes vont être pour nombre d'entre elles des phrases complètes.

4.3.1.1 La suppression d'expressions communes

Le code source de notre exemple est donné figure 4.6(a). À la compilation, ce programme est optimisé par la suppression de sous-expressions communes. Notre compilateur collecte les informations relatives à ces changements grâce aux algorithmes donnés aux sections 4.3.1 et 4.3.2 et les ajoute au fichier exécutable. Le débogueur est à présent capable de les utiliser afin d'instancier correctement les points d'arrêt et de rendre fidèlement les informations de localité à

l'utilisateur.

```
1  a = b + c * d;
2  e = a + b;
3  g = f + c * d;
```
(a) Code source

```
1  tmp = c * d;
2  a = b + tmp;
3  e = a + b;
4  g = f + tmp;
```
(b) Après CSE

FIG. 4.6 – Listings de code source avant et après modification par la suppression de sous-expressions communes

La suppression de sous-expressions communes — *Common Subexpression Elimination*—, CSE, est le fait de rechercher les instances d'expressions identiques (c'est-à-dire celles à même valeur finale) et d'analyser s'il est intéressant de la remplacer par une variable temporaire contenant cette valeur pré-calculée, le listing de la figure 4.7(b) est le résultat d'une telle modification appliquée au listing de la figure 4.7(a). Les définitions de la section 4.2 nous donnent S^0 et S^1 comme étant respectivement les séquences (s_0^0, s_1^0, s_2^0) et $(s_0^1, s_1^1, s_2^1, s_3^1)$.

s_0^0 : `a = b + c * d;`
s_1^0 : `e = a + b;`
s_2^0 : `g = f + c * d;`
(a) Code Source Initial Annoté

s_0^1 : `tmp = c * d;`
s_1^1 : `a = b + tmp;`
s_2^1 : `e = a + b;`
s_3^1 : `g = f + tmp;`
(b) Code Modifié Annoté

FIG. 4.7 – Codes source et optimisé par CSE

s_0^1 n'est une instruction clé ni de s_0^0 ni de s_2^0 alors que s_1^1 et s_3^1 le sont. En effet, pour chaque affectation de `a` et `g` (resp. s_0^0 et s_2^0), les instructions rendant compte au plus près de la sémantique générale des instructions source sont respectivement s_1^1 et s_3^1. Dans les figure 4.7(a) et 4.7(b), la modification faite au programme implique :

$$r^1 = \{(s_0^0, s_0^1), (s_2^0, s_0^1), (s_0^0, s_1^1), (s_1^0, s_2^1), (s_2^0, s_3^1)\}$$

Après l'exécution de CSE, M^1 est donnée tableau 4.1, où $CSE1$ et $CSE2$ sont des étiquettes choisies arbitrairement. Elles correspondent aux deux modifications que l'optimiseur apporte aux instructions s_0^0 et s_2^0. $CSE1$ étiquette la sous-expression commune alors que $CSE2$ étiquette le reste de l'instruction.

4.3.1.2 L'inversion de boucle

Nous appliquons ici notre approche à une autre optimisation : l'inversion de boucle. Il s'agit de changer le sens de parcours en faisant décroître (resp. croître) l'itérateur s'il croissait (resp. décroissait) initialement. L'intérêt d'une telle optimisation est variable. Dans notre cas, le MMDSP+ propose un mécanisme de boucles matérielles si elles sont décroissantes avec un pas de 1. Afin d'accélérer le parcours de boucle, le compilateur va donc chercher à avoir le maximum de boucle à décrément. Le listing des figures 4.8(a) et 4.8(b) donne un exemple d'un tel changement. Le parcours de la boucle `for` ne se fait plus de 0 à 10 exclus (valeur de `MAX`)

$$\begin{array}{lcl} M^1 & = & (m^1, K^1) \\ m^1 & = & ((s_0^0, s_0^1, CSE1), \\ & & (s_2^0, s_0^1, CSE1), \\ & & (s_0^0, s_1^1, CSE2), \\ & & (s_1^0, s_2^1,), \\ & & (s_2^0, s_3^1, CSE2)) \\ K^1 & = & ((s_0^0, s_1^1), (s_1^0, s_2^1), (s_2^0, s_3^1)) \end{array}$$

TAB. 4.1 – M^1 (après la suppression d'expressions communes)

```
s_0^0 : for (i=0; i<MAX; ++i){
s_1^0 :     sum += x[i];
s_2^0 : }
```

(a) Code source

```
s_0^1 : for (i=MAX-1; i<=0; --i){
s_1^1 :     sum += x[i];
s_2^1 : }
```

(b) Après inversion de boucle

FIG. 4.8 – Listings de code source avant et après modification par inversion de boucle

mais de 9 à 0 inclus, ce qui inverse donc également l'ordre des instances des instructions du corps de boucle. La liste des valeurs de la variable compteur de boucle `i` n'est plus croissante, mais décroissante.

Les boucles annotées sont données par les figures 4.8(a) et 4.8(b). Dans ces figures, la modification faite au programme implique :

$$r^1 = \{(s_0^0, s_0^1), (s_1^0, s_1^1), (s_2^0, s_2^1)\}$$

Il n'y a pas de changement visible à ce niveau de correspondance entre le code source et le code optimisé. Après l'exécution de l'optimisation par inversion de boucle, M^1 est donnée tableau 4.2, où $LR1$ et $LR2$ sont des étiquettes choisies arbitrairement, tout comme l'étaient $CSE1$ et $CSE2$ dans notre exemple précédent. Elles correspondent ici aux indications de réécriture de borne ($LR1$) et d'appartenance au corps de boucle d'une boucle inversée ($LR2$).

$$\begin{array}{lcl} M^1 & = & (m^1, K^1) \\ m^1 & = & ((s_0^0, s_0^1, LR1), \\ & & (s_1^0, s_1^1, LR2), \\ & & (s_2^0, s_2^1,)) \\ K^1 & = & ((s_0^0, s_0^1), (s_1^0, s_1^1), (s_2^0, s_2^1)) \end{array}$$

TAB. 4.2 – M^1 (après inversion de boucle)

4.3.2 Le pipeline logiciel

Cette optimisation est traitée à part car, comme nous allons le voir, l'exécution du code qui en découle a des propriétés que l'on ne retrouve nulle part ailleurs.

Il existe plusieurs raisons pour lesquelles un programme peut s'arrêter durant une session de débogage : point d'arrêt utilisateur, erreur programme. Même si les mécanismes impliqués sont différents, ils finissent tous par arrêter le programme sur une adresse. Dans la suite de ce

chapitre, nous appellerons cette adresse particulière l'*adresse d'arrêt*. À ce moment, l'utilisateur peut interroger le débogueur sur les valeurs des variables : ceci constitue le point de départ de notre algorithme.

Sans considérer un quelconque format d'informations de débogage, nous présentons ici l'information dont notre algorithme a besoin pour résoudre notre problème initial. D'un point de vue implémentation, nous incluons cette information à notre système d'étiquetage tel que défini dans les sections précédentes.

Quand un compilateur optimise une partie de code, il connaît bien plus de choses que ce qu'il fournit en tant qu'information de débogage. Un exemple trivial est le cas de suppression de code mort. Quand un compilateur décide de supprimer une instruction d'un programme, il ne l'enregistre pas dans ses informations, mais alors le débogueur ne peut pas informer l'utilisateur quand ce dernier tente d'instancier un point d'arrêt sur cette instruction. Quand le compilateur pipeline une boucle au niveau logiciel, il calcule le graphe de dépendances, et ainsi sait où chaque variable est (ou sera) localisée, il calcule le nombre d'itérations à superposer ainsi que d'autres informations reliées à la correspondance entre le code source et le programme optimisé. Pendant l'écriture de la nouvelle boucle, toutes ces informations sont obligatoires pour la délimitation des trois parties de cette boucle : le prologue, le corps de boucle et l'épilogue. Une fois cette information connue, il est alors facile de l'enregistrer plutôt que de l'oublier en fin de compilation. Nous instrumentons le compilateur pour qu'il note une partie de cette information.

Pour chaque partie de la boucle pipelinée (figure 4.9 reprise de la section 3.3.4.2), nous instrumentons le compilateur afin qu'il enregistre quatre valeurs qui correspondent informellement aux bornes de la boucle. Nous récupérons le plus petit et le plus grand numéro d'itération pour lesquels les instructions sont exécutées et nous les appelons **first** et **last**. Étant donné que les itérations sont superposées, le plus grand numéro d'itération du prologue sera très souvent plus grand que le premier numéro d'instance d'instruction du corps de boucle. Cela va également se produire entre le corps de boucle et l'épilogue.

Comme chaque partie est faite d'un bloc d'adresses continues, nous stockons la première et la dernière adresse en les appelant **starting** et **ending**. Un exemple de telles informations est donné dans le tableau 4.3. La première affectation du prologue appartient à la première itération

	`firstIter`	`lastIter`	`startAddr`	`endAddr`
`prologue`	1	4	`0x01`	`0x07`
`loopbody`	3	N	`0x09`	`0x0B`
`epilogue`	$(N-1)$	N	`0x0D`	`0x11`

TAB. 4.3 – Informations générales de boucle

alors que la dernière affectation, en terme de numéro d'instance, appartient à la quatrième itération. Nous stockons aussi ses bornes, c'est-à-dire les adresses, ici de `0x01` à `0x07`. La portée des itérations de l'épilogue dépend du nombre total d'itérations de la boucle, et, ici, va de `N-1` à `N` tandis que son espace d'adresses va de `0x0D` à `0x11`. Ensuite, nous stockons un entier par

Numéro de la ligne source	2	3	4	5	6	7	8	9	10
Variable définie	`a`	`b`	`c`	`d`	`e`	`f`	`g`	`h`	`j`
Offset	2	1	1	0	0	2	1	0	2

TAB. 4.4 – Informations sur les instructions

		Itérations					
	Addr	1	2	3 (N-2)	4 (N-1)	5 (N)	numéro de ligne du code source
Prologue	0x01	acfj					(2,4,7,10)
	0x02	bd	fj				(3,5,7,10)
	0x03	egh	a				(6,8,9,2)
	0x04		bc	fj			(3,4,7,10)
	0x05		dg	a			(5,8,2)
	0x06		eh	b	fj		(6,9,3,7,10)
	0x07			cg	a		(4,8,2)
			for i=3 to N-2 {				(1)
	0x09			d	b		(5,3)
Corps de boucle	0x0A			eh	g	fj	(6,9,8,7,10)
	0x0B				c	a	(4,2)
			}				(11)
	0x0D				d	b	(5,3)
	0x0E				eh	g	(6,9,8)
Épilogue	0x0F					c	(4)
	0x10					d	(5)
	0x11					eh	(6,9)

FIG. 4.9 – Détail d'ordonnancement de la boucle du listing de la figure 4.3.

instruction : son offset par rapport au numéro d'itération courante dans le corps de boucle. Dans notre exemple, tableau 4.4, Le corps de boucle itère de 3 à $N-2$. À la 3^{ieme} itération, nous calculons la 5^{ieme} instance de l'opération qui affecte `a` à la ligne 2, son offset est alors égal à 2.

Si l'affectation d'une variable a lieu en avance ou en retard par rapport au moment attendu durant l'exécution, le débogueur n'essaie pas de calculer la valeur attendue en fonction du code source. Il calcule le numéro d'itération correspondant au code source afin que l'utilisateur sache exactement l'état réel d'exécution de son programme. L'idée principale est de calculer statiquement un offset entre l'itération courante de la boucle et l'instance courante d'une instruction appartenant au corps de boucle. Ceci peut être fait si l'offset d'exécution d'une instruction n'est pas modifié dynamiquement à l'exécution. De plus, cela implique que la boucle pipelinée ait la propriété 4.5 qui dit que *'L'ordre des instances d'une instructions est identique entre le code source et la boucle pipelinée.'* Notre travail sur le pipeline logiciel repose sur cette propriété d'ordre. Le modulo-pipeline logiciel implémenté par le compilateur *C* du MMDSP+ a cette propriété.

L'algorithme présenté ici a pour but de répondre à la question suivante : *pour une variable donnée à une adresse donnée, à quelle itération de la boucle source correspond la valeur courante de cette variable ?*

Nous définissons des types structurés, `t_part` et `t_address`, qui contiennent l'information collectée durant la compilation. Le type `t_part` peut-être assimilé à une structure `struct` en *C*. Il contient sous forme d'entiers l'information relative aux trois parties de la boucle pipelinée : le prologue, le corps de boucle et l'épilogue. Les variables `prologue`, `loopbody` et `epilogue` sont utilisées pour y référer. Voir les structures de la figure 4.10.

De la même manière, le type `t_address` contient l'information correspondant aux adresses

```
typedef struct {
  int *firstIter;
  int *lastIter;
  int *startAddr;
  int *endAddr;
} t_part;
```
(a) Structure `t_part`

```
typedef struct {
  char *name;
  int  *lnum;
} t_variable;
```
(b) Structure `t_variable`

```
typedef struct {
  int        *addr;
  t_variable *setVar[];
  t_variable *var;
  int        *offset;
} t_address;
```
(c) Structure `t_address`

FIG. 4.10 – Trois types définis pour manipuler les structures de données.

du code machine. Chaque adresse représente une instruction. Par conséquent, chaque adresse affecte un ensemble de variables, et nous notons `addr->setVar` cet ensemble. Il est représenté en *C* par un tableau de variables de type `t_variable`. Chaque opération correspond à une instruction source. Comme une variable ne peut être affectée qu'une seule fois par instruction de manière visible au niveau utilisateur, nous utilisons une notation simple pour l'accès au numéro de ligne `lnum` correspondant à une variable `v` de `addr->setVar` : nous écrivons `addr->v->lnum` et nous écrivons de la même manière `addr->offset`.

Nous appelons `i` la variable d'itération, sa valeur dans le prologue et l'épilogue n'a pas à être définie. Dans notre exemple, figure 4.9, l'ensemble des valeurs de `i` est $[3..N-2]$. La variable de type `t_variable` pour laquelle l'utilisateur désire la valeur est appelée `rv`. L'adresse d'arrêt de type `t_address` est notée `ba`. Par exemple, si la variable investiguée est affectée à l'adresse d'arrêt, nous utilisons la notation suivante : `rv` $\in$ `ba->setVar`. Les structures de la

Nom de variable	Type	Description	Fournie par
`ba`	`t_address`	adresse d'arrêt	les informations de débogage
`i`	`int`	numéro d'itération	l'état courant du programme
`rv`	`t_variable`	variable demandée	l'utilisateur
`prologue`	`t_part`	prologue	les informations de débogage
`loopbody`	`t_part`	corps de boucle	les informations de débogage
`epilogue`	`t_part`	épilogue	les informations de débogage

TAB. 4.5 – Variables d'entrée

figure 4.10 résument les structures de données. Le tableau 4.5 résume les entrées de l'algorithme. Le tableau 4.6 donne le format du résultat de l'algorithme. Nous utilisons également une adresse temporaire de type `t_address` appelée `tmpAddr`.

Nom de Variable	Type	Description
`result->addr`	`t_address`	Dernière adresse d'affectation `rv`
`result->inst`	`int`	Numéro d'instance d `rv`

TAB. 4.6 – Variable de sortie

Une condition a été omise en début de chacune des trois parties d'algorithme : `rv` est censée être affectée au moins une fois dans le corps de boucle, par conséquent, au moins une fois dans le corps de boucle de la boucle pipelinée au niveau logiciel.

Cette hypothèse nous autorise à écrire la condition de sortie de la boucle `while` comme étant la présence de `rv` dans l'ensemble `setVar` sans crainte d'une boucle infinie. Et puisqu'aucune

des boucles de l'algorithme n'a de boucle imbriquée et sort sur cette condition, nous pouvons être sûr que l'algorithme est linéaire c'est-à-dire en O(n) avec n le nombre d'instruction du corps de boucle source.

Nous donnons à présent l'algorithme en pseudo-code. Il permet au débogueur de répondre à notre question de départ : *À une adresse et pour une variable donnée, à quelle itération du code source correspond la valeur courante de cette variable ?*

L'adresse d'arrêt `ba` peut être dans le prologue, dans le corps de boucle ou dans l'épilogue. Nous présentons un algorithme en trois parties, chacune correspondant à une de ces parties. L'algorithme entier, le listing de la figure 4.11, pourrait être écrit de manière plus efficace, mais nous choisissons de le diviser en trois parties afin de faciliter la lecture.

```
1  ask_value (t_address ba, t_variable rv) {
2     if (ba ∈ prologue) then A1
3              // voir le listing de la figure 4.12
4     else if (ba ∈ loopbody) then A2
5              // voir le listing de la figure 4.13
6     else if (ba ∈ epilogue) then A3
7              // voir le listing de la figure 4.14
8     else
9              // ba n'est pas affectee dans la boucle.
10 }
```

FIG. 4.11 – Algorithme général

4.3.2.1 Le prologue :

Quand `ba` est dans le prologue, nous regardons d'abord en arrière si la dernière adresse est une affectation de `rv`. Ensuite nous comptons le nombre de fois qu'elle a été affectée par cette opération. Vérifier le numéro de ligne et pas seulement la variable permet de valider cette algorithme également dans le cas de multiples affectations d'une même variable dans le corps de boucle original.

4.3.2.2 Le corps de boucle :

Si `ba` est dans le corps de boucle, nous devons chercher l'adresse correspondant à l'affectation de `rv` d'une manière un peu différente. En effet, si l'arrêt se produit pendant la première itération, alors l'adresse que nous cherchons doit se trouver dans le prologue.

Une fois que nous savons où `rv` est affectée, grâce à l'offset, nous pouvons dire à quelle itération cette affectation correspond.

4.3.2.3 L'épilogue :

L'épilogue est très similaire au prologue, à cela près que nous comptons le nombre fois que `rv` a été affectée en partant de la fin, et non du début.

Algorithme A_1

```
1  tmpAddr = ba - 1
2  while (tmpAddr ≥ prologue->startAddr)
3        ∧ (rv ∉ tmpAddr->setVar) do
4       // search backward for rv
5       tmpAddr = tmpAddr-1
6  if (rv ∉ tmpAddr->setVar) then
7       result->addr = null
8       return // rv has not been assigned  yet
9  else
10      result->addr = tmpAddr
11
12 assAddr = result->addr
13
14 result->inst = prologue->firstIter
15 tmpAddr = prologue->startAddr
16 while (tmpAddr ≠ assAddr) do
17     if (rv ∈ tmpAddr->setVar)
18    ∧ (tmpAddr->rv->lnum == assAddr->rv->lnum) then
19             result->inst++
20     tmpAddr = tmpAddr + 1
```

FIG. 4.12 – `ba` est dans le prologue : `prologue`

4.4 Conclusion

Dans ce chapitre, nous avons donné les définitions du vocabulaire nécessaire à la compréhension du débogage de code optimisé. Définissant une réécriture comme une fonction prenant en entrée une représentation intermédiaire de programme, nous avons regroupé les briques élémentaires que sont les instructions par ensembles en fonction de leur correspondance avec le code source. Nous les avons étiquetées afin de tracer les éventuelles modifications qu'elles ont pu subir. Notre but n'est pas de restituer ces étiquettes mais bien de les utiliser afin de clarifier l'exécution réelle du programme optimisé que l'utilisateur cherche à déboguer. Nous avons conçu les algorithmes permettant le maintien des informations de débogage lors de l'optimisation d'un programme. Ils sont détaillés dans ce chapitre et illustré par trois exemples : la suppression de sous-expressions communes, l'inversion de boucle et le pipeline logiciel.

Le pipeline logiciel est traité à part car l'ordonnancement des instructions a des caractéristiques très particulières. Même si beaucoup de travaux ont été faits dans le domaine du pipeline logiciel, ceci constitue à notre connaissance la première tentative de débogage non-transparent de code optimisé par cette optimisation.

Pour les autres optimisations, nous proposons et utilisons une écriture ne dépendant d'aucune caractéristique différenciant les représentations intermédiaires entre elles, ce qui nous permet de les traiter de manière simple et uniforme. Dans l'écriture des algorithmes à implémenter, nous n'avons pas à nous soucier de savoir si ces algorithmes seront implémentés à haut ou à bas niveau. Nous allons voir au chapitre suivant l'intégration de tous ces éléments dans les outils présentés au chapitre 3.

Algorithme A_2

```
1  if (i ≤ loopbody->startIter) then
2      tmpAddr = ba - 1
3      while (tmpAddr ≥ prologue->startAddr)
4           ∧ (rv ∉ tmpAddr->setVar) do
5        // search backward for rv
6        tmpAddr = tmpAddr - 1
7
8      if (rv ∉ tmpAddr->setVar) then
9              result->addr = null
10             // rv has not been assigned yet
11             return
12     else
13             result->addr = tmpAddr
14 else
15     tmpAddr = ba - 1
16     while (tmpAddr ≥ loopbody->startAddr)
17              ∧ (rv ∉ tmpAddr->setVar) do
18               tmpAddr = tmpAddr-1
19     if (rv ∉ tmpAddr->setVar) do
20           tmpAddr = loopbody->endAddr
21           while (tmpAddr≥loopbody->startAddr)
22                ∧ (rv ∉ tmpAddr->setVar) do
23                   tmpAddr = tmpAddr - 1
24     result->addr = tmpAddr
25
26 assAddr = result->addr
27
28 if assAddr > ba then
29     result->inst = i + assAddr->rv->offset
30 else
31     result->inst = i - 1 + assAddr->rv->offset
```

FIG. 4.13 – `ba` est dans le corps de boucle : `loopbody`

Algorithme A_3

```
1  tmpAddr = ba - 1
2  while (tmpAddr ≥ prologue->startAddr)
3       ∧ (rv ∉ tmpAddr->setVar) do
4          tmpAddr = tmpAddr-1  // search backward for rv
5  result->addr = tmpAddr
6
7  assAddr = result->addr
8
9  result->inst = epilogue->lastIter
10 tmpAddr = epilogue->endAddr
11 while (tmpAddr ≠ assAddr) do
12         if (rv ∈ tmpAddr->setVar)
13        ∧ (tmpAddr->rv->lnum == assAddr->rv->lnum) then
14                 result->inst--
15         tmpAddr = tmpAddr - 1
```

FIG. 4.14 – `ba` est dans l'épilogue : `epilogue`

Chapitre 5

Expérimentations

Bohr bug (bogus vulgaris ou bogue classique):
Bogue qui, contrairement aux heisenbugs, ne disparaît pas, ni n'a ses caractéristiques modifiées lorsqu'il est recherché.

Sommaire

5.1 Introduction

La validation que nous nous proposons de faire est celle du principe de débogage de code optimisé de manière non-transparente à travers l'utilisation de notre débogueur. De manière générale, un débogueur est un outil utilisé pour inspecter l'état d'un programme lors de son exécution. Contrairement à un profileur qui va collecter des informations lors de l'exécution, le débogueur permet d'arrêter le programme à un endroit donné puis d'en inspecter l'état courant. Les mécanismes essentiels sont l'interruption d'exécution et l'observation de l'état du programme. Le débogueur doit être capable de faire le lien, à la demande de l'utilisateur, entre le code initialement écrit, pouvant provenir de plusieurs fichiers textuels, et le programme binaire exécuté. Nous pouvons mesurer la qualité d'un débogueur à sa capacité à fournir les bonnes commandes de manipulation du programme ; la qualité du débogage est elle directement liée à l'utilisateur de l'outil.

À la section 4.2.5, nous avons défini les informations nécessaires à fournir afin de répondre aux attentes du développeur de programmes embarqués. Pour expérimenter cette approche, nous avons instrumenté le compilateur `mmdspcc` et le débogueur `IDbug` de STMicroelectronics en conséquence.

Le compilateur *C* du MMDSP+ est modulaire et sa chaîne d'optimisation est paramétrable. Il est basé sur l'atelier de développement de compilateurs ACE-CoSy [ACE03] dans lequel un module est appelé moteur (voir la section 3.3.1). Après étude de ses optimisations et de leurs influences sur le programme compilé, nous en avons sélectionné une partie et l'avons instrumentée de manière à ce qu'elle génère les informations de débogage nécessaire à notre débogueur.

Un débogueur fournit une aide à la compréhension de la correspondance entre le code source et le fichier exécutable. Il n'existe pas de métrique permettant de quantifier cette aide car elle est propre à chaque utilisateur. Le contexte industriel dans lequel nous nous trouvons nous impose un outil fournissant des commandes similaires à celles de `gdb` (`run`, `break`, `delete`, `continue`, `step`, `finish`, `display`, `print`, etc ...). Nous sommes partis de l'infrastructure de construction de débogueur `IDbug` [GPR+05] proposant une implémentation classique de ces commandes. Nous les avons ré-implémentées afin qu'elles prennent en compte la correspondance que nous proposons au chapitre 4.

Dans la section 5.2, nous présentons et justifions les choix d'optimisations effectués lors l'instrumentation du compilateur. Nous présentons ensuite notre débogueur à travers quatre exemples. Les trois premiers sont les observations d'optimisations isolées : d'abord CSE à la section 5.3 et l'inversion de boucle à la section 5.4 qui sont des optimisations simples illustrant bien les principes d'étiquetage des instructions, puis le pipeline logiciel [VRFS07] à la section 5.5 qui nous permet d'aborder la parallélisation des instructions. Le quatrième et dernier exemple illustre l'observation de l'exécution d'un programme optimisé à l'aide de plusieurs techniques (section 5.6).

5.2 Choix empirique des optimisations

Il s'agit de ne pas instrumenter tous les optimiseurs présents dans le compilateur. Nous présentons une instrumentation du compilateur qui nous a permis de restreindre notre champ d'expérimentation tout en le conservant significatif. La métrique choisie est celle du nombre d'instructions ajoutées et supprimées au programme. Cette métrique nous permet d'ordonner les optimisations non plus en terme de type de modifications, mais en terme d'impact sur le programme original. Nous ne nous intéressons pas aux performances du programme final.

La figure 5.1 est un schéma simplifié de l'ordonnancement des optimisations dans le compilateur *C* du MMDSP+. Il s'agit d'une séquence d'optimiseurs dont chaque élément a accès à une représentation du programme et peut, si prédéfini, passer plusieurs fois.

Le compilateur `mmdspcc` est composé de 139 moteurs. Une partie de ces moteurs est composée d'analyseurs divers et de réparateurs d'informations tels que les arbres de dominance et de post-dominance par exemple. Une partie de ceux qui modifient le programme n'impliquent pas de changements dans les informations de débogage. Le tableau 5.1 présente le classement des optimisations ajoutant (resp. supprimant) des instructions à la CCMIR en fonction de l'importance de l'ajout (resp. suppression). Par exemple, la réduction de force (ligne 2 du tableau 5.1) ajoute ou réécrit 16 lignes au code source. L'inlining (ligne 19), ajoute 27.272 lignes et en supprime 2.596.

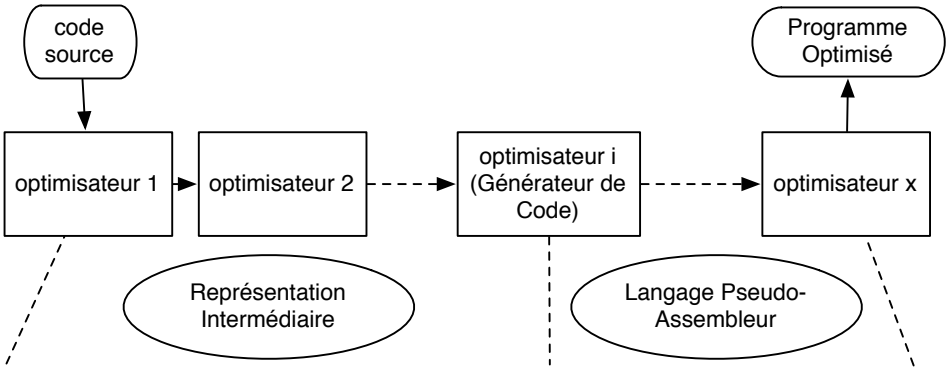


FIG. 5.1 – La séquence des optimiseurs accédant à une représentation du programme.

La suite de tests du tableau 5.2 représente un total de plus de 1.124.000 lignes de code source effectives selon la définition de RSM^{TM} :

Définition 5.1 (eLOC) *On appelle ligne de code source effective C/C++toute ligne du code source qui n'est ni commentée, ni vide, ni composée d'une unique parenthèse.*

Ce classement a été effectué lors de la compilation de la suite de tests du MMDSP+ composée des programmes du tableau 5.2 après instrumentation du `mmdspcc`. Schématisée par la figure 5.2, cette instrumentation a consisté à encadrer chaque moteur par un module permettant alternativement de sauvegarder l'état du programme puis de comparer l'état courant avec l'état précédemment enregistré. Nous avons fait ressortir les données du tableau 5.1 après analyse des résultats de cette implémentation.

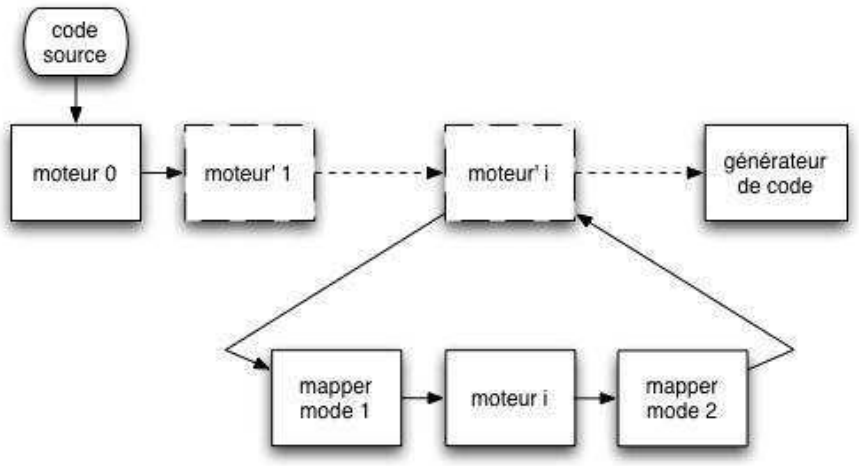


FIG. 5.2 – Instrumentation du MMDSP+ pour la collecte des données synthétisées par le tableau 5.1

Le moteur supprimant le plus d'instructions est celui fusionnant les blocs de base consécutifs sans branchement : l'instruction de saut d'un bloc à l'autre est supprimée. La normalisation des

	Fonction du Moteur	**Nombre d'ajouts et de réécritures**	**Nombre de suppressions**
1	Simplification de l'allocation des variables de pile	12	6
2	Réduction de force	16	
3	Optimisations de boucle (code sinking/code hoisting)	18	18
4	Suppression de boucle jamais exécutée	23	23
5	Simplification des `struct` retour de fonction	34	34
6	Utilisation de pointeurs en retour de fonction	149	51
7	Propagation de constantes	160	
8	Divers déplacements de code	267	4.776
9	Réécriture de boucle matérielle	342	
10	Inversion de boucle	658	
11	Déplacement d'opérations post-incrémentales	1.190	
12	Réécriture de `switch`	1.645	318
13	Réécriture de booléen	1.800	99
14	Réécriture de copies de `struct`	2.988	332
15	Initialisation de variables globales en PIC (resp. PID)[1]	4.464	
16	Reconnaissance de motifs pour l'utilisation d'intrinsics	12.523	6.169
17	selregset	22.307	
18	Transformation de boucle `while`↔`do_while`	22.719	
19	Inlining de fonction	27.272	2.596
20	Création de boucles matérielles	28.326	41.835
21	Diverses Optimisations (déplacements, CSE)	39.587	
22	Réécriture de boucle sous forme canonique	46.973	14.217
23	arT	54; 872	
24	postopizer	61.986	25.204
25	Normalisation de boucle et ajout des annotations	226.759	108.376
26	Suppression de code mort		87.149
27	Fusion de blocs de base contiguës		173.999
	TOTAL	557.090	465.202

TAB. 5.1 – Ajout et suppression d'instructions de la CCMIR lors de la compilation par le `mmdspcc` de la suite de tests composée des programmes du tableau 5.2, soit un total de plus de 1.124.000 lignes de code source.

boucles avec ajout d'annotations revient à réécrire une boucle de manière à ce quelle ait au moins trois blocs de bases, éventuellement vides : le bloc contenant les instructions d'initialisation, le bloc contenant les instructions d'incrémentation, et le bloc contenant le saut conditionnel. Ainsi il est normal que ces deux moteurs se situe en tête des moteurs ajoutant et supprimant des instructions.

La forte présence de manipulations de tableaux dans les codecs multimedia explique la forte utilisation de ArT (ligne 23), présenté section 3.3.3.1.. La création de boucles matérielles et l'utilisation des intrinsics sont aussi décrits sections 3.3.3.3 et 3.3.5.

La réécriture de boucles sous forme canonique revient à réécrire les boucles de manière à ce que les annotations soient plus facilement ajoutées par le moteur de normalisation de boucle.

Le reste des optimisations telles que décrites dans le tableau peuvent paraître redondantes, mais elles sont classées par moteur et non par modifications. Ainsi on retrouve la suppression de code mort et la suppression de boucle jamais exécutée de manière disjointe.

AMR	(Adaptive Multi Rate)	COdeur/DÉCodeur
Arkamys		
G.723.1	Vocoder	codec
Pro Logic Decoder		codec Dolby
SBC		codec under Bluetooth®
MP3	(Motion Picture Expert Group Audio Layer 3)	codec
MIDI	(Musical Instrument Digital Interface)	codec
AAC	(Advanced Audio Coding)	codec
EFR	(Enhanced Full Rate)	Firware mode (e.g. Nokia 3310)

TAB. 5.2 – Suite de tests

5.3 Un premier exemple : CSE

Le but de ce premier exemple est de présenter notre implémentation sur un programme simple. Mis-à-part CSE, toutes les optimisations sont désactivées à la compilation. Le fichier binaire généré est chargé par notre débogueur et manipulé avec des commandes simples. Deux listings reproduisant les sorties du débogueur sont commentés afin de faire ressortir l'utilité de notre méthode.

Nous reprenons figure 5.3 l'exemple de la section 4.3.1.1. Rappelons que le résultat de l'expression `c * d` est stocké dans une variable temporaire `tmp` puis réutilisé aux lignes 2 et 4 du code ainsi optimisé. Une fois compilée, cette portion de code a pour information de débogage

s_0^0 : `a = b + c * d;`
s_1^0 : `e = a + b;`
s_2^0 : `g = f + c * d;`

(a) Code source

s_0^1 : `tmp = c * d;`
s_1^1 : `a = b + tmp;`
s_2^1 : `e = a + b;`
s_3^1 : `g = f + tmp;`

(b) Après CSE

FIG. 5.3 – Listings de code source avant et après modification par la suppression de sous-expression commune — CSE

relatives la fonction du tableau 5.3.

$$
\begin{array}{lcl}
M^1 & = & (m^1, K^1) \\
m^1 & = & ((s_0^0, s_0^1, CSE1), \\
 & & (s_2^0, s_0^1, CSE1), \\
 & & (s_0^0, s_1^1, CSE2), \\
 & & (s_1^0, s_2^1,), \\
 & & (s_2^0, s_3^1, CSE2)) \\
K^1 & = & ((s_0^0, s_1^1), (s_1^0, s_2^1), (s_2^0, s_3^1))
\end{array}
$$

TAB. 5.3 – M^1 (après la suppression d'expressions communes)

L'utilisation de la représentation de la fonction M^1, tableau 5.3, est exhibée à travers les figures 5.4 et 5.5. Dans les deux cas, après avoir instancié un point d'arrêt à la ligne 15, le programme est démarré. Quand il s'arrête au niveau du premier point d'arrêt, il affiche également les informations relatives à l'instruction d'arrêt tel qu'un débogueur standard le ferait.

Dans le cas de la figure 5.4, nous avançons ensuite le programme de trois pas avant d'arrêter l'exécution du débogueur. La séquence d'exécution est la suivante : `b myTest.c :15 ; run ; step ; step ; step ; quit ;` Nous avons vu à la section 4.2 qu'une instruction clé est celle qui colle au plus près de la sémantique de l'instruction source correspondante (définition 4.2). C'est pourquoi le débogueur n'instancie pas un point d'arrêt sur s_0^1 (voir la listing de la figure 4.7(b) page 80). Après avoir instancié un point d'arrêt sur l'instruction de la ligne 15, un pas en avant mène l'exécution jusqu'à l'instruction clé suivante (s_1^1), c'est-à-dire ligne 16 (voir figure 5.4).

```
1 (idbug) b myTest.c:15
2  Breakpoint 1 : file myTest.c, line 15.
3 (idbug) run
4  Starting program: /tmp/debug/example/myTest.elf
5  Breakpoint 1, in main () at myTest.c:15
6  15         int f = 2;
7 (idbug) step
8  16         a = b + c * d;
9          Subexpression hoisted.
10 (idbug) step
11  17         e = a + b;
12 (idbug) step
13  18         g = f + c * d;
14          Subexpression hoisted.
15 (idbug) quit
```

FIG. 5.4 – Comportement dit *correct* [Zel83] avec un pas d'exécution de la ligne 15 à la ligne 16.

Dans le cas de la figure 5.5, après l'arrêt de l'exécution du programme sur la ligne 15, nous avançons d'abord d'une opération machine avant d'avancer le programme de trois pas puis d'arrêter l'exécution du débogueur. La séquence d'exécution est alors la suivante : `b myTest.c :15 ; run ; stepi, step ; step ; step ; quit ;` Quand nous faisons avancer le programme d'une instruction machine en avant, le débogueur affiche l'information de localité correcte, c'est-à-dire l'instruction machine suivante correspond à l'affectation de la variable temporaire. Ce que nous appelons les étiquettes *CSE1* et *CSE2* sont ici explicitées : elles correspondent respectivement aux messages *Common subexpression stored in a temporary variable.* et *Subexpression hoisted.*.

Dans ces deux cas d'utilisation, la présentation de l'exécution réelle du programme est faite grâce à l'affichage d'informations sur l'optimisation apportée au programme lors de sa compilation. L'utilisation d'étiquettes explicites permet de comprendre que l'instruction qui suit la définition de la variable `f` est l'opération `c*d`. Supposons qu'une erreur soit soulevée par cette instruction commune, sans informations relatives à ses origines dans le source le débogage deviendrait complexe.

5.4 Un second exemple : l'inversion de boucle

Notre second exemple est l'utilisation de notre outil sur une portion de code optimisée par inversion de boucle. Tout comme pour CSE, nous voulons montrer ici comment l'ajout d'informations lors de la session de débogage rend compréhensible l'exécution du programme. Nous avons la aussi désactivé les optimisations lors de la compilation, sauf celle que nous cherchons

```
1 (idbug) b myTest.c:15
2  Breakpoint 1 : file myTest.c, line 15.
3 (idbug) run
4  Starting program: /tmp/debug/example/myTest.elf
5  Breakpoint 1, in main () at myTest.c:15
6  15        int f = 2;
7 (idbug) stepi
8  16        a = b + c * d;
9  18        g = f + c * d;
10         Common subexpression stored in a temporary variable.
11 (idbug) step
12 16        a = b + c * d;
13         Subexpression hoisted.
14 (idbug) step
15 17        e = a + b;
16 (idbug) step
17 18        g = f + c * d;
18         Subexpression hoisted.
19 (idbug) quit
```

FIG. 5.5 – Comportement dit *correct* [Zel83] avec un pas d'exécution non plus de la ligne 15 à la ligne 16 mais à l'opération suivante, instruction `stepi`

à observer : l'inversion de boucle.

Le code, précédemment commenté section 4.3.1.2, est celui de la figure 5.6. Ainsi qu'expliqué dans la section 2.8, cet optimiseur fait partie des plus perturbants dans le cadre d'une session de débogage non-transparent car il inverse l'ordre d'itération.

```
s0^0 : for (i=0; i<MAX; ++i){
s1^0 :     sum += x[i];
s2^0 : }
```

(a) Code source

```
s0^1 : for (i=MAX-1; i<=0; --i){
s1^1 :     sum += x[i];
s2^1 : }
```

(b) Après inversion de boucle

FIG. 5.6 – Listings de code source avant et après modification par inversion de boucle

La fonction M^1 dont la représentation correspond aux informations de débogage est donnée par le tableau 5.4.

$$\begin{array}{lcl} M^1 & = & (m^1, K^1) \\ m^1 & = & ((s_0^0, s_0^1, LR1), \\ & & (s_1^0, s_1^1, LR2), \\ & & (s_2^0, s_2^1,)) \\ K^1 & = & ((s_0^0, s_0^1), (s_1^0, s_1^1), (s_2^0, s_2^1)) \end{array}$$

TAB. 5.4 – M^1 (après inversion de boucle)

La figure 5.7 montre une vue utilisateur du débogueur lors de l'instanciation d'un point d'arrêt sur une instruction d'une boucle inversée à la compilation. Après avoir instancié un point d'arrêt à la ligne 20, l'utilisateur démarre le programme. Ce dernier s'arrête sur la ligne

en indiquant la modification subie : *Loop reversal optimisation applied*. Après avoir fait un pas d'exécution en avant, le débogueur est dans le corps de la boucle inversée, donc les instructions sont étiquetés $LR2$. Nous avons traduit cette étiquette dans notre prototype par l'information suivante : *Instruction from a loop reversed by the compiler*.

```
1 (idbug) b myTest.c:20
2  Breakpoint 1 : file myTest.c, line 20.
3 (idbug) run
4  Starting program: /tmp/debug/example/myTest.elf
5  Breakpoint 1, in main () at myTest.c:21
6  21        for (i=0; i<10; ++i){
7         Loop reversal optimisation applied
8 (idbug) step
9  22             sum += x[i];
10        Instruction from a loop reversed by the compiler
11 (idbug) quit
```

FIG. 5.7 – Annotation informative d'une boucle inversée.

5.5 Un troisième exemple : le pipeline logiciel

Les problèmes relatifs à la localité du code et les problèmes relatifs aux données sont observables avec un débogueur classique lors de l'inspection de l'exécution d'une boucle préalablement pipelinée au niveau logiciel. Nous avons vu aux sections 3.3.4.2 et 4.3.2 que le pipeline logiciel instaure une forte confusion pour ce qui est de la compréhension de l'exécution du programme. Nous proposons de raffiner l'information sur l'optimisation qui peut ainsi être plus complète et non plus simplement indicative. Notre utilisation de ces informations comme conteneur plus complexe permet de répondre à la question suivante : *Quand, à une adresse donnée, la valeur d'une variable est donnée dans une boucle pipelinée au niveau logiciel, pouvons-nous savoir à quelle numéro d'itération cette même valeur aurait été identique si la boucle n'avait pas été pipelinée ?*

L'idée de la non-transparence telle que définie par P. Zellweger [Zel83] est de ne rien cacher au programmeur mais plutôt de le guider lors de la session de débogage. Quand l'affectation d'une variable a été reportée à l'exécution, l'information fournie nous permet de connaître non seulement la valeur courante de la variable, mais également le délai, positif ou négatif, instauré par le pipeline logiciel de la boucle contenant cette affectation. Nous avons nommé ce délai *offset* (définition 4.6).

Le listing de la figure 5.8(a) contient une boucle `while` qui calcule une suite de sommes à travers la variable `b`, et stocke les résultats successifs dans le tableau `c`. Nous faisons ensuite afficher un certain résultat[2].

Le listing de la figure 5.8(b) contient la boucle pipelinée écrite en langage assembleur. La boucle a d'abord été transformée en boucle matérielle, ceci s'observe à la ligne 13 par la présence de l'instruction `repi` (voir la section 3.3.3.3). Le corps de boucle, de la ligne 16 à la ligne 23 est donc répété 29 fois. Ce corps de boucle est composé d'un mot contenant trois instructions : les

[2]Ce programme contient vraiment un bogue, saurez vous le retrouver ?

lignes 19 et 18 du code source. Le prologue est composé d'une instruction : ligne 10. L'épilogue est composé de deux instructions en langage assembleur : lignes 29 et 30.

Nous exécutons ce programme dans notre débogueur, figure 5.9. Nous cherchons à montrer le décalage instauré par le compilateur entre la valeur attendue et la valeur réelle de la variable `b`. Pour ce faire, un point d'arrêt est instancié sur l'entrée de la boucle (ligne 16 du code source) par la commande `break 16` (ligne 1 du listing de la figure 5.9). Ensuite, nous repassons par deux fois par ce point, ce qui correspond à exécuter 2 fois l'incrémentation de la variable `b`. Or, à la demande d'affichage de sa valeur, commande `print`, l'affectation ayant été dupliquée dans le prologue de la boucle, la valeur de `b` correspond à celle de la troisième itération et non à la seconde comme réellement exécutée et donc attendue.

Dans le cas du pipeline logiciel, l'observation de l'exécution en *boite noire* ne permet pas de comprendre les états successifs du programme. L'ajout d'informations sur l'optimisation apportée permet derechef de saisir les subtilités nécessaires à son débogage. L'affichage combiné des variables et de leurs décalages met en lumière l'ordonnancement de la boucle. Sans être un expert de ce type d'optimisations ni des architectures permettant l'ILP, le programmeur de logiciels embarqués a les connaissances nécessaires qui, complétées par les informations du débogueur, lui permettent de comprendre les états internes successifs du programme lors de son exécution.

```
11  int a = 30;
12  int b = 0;
13  int c[30];
14
15  int i = 0;
16  while ( i<a )
17    {
18      b = b + 1;
19      c[i] = b;
20      i = i + 1;
21    }
22
23  afficher(b,c[a]);
```

(a) Code source

```
1  L4:
2  .begin_bundle
3          mea     sp0,--#30
4  ;; Line 12 of file myTest.c
5          sub     r2h,r2h,r2h
6  .end_bundle
7  .begin_bundle ;; prologue
8          xmv     sp0,ax1 ;; Variable c
9  ;; Line 18 of file myTest.c
10         add     r2h,#1,r2h
11 ;; Loop pipelined with II=1, SC=2 (RecMII=1, ResMII=1)
12 ;; Loop not pipelined (1 cycles, RecMII=1, ResMII=1)
13         repi    PIPEL6,#29
14 .end_bundle
15 L5:
16 .begin_bundle ;; loop body
17 ;; Line 19 of file myTest.c
18         stx_f   r2h,ax1
19         mea     ax1,++#1
20 ;; Line 18 of file myTest.c
21         add     r2h,#1,r2h
22 .end_bundle
23 PIPEL6:
24 .begin_bundle
25         lea     #30+sp0,sp1     ;; Variable c
26 .end_bundle
27 .begin_bundle ;; epilogue
28 ;; Line 19 of file myTest.c
29         stx_f   r2h,ax1
30         mea     ax1,++#1
31 .end_bundle
32 .begin_bundle
33         ldx_f   sp1,r2l ;; Variable c
34 .end_bundle
35 .begin_bundle
36         jsr     _afficher
37 .end_bundle
```

(b) Extrait du fichier généré par le compilateur dans le cas d'une boucle pipelinée. Les commentaires relatifs aux trois parties de la boucle ont été, eux, rajoutés après.

FIG. 5.8 – Exemple simple de pipeline logiciel.

```
1 (idbug) break 16
2  Breakpoint 1: file myTest.c, line 16.
3 (idbug) run
4  Starting program
5  Reading symbols for shared libraries . done
6
7  Breakpoint 1 at myTest.c:16
8  18                    while ( i<a ) {
9  WARNING: software-pipelined loop:
10           2 iterations overlapped
11 (idbug) continue 2
12 Will ignore next 2 crossings of breakpoint 1.
13 Continuing.
14
15 Breakpoint 1 at myTest.c:16
16 16                    while ( i<a ) {
17 WARNING: software-pipelined loop:
18           2 iterations overlapped
19 (idbug) print b
20 b = 3
21 WARNING: software-pipelined loop:
22           accurate for iteration (i == 3)
```

FIG. 5.9 – Un exemple de débogage du programme du listing de la figure 5.8(a).

5.6 Présentation d'une étude de cas

La méthodologie utilisée pour trouver un bogue dans un programme est propre à chaque développeur. Par exemple, que ce dernier ait écrit ou non le code source fait que les choix d'instanciation de points d'arrêt ne seront pas du tout les mêmes que s'il n'en faisait que le support. Sa connaissance du code source influencera ses choix lors de l'investigation du programme.

Présenter nos expérimentations sur des exemples simples tels que ceux que nous venons de présenter ne donne qu'un aperçu. Nous avons présenté trois optimisations, indépendamment les unes des autres. Le compilateur ne choisit pas une optimisation qu'il applique mais plusieurs qu'il entrelace, réitère. Ainsi le programme subit un nombre de passes d'optimisations bien plus important que ce que nous avons présenté jusqu'à présent.

Nous avons choisi de ne pas utiliser un programme issu de la suite de tests dans la mesure où aucun ne contenait de portion de code permettant de présenter autant d'optimisations en si peu de lignes. Notre exemple est lui un condensé de ce que l'on trouve en général dans les codecs du tableau 5.2.

Le but de cette section n'est pas de dérouler une session de débogage en la présentant comme classique, ou typique. Nous présentons un programme susceptible d'être optimisé par le `mmdspcc`. Nous exécutons le fichier binaire généré dans notre débogueur et instancions une série de points d'arrêt. La présentation des sorties est ensuite commentée.

Nous savons que le `mmdspcc` propose un large panel d'optimisations dont les principales sont présentées en annexe A. Nous avons classifié ces optimisations. Section 4.3.1, nous avons dégagé trois groupes d'optimisations. Dans chacun de ces groupes, et en fonctions de ce que le `mmdspcc` proposait, nous avons décidé d'instrumenter les optimisations suivantes :

1. les optimisations qui suppriment du code. Les cas typiques sont les suppressions de code inutile et inaccessible
   – la suppression de **code mort**,
2. les optimisations qui déplacent des portions de code sans modifier les instructions.
   – le déplacement d'opérations invariantes hors d'un corps de boucle (**code hoisting**, **code sinking**)
3. les optimisations qui réécrivent les instructions.
   – la suppression de sous-expressions communes (**CSE**),
   – la création de **boucles matérielles**,
   – l'ordonnancement de type **pipeline logiciel**,
   – l'utilisation de l'arithmétique de pointeur à travers **arT**,
   – la **propagation de constantes**,
   – l'**inversion de boucle**,
   – la **réduction de force**,
   – le **déroulage de boucle**

L'ensemble des optimiseurs sélectionnés est pertinent car il recouvre toutes les modifications applicables au code source par les optimiseurs, à savoir suppression, duplication, réécriture, réordonnancement. L'expérimentation sur l'inversion de boucle permet d'observer les modifications faites à l'ordre des instances des instructions. De plus, cet ensemble comprend les moteurs les plus utilisés par notre compilateur `mmdspcc`.

Le code source initial est celui du listing de la figure 5.10. Il est composé d'une simple fonction `main` (ligne 6) et fait appel à trois fonctions externes non présentées ici : les fonctions `initVariables` (ligne 9), `afficher1` (ligne 27) et `afficher2` (ligne 19). Après initialisation de toutes les variables aux lignes 7, 8 et 9, nous trouvons deux boucles imbriquées. La première est une boucle `for` allant de la ligne 10 à la ligne 20 et dont le nombre d'itérations est `MAX`, à savoir 10 (ligne 2). Elle contient la seconde qui est une boucle `do-while` allant de la ligne 13 à la ligne 18 et dont le nombre d'itérations est `b`. Suivent ensuite une boucle `for` (lignes 21 à 23) et un branchement conditionnel de type `if` dépendant de la variable `verbose` (lignes 24 à 26). Après l'appel de la fonction `afficher1` de la ligne 27, la fonction retourne la valeur 0 (ligne 28).

Les optimisations appliquées modifient le flot d'exécution du programme. Le lecteur reconnaîtra certains exemples pris de manière isolée dans les exemples précédents. Les lignes 14 et 16 ont une sous-partie commune, l'opération `c*d`. L'accès au tableau `x` dans la boucle `for` ligne 21 est un candidat idéal pour l'utilisation d'arithmétique de pointeur. Le listing de la figure 5.10 est un exemple permettant de mettre en exergue les optimisations sélectionnées pour notre étude.

Les illustrations de l'utilisation de notre débogueur sont fournies par les deux listings subséquents.

```
1 (idbug) break 11
2  Breakpoint 1: file testexemple.c, line 11.
3 (idbug) break 22
4  Breakpoint 2: file testexemple.c, line 22.
```

```
1     #include <stdio.h>
2     #define MAX 10
3
4     typedef enum { FALSE, TRUE} bool;
5
6     int main(int argc, char *argv[]) {
7       bool verbose = FALSE;
8       int a, b, c, d, e, f, g, i, k, sum, x[MAX];
9       initVariables(a, b, c, d, e, f, g, i, k, x, sum);
10      for (k = 0; k<MAX; ++k) {
11        sum = 0;
12        i = 0;
13        do {
14          a = a + c*d + 1;
15          x[i] = a;
16          f = f + c*d;
17          i = i + 1;
18        } while ( i<b );
19        afficher2(a,x[a-1]);
20      }
21      for (i=0; i<MAX; ++i) {
22        sum += x[i];
23      }
24      if (verbose) {
25        printf("Calcul␣de␣la␣somme␣terminee.\n") ;
26      }
27      afficher1(sum);
28      return 0;
29    }
```

FIG. 5.10 – Code source principal.

```
5  (idbug) run
6   Starting program
7   Reading symbols for shared libraries . done
8
9   Breakpoint 1 at testexemple.c:11
10  11                      sum = 0;
11          Code hoisted.
12 (idbug) continue
13  Continuing.
14
15  Breakpoint 2 at testexemple.c:22
16  22                      sum += x[i];
17          Pointer to array used. See mmdspcc/arT documentation.
18 (idbug)
```

Ce listing illustre une utilisation de notre outil avec le programme de la figure 5.10. Nous lançons le fichier binaire dans notre outil puis instancions deux points d'arrêt sur les utilisations de la variable `sum` : lignes 11 et 22.

```
1 (idbug) break 15
```

```
 Breakpoint 1: file testexemple.c, line 15.
(idbug) break 16
 Breakpoint 2: file testexemple.c, line 16.
(idbug) run
 Starting program
 Reading symbols for shared libraries . done

 Breakpoint 1 at testexemple.c:15
 15                      x[i] = a;
        Loop transformed into a while-loop.
 WARNING: software-pipelined loop:
          2 iterations overlapped
(idbug) continue
 Continuing

 Breakpoint 2 at testexemple.c:16
 16                      f = f + c*d;
        Loop transformed into a while-loop.
        Subexpression hoisted.
 WARNING: software-pipelined loop:
          accurate for iteration (i == 0)
(idbug) disable 1
(idbug) continue
 Continuing

 Breakpoint 2 at testexemple.c:16
 16                      f = f + c*d;
        Loop transformed into a while-loop.
        Subexpressiion hoisted.
 WARNING: software-pipelined loop:
          accurate for iteration (i == 1)
(idbug)
```

Ce listing illustre une utilisation de notre outil avec le programme de la figure 5.10. Nous lançons le fichier binaire dans notre outil puis instancions un points d'arrêt sur l'affectation de la variable `x` : ligne 15.

5.7 Impact de l'ajout des informations de débogage

Nous avons implémenté notre méthode de maintien et de propagation des informations de débogage dans le compilateur *C* du MMDSP+, fourni par STMicroelectronics. Le fichier exécutable généré est ainsi augmenté de la taille de ces informations. Nous avons observé une augmentation de la taille du fichier binaire d'environs 30% en moyenne, mais le code du programme n'est en rien modifié car notre méthode est non-intrusive Ainsi le temps d'exécution n'est pas modifié non plus. De plus, notons que les informations de débogage peuvent être dans un fichier indépendant du fichier contenant le code machine du programme, tout comme le fait MS-VisualC++.

Les modifications nécessaires à la lecture de ces informations ont été portées à `IDbug`, l'infrastructure de construction de débogueur fournie par STMicroelectronics elle aussi. Lors de la session de débogage l'augmentation de la taille des informations de débogage a un impacte

négligeable car ces dernières sont lues à la demande.

5.8 L'intégration aux outils d'STMicroelectronics

Le choix du développement d'un débogueur de code optimisé s'est fait dans le contexte de création de `IDbug` [GPR+05]. L'équipe IDTEC compte dans ses rangs des mainteneurs et contributeurs du GNU Debugger, ce qui confère à STMicroelectronics une expertise dans le domaine du débogage. L'approche non-transparente que nous avons choisie et implémentée a été partiellement concertée avec les ingénieurs de l'équipe IDTEC et de l'équipe CEC. L'instrumentation du compilateur s'est faite sur une version du `mmdspcc` produite dans le courant du premier trimestre de l'année 2006. Après avoir figé cette version, nous avons procédé à des expérimentations sur les modules ainsi que sur les bibliothèques des sources du compilateur. Les corrections de l'équipe de support du compilateur ainsi que les mises-à-jour n'ont pas été reportées sur notre version du compilateur. Le développement sur `IDbug` a été fait dans notre branche du gestionnaire de version `svn`. L'intégration de notre travail ne sera pas fait en l'état car l'instrumentation était expérimentale. Elle a nécessité certaines modifications du logiciel initial qui demanderaient une réingénierie plus profonde pour prétendre à une production industrielle. En revanche, la politique de STMicroelectronics encourage le dépôt de brevet. Une proposition a donc été rédigée dans ce sens [VR07] et est en attente de validation par le service juridique de l'entreprise.

Eclipse IDE[3] est un environnement de développement ouvert. Écrit principalement en Java, il est facilement extensible et propose notamment des mécanismes d'interfaçage avec des outils tels que `gdb`. La spécificité d'Eclipse IDE vient de son architecture totalement développée autour de la notion de *plug-in* : toutes les fonctionnalités de cet atelier logiciel sont développées en tant que *plug-in*.

Nous avons présenté ce que nos travaux nous ont amenés à modifier dans la CLI — *Command Line Interface* — du débogueur `IDbug`. La modification de la MI — *Machine Interface* — suivait logiquement dans la phase de développement. Nous avons développé la vue utilisateur permettant de tirer parti des modifications apportées à la MI. Elle est composée de cinq fenêtres. En haut à gauche sont affichées les informations sur l'exécution du programme, en haut à droite les informations relatives aux variables du programme. Viennent ensuite les codes source et optimisé. Ce dernier est représenté à partir du fichier de sortie du `mmdspcc`, il contient donc certains commentaires ajoutés par le compilateur au code assembleur. Contrairement à ces quatre fenêtres, la dernière fenêtre n'est pas présente dans la version standard du plug-in. Elle permet l'affichage des informations relatives à une ligne sélectionnée.

- La valeur ajoutée de cette interface par rapport à notre outil dans la console est la visualisation de plusieurs lignes lors de correspondances multiples entre le code source et le programme optimisé.
- La valeur ajoutée de cette interface par rapport à l'interface classique de débogage d'Eclipse est qu'elle permet de renseigner le développeur sur l'exécution de son programme sans en modifier le code source.

Nous avons repris figure 5.11 l'exemple de la section 5.5 : une boucle simple optimisée par pipeline logiciel. La boucle de la fenêtre de gauche intitulée `myTest.c` présente le code source. Un point d'arrêt a été instancié à la ligne 18. L'exécution est démarrée puis interrompue à la

[3] `http://www.eclipse.org`

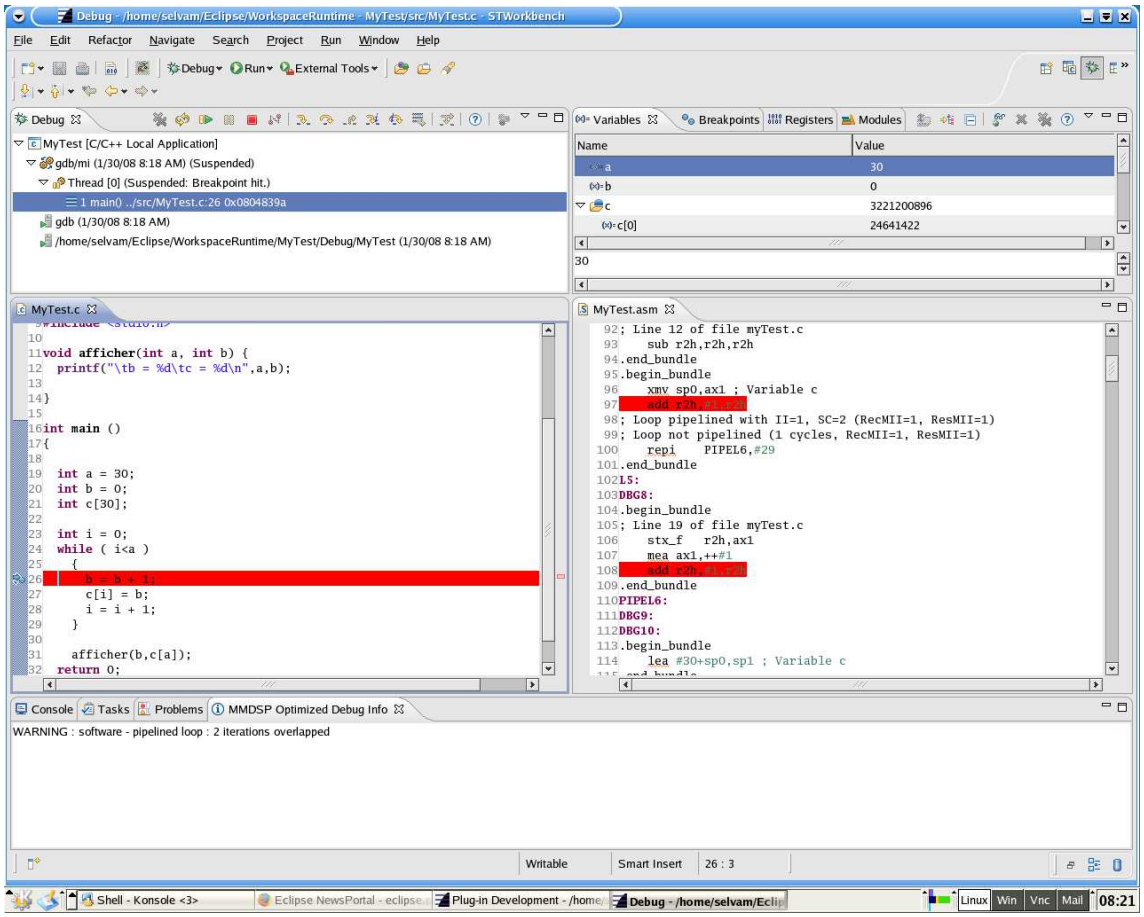


FIG. 5.11 –
Impression d'écran de la vue d'Eclipse lors de l'exécution de l'exemple de la section 5.5.

rencontre du point d'arrêt. L'instruction d'arrêt est surlignée dans le code source, instruction d'incrémentation de la variable `b`, ainsi que les opérations correspondantes dans le codeur assembleur. La fenêtre du bas contient un onglet *MMDSP Optimized Debug Info* reprenant les informations déjà fournies par la CLI.

Notre implémentation n'a pas été distribuée comme a pu l'être celle de C. Tice qui ajoute en annexe de sa thèse les retours utilisateurs. Le contexte a fait que nous n'avons eu aucune interaction avec de réels utilisateurs du `mmdspcc` ou de `IDbug`.

5.9 Conclusion

Notre proposition de développement nécessite une certaine connaissance des optimisations de la part du développeur de compilateur mais également de la part de l'utilisateur du débogueur.

Le développeur de compilateur doit, au moment de l'implémentation d'un module d'optimisation, également maintenir les informations de débogage. Il s'agit d'utiliser la bibliothèque que nous avons développée afin de manipuler un système d'annotations. Le développeur a une connaissance approfondie de l'optimisation qu'il implémente et donc des informations à maintenir afin de suivre notre proposition. L'effort demandé est donc minime.

L'utilisateur du débogueur doit avoir une certaine connaissance des optimisations de code en général, et de celles appliquées par le compilateur qu'il utilise en particulier. Pouvant être considéré de prime abord comme une restriction d'utilisation, il s'agit en fait de notre intention de ne rien cacher. Cela permet ainsi à l'utilisateur d'en savoir plus sur le comportement de son programme sur la plate-forme sur laquelle il développe. Le développeur de programmes embarqués a en principe une bonne connaissance et compréhension des informations additionnelles que nous lui fournissons.

Ce chapitre a abordé plus spécifiquement le débogage non-transparent de boucles pipelinées au niveau logiciel. Nous avons proposé un algorithme pour que le débogueur rende de manière non-transparente à l'utilisateur l'exécution réelle du programme grâce à l'utilisation d'informations collectées à la compilation[VRFS07]. Cette approche ne dépend que des algorithmes d'optimisation mis en jeux par le compilateur et non du processeur. À notre connaissance, ce problème n'a jamais été abordé auparavant.

Troisième partie

Conclusion et perspectives

Chapitre 6

Conclusions et perspectives

It's not a bug, it's a feature.
non-attribué

Ce chapitre clôt le manuscrit en récapitulant les objectifs proposés, les travaux réalisés et les perspectives.

6.1 Objectifs de la thèse

Bien que les optimisations effectuées par un compilateur soient cruciales pour améliorer les performances d'une application, le programmeur les désactive généralement pendant la phase de développement. La raison principale de ce choix est le manque d'outils efficaces pour le débogage de code hautement optimisé. En effet, l'optimisation du code détruit l'association qui existe entre le code source et le code binaire généré. Ainsi, par exemple, si le programmeur veut consulter la valeur d'une variable à un certain point, le debogueur peut ne pas être en mesure de lui donner une valeur correcte si le calcul de la valeur a été différé par une optimisation et n'est donc pas disponible. Il devient également extrêmement difficile de définir 'un point' du programme exécutable par rapport au code source, car les instructions générées pour chaque ligne de code vont se retrouver mélangées dans le code optimisé.

Il peut sembler acceptable de désactiver les optimisations durant le développement d'un programme, mais le fait est que la plupart des programmes mis en production vont être compilés optimisés. Le comportement du programme optimisé est susceptible de changer et donc d'engendrer des effets pervers :

- Une optimisation peut révéler un bogue n'apparaissant pas avec du code non optimisé.
- Une optimisation peut elle-même être boguée.

Ces points montrent l'intérêt de pouvoir travailler sur du code optimisé pendant la phase de développement. Dans le monde du logiciel embarqué, cet intérêt est même une nécessité car c'est le programme final qui sera embarqué et qui, par conséquent, doit être validé. Par ailleurs, des contraintes supplémentaires viennent s'y ajouter à cause des limitations du matériel et du type d'applications exécutées. Une demande de la part des développeurs existe donc pour un support du code embarqué hautement optimisé.

6.2 Travaux réalisés

Au chapitre 2, nous avons investigué le domaine du débogage de code optimisé, depuis J. Hennessy [Hen82] jusqu'à nos jours. L'étude des différentes publications et réalisations nous a permis de définir les deux grandes orientations pour lesquelles nous pouvions opter : la *transparence* et la *non-transparence*.

Le concept de transparence laisse entendre qu'il n'y aura pas de différence entre le débogage de code optimisé et le débogage du même code non-optimisé, ce qui est contre-intuitif. Que le programme soit optimisé ou non, le suivi d'exécution est le même. Bien que séduisante, cette approche s'avère peu réaliste dans le contexte industriel. Le *continuum* de C. Tice reproduit figure 2.9 est explicite quant à la finalité de ces deux approches. Nous choisissons la **non-transparence**, il s'agit de trouver une solution pour permettre le débogage de code optimisé sans cacher les détails de l'exécution du programme. En d'autres mots, contrairement à l'outil de C. Jaramillo [Jar00], notre débogueur ne renverra pas les mêmes informations au développeur s'il travaille sur le programme optimisé ou non.

Il existe deux grandes classes de problèmes. La première classe regroupe les problèmes relatifs aux données du programme (section 2.3.1). Quand le développeur veut observer la valeur d'une variable, le débogueur doit répondre à trois questions :

1. La valeur de cette variable existe-t-elle quelque part ?
2. Si oui, dans quel registre et/ou espace mémoire peut-on la trouver ?
3. À quelle réalité correspond cette valeur ?

Les deux premières questions ont des réponses techniques venant de partisans de la transparence et de la non-transparence. Un ensemble de solutions notables a émergé des travaux de A. R. Adl-Tabatabai [AT96]. La *troisième* et dernière question est celle du contexte. Il s'agit pour le débogueur d'être capable de justifier cette valeur et donc de savoir si l'on cache la réalité à l'utilisateur ou pas. Nous proposons, dans le cas particulier du pipeline logiciel, une solution à ce problème [VRFS07]. À notre connaissance, cette optimisation a toujours été supprimée des ensembles d'optimisations utilisés à des fins expérimentales.

La seconde classe est celle des problèmes relatifs à la localité des instructions (section 2.3.2). Les instructions sont ordonnancées relativement les unes aux autres. Les raisons peuvent être dépendantes du matériel utilisé (la plupart des algorithmes profitant de l'ILP proposé par le processeur), ou non (la majorité les manipulations de représentations intermédiaires de haut niveau). Le point commun à toutes ces raisons est que l'ordre d'instanciation des points d'arrêt sera différent en fonction des optimisations apportées au programme et différent de celui spécifié dans le code source de l'utilisateur. Il s'agit des ordres d'instanciation de points d'arrêt sémantiques et syntaxiques. Traiter la classe de problèmes relatifs à la localité du code revient à chercher l'information à retenir afin de faire correspondre ces deux ordres. Quand la correspondance est connue, il faut choisir un moyen afin de la révéler à l'utilisateur. Nous proposons une solution basée sur le concept d'*instruction clé* de C. Tice couplé à un principe d'étiquetage systématique des instructions [**?**]. Ces étiquettes sont maintenues à jour et enrichies par les différents optimiseurs tout au long de la compilation pour être ajoutées au fichier contenant les informations de débogage [VR07]. Elles sont stockées sous la forme d'une seule table de correspondance et lues à la demande par le débogueur. Cette solution peut être vue comme la superposition de la chaîne d'optimisation avec la chaîne de collecte des informations de débogage. En effet, jusqu'à présent, les informations de débogage étaient collectées en début du

processus de compilation et ajoutée après optimisation au fichier binaire. Le fait d'optimiser rendait ainsi obsolètes ces informations.

`CXdb` et `Optview` ont proposés des solutions similaires à la notre à ceci près que `CXdb`, selon ses auteurs, ne permet pas de suivre les points d'arrêt sémantiques et donc n'en permet pas une utilisation efficace dans le cas où le développeur ne connaîtrait pas exactement les optimisations et leurs implémentations dans le compilateur. `Optview` ne permet pas de retour utilisateur pour des optimisations qui ne sont pas représentables en terme de langage source. Il s'agit donc de la première définition et implémentation d'une solution au problème de la localité du code de manière non-transparente pour l'utilisateur, expérimentée sur des optimisations de haut et de bas niveau. Il s'agit également de la première expérimentation purement non-transparente de réponses à la *troisième question* relative aux contextes des valeurs de variable.

STMicroelectronics développe aujourd'hui des outils d'aide au développement logiciel, tels que compilateurs, débogueurs, et analyseurs de performance. Les expérimentations de la thèse ont été développées sur la base du compilateur `mmdspcc` et du débogueur `IDbug`. Une bibliothèque de fonctions permettant une manipulation des informations a été créée et ajoutée au compilateur `mmdspcc`. L'effort demandé au développeur d'optimiseur est minime car il s'agit d'une manipulation d'annotations de la représentation intermédiaire, or toutes les optimisations manipulent déjà ces annotations. Le développeur n'a plus qu'à en sauvegarder certaines choisies en fonctions des méthodes que nous proposons au chapitre 4. Afin de permettre une meilleure intégration au sein de la chaîne de développement, nous avons créé une vue à l'environnement Eclipse s'interfaçant avec nos outils (voir la section 5.8).

6.3 Perspectives

La correspondance entre les lignes du code source et les lignes du code désassemblé que nous proposons est augmentée d'informations relatives aux modifications appliquées. L'utilisation des étiquettes permet, de manière très simple, de faire un lien entre les deux types de problèmes relatifs au débogage de code optimisé. Mais il reste encore du travail pour ce qui est de proposer un outil que nous pourrions considérer comme complet.

La finalité de notre recherche est de proposer un outil permettant de déboguer du code optimisé. Notre constat est le suivant : il existe des solutions aux problèmes relatifs aux données tout comme il existe des solutions aux problèmes relatifs à la localité du code, toutes n'étant pas compatibles entre elles. L'approche non-transparente choisie permet au développeur de systèmes embarqués de rester proche du matériel pour lequel il écrit ses programmes car rien ne lui est caché lors de la phase de développement. Le support peut, quant à lui, manipuler le même programme binaire que l'utilisateur final du système. Nous voyons deux directions dans lesquelles ces travaux pourraient continuer, chacunes d'elles s'articulant en deux étapes.

6.3.1 Développement et intégration

La première direction est la finalisation d'un débogueur permettant le débogage de code optimisé. Cette proposition est celle d'un travail d'ingénierie.

6.3.1.1 Informations de données

Ce premier travail serait l'implémentation d'une partie des travaux de A. R. Tabatabai [AT96]. Ce dernier a principalement travaillé sur les problèmes relatifs aux données. Durant sa thèse,

il a cherché à compléter les informations de débogage en se basant sur une analyse de flot de données faite par le compilateur. Les résultats de cette analyse permettent de raffiner les informations de débogage relatives aux données. Deux booléens par variables sont ensuite calculés par le débogueur en fonction de la demande de l'utilisateur :

problème de résidence : la valeur de la variable est-elle résidente ?

problème de concordance : cette variable est-elle compromise ?

Les réponses à ces deux questions sont fournies avec la réponse à la question du *problème de la localisation* : les informations de localisation des données. Elles sont collectées par le compilateur. Ce dernier élément permettrait l'activation d'optimisations lors de l'allocation de registres. il faut bien garder en tête qu'avant de pouvoir informer l'utilisateur, le débogueur doit lui-même être capable de localiser les informations sur la plate-forme cible. L'information de ligne seule ne suffit pas pour répondre à notre question initiale.

6.3.1.2 Standardisation des informations de débogage

L'intégration de ses résultats sur l'analyse et la comparaison des flots de données permettrait de compléter nos travaux et ainsi de proposer une réponse plus globale au débogage de code optimisé.

Si cette expérimentation s'avérait concluante, l'étape suivante serait la recherche d'intégration de l'ensemble des informations nécessaires à un format d'information de débogage standard tel que DWARF 3 [1]. La mise à plat d'une utilisation de ces informations permettrait de l'implémenter dans plusieurs compilateurs et débogueurs de manière systématique et ainsi proposerait aux utilisateurs un choix d'outils plus vaste qu'un simple couple tel que le `mmdspcc/IDbug` limité à un seul processeur, le MMDSP+.

6.3.2 Le parallélisme de manière plus générale

La seconde direction est l'élargissement du champ d'investigation. Nous avons travaillé sur les optimisations destinées à un ASP-VLIW. Mais de plus en plus de produits n'intègrent plus un processeur mais plusieurs. La mise en parallèle de portions de code par le compilateur est de plus en plus fréquente. Or nous n'avons abordé qu'un type de parallélisation, celui des instructions, et ce au travers d'un seul algorithme, le pipeline logiciel.

Le deuxième travail qui nous semble intéressant serait d'étendre nos travaux à d'autres algorithmes de pipeline logiciel afin que d'autres compilateurs puissent générer les informations de débogage que nous avons définies. La pierre angulaire de ce travail serait de montrer que l'algorithme étudié respecte la propriété d'ordre 4.5.

Les résultats de ces travaux pourraient servir de base à une recherche plus large du débogage d'applications compilées avec des méthodes de parallélisation automatique. Notre travail s'est focalisé sur une forme d'ILP très particulière et dont la solution est finalement dépendante de l'implémentation dans le compilateur.

6.3.3 En dehors du monde embarqué

Notre principale motivation est notre contexte : le monde des systèmes embarqués. Il s'agit d'un monde dans lequel les systèmes sont fortement contraints, et ce sont ces contraintes qui

[1] `http://dwarfstd.org/`

poussent le développement d'optimisations toujours plus agressives. La recherche de performances des exécutions suit de près la grandissante complexité des systèmes embarqués ainsi que leur miniaturisation. Le débogage de ces programmes demande aux systèmes qui les embarquent une certaines réactivité lors, par exemple, de l'instanciation de points d'arrêt logiciels. L'augmentation du nombre de points d'arrêt diminue drastiquement la réactivité des débogueurs face au mécanisme ce type de point d'arrêt.

Sur des machines bien moins contraintes, comme les machines de bureau par exemple, l'adaptation de nos travaux au débogage transparent de code optimisé serait possible en utilisant des points d'arrêt invisibles à l'utilisateur. C. Jaramillo [Jar00] en fait déjà l'utilisation dans ses travaux sans aborder d'optimisations de bas niveau. Les informations ajoutées et/ou enrichies par notre compilateur sont rendues lisible pour l'humain utilisateur. La recherche de principes de communication d'informations du compilateur vers le débogueur permettrait à ce dernier de savoir comment émuler le programme d'origine en fonction de la modification appliquée, et ainsi rendrait possible, sous certaines conditions, le débogage de code optimisé de manière transparente.

Bibliographie

[ACE03] ACE Associated Compiler Experts bv. CoSy Compilers, Overview of Construction and Operation. White paper, 24 April 2003.

[ALSU07] Alfred V. Aho, Monica Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2007.

[App98] Andrew W. Appel. *Modern Compiler Implementation in C.* Cambridge University Press, 1998.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1986.

[AT92] Ali-Reza Adl-Tabatabai. Nonresident and endangered variables : The effects of code generation optimizations on symbolic debugging. Technical report, Pittsburgh, PA, USA, 1992.

[AT96] Ali-Reza Adl-Tabatabai. *Source Level Debugging of Globally Optimized Code.* PhD thesis, Carnegie Mellon University, Pittsburgh PA 15213-3891, June 1996.

[ATG93] Ali-Reza Adl-Tabatabai and Thomas Gross. Detection and recovery of endangered variables caused by instruction scheduling. In *PLDI : Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation*, volume 28, pages 13–25, New York, NY, USA, June 1993.

[AU77] Alfred V. Aho and Jeffrey D. Ullman. *Compilers : Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1977.

[BDG+02] Valérie Bertin, Jean-Marc Daveau, Philippe Guillaume, Thierry Lepley, Denis Pilat, Claire Richard, Miguel Santana, and Thomas Thery. Flexcc2 : An optimizing retargetable c compiler for dsp processors. In *EMSOFT '02 : Proceedings of the Second International Conference on Embedded Software*, pages 382–398, London, UK, 2002. Springer-Verlag.

[BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4) :345–420, 12 1994.

[BHS92] Gary Brooks, Gilbert J. Hansen, and Steve Simmons. A new approach to debugging optimized code. In *PLDI : Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation*, volume 27, pages 1–11, New York, NY, USA, July 1992. ACM Press.

[BS94] Gary S. Brooks and Steven M. Simmons. Debugger program which includes correlation of computer program source code with optimized object code. Patent number : 5371747, dec 1994.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4) :451–490, Oct 1991.

[CM90] Deborah S. Coutant and Sue A. Meloy. Patent : Method and apparatus using variable ranges to support symbolic debugging of optimized code. Patent number : 4953084, Aug 1990.

[CM93] Max Copperman and Charles E. McDowell. A further note on hennessy's "symbolic debugging of optimized code". *TOPLAS : ACM Transactions on Programming Languages and Systems*, 15(2) :357–365, 1993.

[CMR88] Deborah S. Coutant, Sue A. Meloy, and Michelle Ruscetta. Doc : a practical approach to source-level debugging of globally optimized code. In *PLDI : Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation*, volume 23, pages 125–134, New York, NY, USA, July 1988. ACM Press.

[Coo92] Lyle E. Cool. Debugging VLIW Code After Instruction Scheduling. Master's thesis, Oregon Graduate Institute of Science and Technology, 1992.

[Cop93] Max Copperman. *Debugging optimized code without being misled.* PhD thesis, University of California, Santa Cruz, Pasadena, CA, USA, 1993.

[Dow97] Gilles Dowek. *Le langage mathématique et les langages de programmation. Voir, entendre, raisonner, calculer.* Cité des sciences et de l'industrie, La Villette, Paris, 1997.

[DS98] D. M. Dhamdhere and K. V. Sankaranarayanan. Dynamic currency determination in optimized programs. *TOPLAS : ACM Transactions on Programming Languages and Systems*, 20(6) :1111–1130, 1998.

[DTLS04] Jean-Marc Daveau, Thomas Thery, Thierry Lepley, and Miguel Santana. A retargetable register allocation framework for embedded processors. In *LCTES '04 : Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, Compilers, and Tools for Embedded Systems*, pages 202–210, New York, NY, USA, 2004. ACM Press.

[Duc99] Mireille Ducassé. Coca : An automated debugger for C. In *Proceedings of the 21st International Conference on Software Engineering*, pages 504–513, New York, NY, USA, May 1999. ACM Press.

[DWA] DWARF Debugging Information Format Workgroup, A Workgroup of the Free Standards Group, http ://dwarf.freestandards.org/. *The DWARF Debugging Standard.*

[ECR99] Christine Eisenbeis, Zbigniew Chamski, and Erven Rohou. Flexible Issue Slot Assignment for VLIW Architectures. In *SCOPES : Proceedings of the international workshop on Software and Compilers for Embedded Systems*, March 1999.

[Elm97] Kim Elms. Debugging optimised code using function interpretation. In *Automated and Algorithmic Debugging*, pages 27–36, 1997.

[GA96] Lal George and Andrew W. Appel. Iterated register coalescing. In *POPL : Proceedings of the ACM SIGACT-SIGPLAN symposium on Principles Of Programming Languages*, pages 208–218, St. Petersburg Beach, Florida, 21–24 January 1996.

[GLE94] John Gough, Jeff Ledermann, and Kim Elms. Interpretive debugging of optimised code. In *the Proceedings of ACSC-17, Christchurch*, 1994.

[GPR+05] Laurent Gerard, Denis Pilat, Frederic Riss, Sylvaine Laheurte, Miguel Santana, and Hugo Venturini. IDBug Technology, Benefits and Added-value. White Paper, STMicroelectronics, F-38921 Crolles France, jul 2005.

[Gui99] Philippe Guillaume. *Contribution aux aspect dorsaux de la synthèse de systèmes monopuces*. PhD thesis, INPG, Crolles, France, 1999.

[Hen82] John Hennessy. Symbolic debugging of optimized code. *TOPLAS : ACM Transactions on Programming Languages and Systems*, 4(3) :323–344, July 1982.

[HMC+93] W.M.W. Hwu, S.A. Mahlke, W.Y. Chen, P.P. Chang, N.J. Warter, R.A. Bringmann, R.G. Ouellette, R.E. Hank, T. Kiyohara, G.E. Haab, et al. The superblock : An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1) :229–248, 1993.

[HP04] HP. Compiler technical overview. Technical report, HEWLETT-PACKARD COMPANY United States of America, 2004.

[Jar00] Clara I. Jaramillo. *Source Level Debugging Techniques And Tools For Optimized Code*. PhD thesis, University of Pittsburgh, 2000.

[JD99] Erwan Jahier and Mireille Ducassé. A generic approach to monitor program executions. In D. De Schreye, editor, *Proceedings of the International Conference on Logic Programming*. MIT Press, November 1999.

[JGS99] Clara Jaramillo, Rajiv Gupta, and Mary Lou Soffa. Comparison checking : An approach to avoid debugging of optimized code. In Oscar Nierstrasz and Michel Lemoine, editors, *ESEC/FSE '99*, volume 1687 of *Lecture Notes in Computer Science*, pages 268–284. Springer-Verlag / ACM Press, 1999.

[JGS00] Clara Jaramillo, Rajiv Gupta, and Mary Lou Soffa. Fulldoc : A full reporting debugger for optimized code. In *Static Analysis Symposium*, pages 240–259, 2000.

[JGS02] Clara Jaramillo, Rajiv Gupta, and Mary Lou Soffa. Debugging and testing optimizers through comparison checking. *Electr. Notes Theor. Comput. Sci*, 65(2), 2002.

[LA04] Chris Lattner and Vikram Adve. LLVM : A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[PS02] Pierre G. Paulin and Miguel Santana. Flexware : A retargetable embedded-software development environment. *IEEE Design & Test of Computers*, 19(4) :59–69, 2002.

[Rau94] B. Ramakrishna Rau. Iterative modulo scheduling : an algorithm for software pipelining loops. In *MICRO : Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74, New York, NY, USA, 1994. ACM Press.

[RG81] B. Ramakrishna Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *MICRO : Proceedings of the 14th Annual Workshop on Microprogramming*, pages 183–198, October 1981.

[RKT78] Dennis M. Richie, Brian W. Kernighan, and Ken Thompson. The C Programming Language. *Bell System Technical Journal*, 57 :1991–2019, 1978.

[SB92] Philip H. Sweany and Steven J. Beaty. Dominator-path scheduling — A global scheduling method. In *MICRO : the 25th Annual International Symposium on Microarchitecture*, pages 260–263, 1992.

[SBB+91] Larry V. Jr. Streepy, Gary Brooks, Tussell Buyse, Mark Chiarelli, Mike Garzione, Gil Hansen, Dave Lingle, Steve Simmons, and Jeff Woods. Cxdb a new view on optimization. In *Proceedings of Supercomputer Debugging Workshop*, pages 1–22, November 1991.

[SPS+02] Richard M. Stallman, Roland Pesch, Stan Shebs, et al. *Debugging with GDB : The GNU Source-Level Debugger*. Free Software Foundation, 2002.

[Sta99] Richard M. Stallman. *Using and Porting the GNU Compiler Collection, For GCC Version 2.95*. Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA, Tel : (617) 876-3296, USA, 1999.

[Swe92] Philip H. Sweany. *Inter-Block Code Motion without Copies*. PhD thesis, Fort Collins, CO, USA, December 27 1992.

[TG98] Caroline M. Tice and Susan L. Graham. Optview : A new approach for examining optimized code. In *PASTE : Workshop on Program Analysis For Software Tools and Engineering*, pages 19–26, 1998.

[TG00] Caroline M. Tice and Susan L. Graham. Key instructions : Solving the code location problem for optimized code. 2000.

[Tic99] Caroline M. Tice. Non-transparent debugging of optimized code. Technical report, University of California at Berkeley, Berkeley, CA, USA, November 1999.

[VR07] Hugo Venturini and Frédéric Riss. Patent : Method and apparatus for debugging information propagation through compilation. STATUS *pending*, 2007.

[VRFS07] Hugo Venturini, Frédéric Riss, Jean-Claude Fernandez, and Miguel Santana. Non-transparent debugging for software-pipelined loops. In *CASES : Proceedings of the international conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 23–32, New York, NY, USA, oct 2007. ACM Press.

[VRFS08] Hugo Venturini, Frédéric Riss, Jean-Claude Fernandez, and Miguel Santana. A fully-non-transparent proposal to the code location problem. In *Proceedings of SCOPES '08 : The 11th International Workshop on Software and Compilers for Embedded Systems*. ACM Press New York, NY, USA, march 2008.

[Wis94] Roland Wismüller. Debugging of globally optimized programs using data flow analysis. *ACM SIGPLAN Notices*, 29(6) :278–289, 1994.

[WMP+99] Le-Chun Wu, Rajiv Mirani, Harish Patil, Bruce Olsen, and Wen mei W. Hwu. A new framework for debugging globally optimized code. In *PLDI : Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 181–191, New York, NY, USA, 1999. ACM Press.

[WmWW98] Le-Chun Wu and Wen mei W. Whu. A new breakpoint implementation scheme for debugging globally optimised code. *Urbana*, 51 :61801, 1998.

[WST85] David Wall, Amitabh Srivastava, and Fred Templin. A note on hennessy's "symbolic debugging of optimized code". *TOPLAS : ACM Transactions on Programming Languages and Systems*, 7(1) :176–181, January 1985.

[Zel83] Polle T. Zellweger. An interactive high-level debugger for control-flow optimized programs. In *SIGSOFT '83 : Proceedings of the symposium on High-level debugging*, volume 18, pages 159–172, New York, NY, USA, August 1983. ACM Press.

[Zel84a] Polle T. Zellweger. *Interactions between high-level debugging and optimised code*. PhD thesis, Computer Science Division, University of California, Berkeley, 1984.

[Zel84b] Polle T. Zellweger. Interactive source-level debugging of optimized programs. Research Report CSL-84-5, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304, May 1984.

Annexe A

Les optimisations dans le compilateur *C* du MMDSP+

Nous présentons ici les principales optimisations sur lesquelles nous avons travaillé. Nous ne redonnons pas le pipeline logiciel qui est largement discuté dans le corps du manuscrit mais détaillons le déroulage de boucle qui en est une composante.

Nous avons occasionnellement privilégié la compréhension du type d'optimisation à une illustration plus réaliste mais moins standard. Le décalage de bit dans le cas de la réduction de force est par exemple bien plus générique. Les exemples de portions de code optimisés utilisés dans les sections de cette annexe ne sont pas toujours des cas réalistes de modifications effectuées par le `mmdspcc`.

A.1 La réduction de force

Certaines opérations sont moins coûteuses en terme d'utilisation du processeur que d'autres. Le compilateur va alors quand cela est possible réécrire certaines opérations et utiliser ces opérations. Cette optimisation est en général dépendante de la machine cible. Mais il existe des cas valables sur la grande majorité des architectures. Par exemple, le décalage de bits est souvent moins coûteux que la multiplication. Si le compilateur peut remplacer une multiplication par une puissance de 2 par un décalage de bits, alors il va le faire.

A.2 La propagation de constantes

Il s'agit ici de remplacer le chargement d'une variable dont la valeur est constante par la valeur elle même. Le gain se fait alors sur le temps de chargement de la variable.

Prenons un programme qui aurait une variable globale `verbose` initialisée une fois en début de programme puis simplement utilisée dans des portions de code de la forme :

```
if (verbose) fprintf(stdout, ``Message'');
```

L'analyse du CFG retourne le fait que la variable `verbose` n'est définie qu'une seule fois dans le programme. En reprenant les notations de la section 2.2.3, cela s'écrit :

$\forall s_i, s_j \in \mathcal{S}^2$, `verbose` $\in def(s_1) \wedge$ `verbose` $\in def(s_2) \Rightarrow i = j$

Alors tous les accès à `verbose` sont remplacés par sa valeur initiale.

A.3 L'inversion de boucle

Il s'agit de changer le sens de parcours en le faisant décroître (resp. croître) s'il croissait (resp. décroissait) initialement. L'intérêt d'une telle optimisation est variable. Dans notre cas, le MMDSP+ propose un mécanisme de boucles matérielles si elles sont décroissantes avec un pas de 1. Afin d'accélérer le parcours de boucle, le compilateur va donc chercher à avoir le maximum de boucles à décrément. Le listing des figures A.1(a) et A.1(b) donne un exemple d'un tel changement. Le parcours de la boucle `for` ne se fait plus de 0 à 10 mais de 11 à 1, ce

```
for (i=0; i<MAX; ++i){
  sum += x[i];
}
```

(a) Code source

```
for (i=MAX-1; i<=0; --i){
  sum += x[i];
}
```

(b) Après inversion de boucle

FIG. A.1 – Listings de code source avant et après modification par inversion de boucle

qui inverse donc également l'ordre des instances des instructions du corps de boucle. La liste des valeurs de la variable `idx` n'est plus croissante, mais décroissante.

A.4 La réécriture d'éléments complexes

Cette optimisation transforme l'affectation à un élément composé en une boucle potentiellement déroulée, qui effectue la copie élément par élément. Une fois la boucle écrite, elle sera soumise aux optimiseurs de boucle suivant dans la chaîne des optimiseurs.

A.5 L'utilisation des idiomes de la machine : *intrinsics*

Certains processeurs dit ASP pour Application Specific Processor proposent un ensemble de fonctions non standards, c'est-à-dire que les autres processeurs ne proposent pas forcément mais qui permettent d'effectuer certaines opérations spécifiques plus rapidement. Généralement, cela se traduit par la réécriture d'un ensemble d'instructions en une unique instruction par le générateur de code. Les fonctions min et max ou le cas des boucles matérielles que nous décrivons à la section A.9 en sont des exemples.

A.6 La réécriture de `while-do` en `repeat-until` et réciproquement

La réécriture d'une boucle en une autre se fait dans le cas où cela permet l'application d'une autre optimisation comme la création d'une boucle matérielle par exemple.

La différence entre ces deux types de boucle réside dans la première itération. Une boucle `repeat-until` exécute son corps de boucle au moins une fois alors qu'une boucle `while-do` peut ne jamais l'exécuter. Si le compilateur peut statiquement caractériser cette différence alors la réécriture de l'une en l'autre est possible.

A.7 L'inlining de fonction

Le but de cette optimisation est de remplacer un appel de fonction par une copie de cette fonction. Ainsi les surcoûts de l'appel sont évités, le changement de contexte n'est pas à faire : les instructions `call` et `return`, la sauvegardes des registres, le passage de paramètres et l'ajustement de la pile sont éventuellement évités.

Cette optimisation a été étudiée par P. Zellweger lors de ses travaux sur les optimisations de haut-niveau et le débogage de code optimisé [Zel84a].

A.8 La suppression de portions de code

Il existe plusieurs cas où la suppression d'une portion de code se justifie. Le cas le plus typique est un branchement conditionnel dont on pourrait statiquement déterminer la valeur de la condition. Le listing qui suit en est un exemple où le compilateur peut savoir que la fonction `f()` ligne 3 n'est jamais exécutée et il peut donc supprimer l'instruction d'appel afin de réduire la taille finale du programme généré. Les quatre lignes initiales sont en partie supprimée pour n'en laisser qu'une, la ligne d'assignation de `i`.

```
1  i = 2;                          i = 2;
2  if (i < 0) {                    // ...
3    f();
4  }
```

A.9 La création de boucles matérielles

Certains processeurs fournissent des mécanismes de boucles matérielles. Le jeu d'instructions du MMDSP+ contient l'instruction `repi` prenant en paramètre un nombre et une étiquette.

Dans le cas où cette optimisation est désactivée, une instruction plus classique telle que `for(int i = 0 ; i < MAX ; i++)` est, à la génération de code, réécrite en un ensemble d'instruction contenant entre autres le saut conditionnel, l'incrémentation du compteur `i`, le stockage de ce compteur et son initialisation. Dans le cas où cette optimisation est activée, le compilateur va chercher à réécrire cette même instruction en une seule instruction de type *boucle matérielle* et ainsi faire un gain certain de temps de calcul lors de l'exécution de cette boucle. Ce gain est d'autant plus significatif que le nombre d'itérations de la boucle est élevé.

A.10 Le déplacement d'invariant(s) de boucle

Un corps de boucle est composé d'un ensemble d'opérations. Certaines ne dépendent pas de l'itération, par exemple il peut s'agir d'une affectation de constante. L'exemple que nous présentons section 2.3.1 et reprenons ici est peu réaliste mais représentatif de l'optimisation dont nous parlons ici. À la fin de l'exécution de cette portion de code, la variable `tmp` a pour valeur la somme des valeurs de `INIT` a `MAX`. La variable `start` a elle pour valeur la constante `INIT`. À chaque itération, `start` a la même valeur, afin d'optimiser l'exécution de cette boucle, nous pouvons déplacer la ligne 4 avant ou après la boucle. Il s'agit respectivement du *code hoisting* et du *code sinking*.

```
1  tmp = INIT;
2  for (cpt = 0; cpt < MAX; ++cpt) {
3    tmp = tmp + cpt;
4    start = INIT;
5  }
```

(a) Boucle originale

```
1  tmp = INIT;
2  start = INIT;
3  for (cpt = 0; cpt < MAX; ++cpt) {
4    tmp = tmp + cpt;
5  }
```

(b) Code Hoisting

```
1  tmp = INIT;
2  for (cpt = 0; cpt < MAX; ++cpt) {
3    tmp = tmp + cpt;
4  }
5  start = INIT;
```

(c) Code Sinking

FIG. A.2 – Déplacement d'instruction hors du corps de boucle.

A.11 La réécriture de conditions booléennes

La réécriture de conditions booléennes est une modification qui consiste à réécrire une expression de la forme `exp1 ? exp2 : exp3` en une instruction `if` de forme standard.

A.12 La suppression de sous-expressions communes

La suppression de sous-expressions communes est une optimisation qui cherche à réduire le nombre de calculs. Il arrive que plusieurs instructions fassent la ou les mêmes opérations. Le listing de la figure A.3 présente un cas où l'expression commune se trouve entre les calculs des variables `a` et `g`. Sans optimisation, le compilateur va naïvement traduire ce programme tel que présenté par le listing de la figure A.3(b). Si la suppression de sous-expressions communes est activée, le compilateur va alors réécrire le programme afin qu'il ne fasse ce calcul qu'une fois. Le listing de la figure A.3(c) présente cette réécriture. Le résultat de l'opération `c * d` sera stocké dans une variable temporaire `tmp` introduite par le compilateur. Toutes les instructions faisant ce calcul sont réécrites de manière à utiliser directement le résultat pré-calculé.

```
1  a = b + c * d;
2  e = a + b;
3  g = f + c * d;
```

(a) Code source original

```
1  tmp1 = c * d;
2  a = b + tmp1;
3  e = a + b;
4  tmp2 = c * d;
5  g = f + tmp2;
```

(b) Sans CSE

```
1  tmp = c * d;
2  a = b + tmp;
3  e = a + b;
4  g = f + tmp;
```

(c) Avec CSE

FIG. A.3 – Suppression de sous-expressions communes

A.13 La réécriture d'accès tableaux en arithmétique de pointeurs

Cette optimisation a été développée par P. Guillaume dans le cadre de sa thèse CIFRE au sein d'STMicroelectronics [Gui99]. Il s'agit de remplacer, quand cela est possible, un accès tableau en une arithmétique de pointeur moins coûteuse en ressources. L'exemple que P. Guillaume prend dans sa thèse est celui de la figure A.4. Appelée ArT, son implémentation au sein du `mmdspcc` est décrite section 3.3.3 page 49.

```
1 sum = 0;
2 for (j = 0; j < ORDER; j++) {
3   sum = sum + a[i+j] * b[j];
4 }
```

(a) Accès tableau

```
1 sum = 0;
2 ptrb = b;
3 ptra = \&a[i];
4 for (j = 0; j < ORDER; j++) {
5   sum = sum + *ptra * *ptrb;
6   ptrb = ptrb + 1;
7   ptra = ptra + 1;
8 }
```

(b) Manipulation de pointeurs

FIG. A.4 – arT : La réécriture d'accès tableau en arithmétique de pointeurs

A.14 Le déroulage de boucle

Le déroulage de boucle est une optimisation qui va permettre de réduire un certain nombre d'opérations telles que l'évaluation de la condition de sortie de la boucle ou encore le calcul du compteur de boucle. De plus, cette optimisation va permettre au compilateur d'appliquer plus efficacement certaines optimisations et algorithmes d'ordonnancement telles que le pipeline logiciel par exemple.

Nous pouvons distinguer deux types de déroulage de boucle. Le premier, *loop unrolling* consiste à effectivement démultiplier le nombre de fois qu'est exécuté le corps de boucle par itération de la boucle. Si la boucle suivante est déroulée une fois :

```
1 for (i=0; i<MAX; i+1) {
2   f();
3 }
```

elle devient alors

```
1 for (i=0; i<MAX; i+2) {
2   f();
3   f();
4 }
```

La variable de boucle `i` n'est incrémentée d'un pas de 1 mais d'un pas de 2 après le déroulage. Il a y bien entendu des conditions à satisfaire. Dans notre exemple, le nombre d'itérations doit être pair.

Le second type est le déroulage partiel, *loop peeling*. Certaines itérations de la boucle sont sorties du corps de boucle et exécutée avant ou après la nouvelle boucle. Dans notre exemple, le résultat de la boucle partiellement déroulée d'une itération serait le suivant :

```
1  f();
2  for (i=1; i<MAX; ++i) {
3    f();
4  }
```

Une fois de plus, des conditions sont à satisfaire. Dans notre cas, le compilateur doit s'assurer que le nombre d'itérations de la boucle source est strictement supérieur à zéro.

A.15 La scalarisation

Il s'agit de l'utilisation d'une variable locale temporaire à la place d'une variable globale. Cette optimisation n'a pas d'incidence directe, mais ainsi le compilateur, lors de l'allocation de registres, peut conserver cette variable temporaire uniquement en registre et ainsi faire un gain de temps d'exécution par la suppression d'instructions de chargement et de stockage en mémoire de la variable.

Nous reprenons ici l'exemple donnée comme illustration section 2.3.1. La variable `a` est remplacée par une variable temporaire le temps du calcul par la boucle.

```
1  int f() {
2    int x;
3    int i;
4    // ...
5
6    a = a + x;
7    for (i=0; i<10; ++i) {
8      a = a + i;
9    }
10
11   // ...
12   return 0;
13 }
```

(a) Avant scalarisation

```
1  int f() {
2    int x;
3    int i;
4    // ...
5    tmp = a;
6    tmp = tmp + x;
7    for (i=0; i<10; ++i) {
8      tmp = tmp + i;
9    }
10   a = tmp;
11   // ...
12   return 0;
13 }
```

(b) Après scalarisation

FIG. A.5 – Remplacement d'une variable globale par une variable temporaire : la scalarisation

Annexe B

Fichier *en* de messages d'IDbug

Les étiquettes générées par le compilateur sont des entiers que le débogueur va rechercher dans un fichier préalablement chargé. Le listing suivant est un extrait du fichier *en* permettant l'affichage des informations en anglais. Le débogueur propose ainsi le changement de langue en fonction de la demande de l'utilisateur.

```
0:UNKNOWN
1:Initial

2:HWLOOPCREATE: test condition inverted
3:HWLOOPCREATE: if (testexpr) { goto thenblock; } else { goto elseblock; }
4:HWLOOPCREATE: replaces an occurrences of 'i' by 'updated(i)'
5:HWLOOPCREATE: Replace Test (mirIf) by mirEndLoop
6:HWLOOPCREATE: Replace enter loop (mirGoto) by mirBeginLoop

20:arT: Converting init expr to the alias type
21:arT: inc. expr.
22:arT: Update sourceBB control statement: it was a goto-stmt
23:arT: Update sourceBB control statement: changing the 'then'
24:arT: Update sourceBB control statement: changing the 'else'
25:arT: new assign stmt
26:arT: incr. stmt.
27:arT: incrementation balance stmt
28:arT: goto of the new BB which contains a control statement.
29:arT: Converting incr expr to the alias type
30:arT: initialization of the pointer
31:arT: referencing an assignment in the loop body
32:arT: replacing by a pointer
33:arT: incrementation of the pointer
34:arT: extraction of a part of stmt

40:LOOPANALYSIS: added by LoopAnalysis
41:LOOPANALYSIS: duplicated by LoopAnalysis
42:LOOPANALYSIS: duplicated by LoopAnalysis
43:LOOPANALYSIS: duplicated by LoopAnalysis
```

```
44:LOOPANALYSIS: added by LoopAnalysis
45:LOOPANALYSIS: added by LoopAnalysis
46:LOOPANALYSIS: added by LoopAnalysis
47:LOOPANALYSIS: added by LoopAnalysis
48:LOOPANALYSIS: added by LoopAnalysis

50:LOWERBOOLVAL: if-STMT added by LowerBoolVal
51:LOWERBOOLVAL: Then-STMT added by LowerBoolVal
52:LOWERBOOLVAL: Else-STMT added by LowerBoolVal
53:LOWERBOOLVAL: End-STMT added by LowerBoolVal

60:StructCopy: *pS1 = *pS2
61:StructCopy: pS1 += PtrInc
62:StructCopy: pS2 += PtrInc
63:StructCopy: LoopCpt++
64:StructCopy: if LoopCpt<IterationsNb then goto LoopBodyBB else goto nextBB
65:StructCopy: pS1 = &S1
66:StructCopy: pS2 = &S2
67:StructCopy: LoopCpt = 0
68:StructCopy: pS2 += PtrInc
69:StructCopy: pS1 += PtrInc
70:StructCopy: *pS1 = *pS2
71:StructCopy: Edge from currentBB to BBtoModify
72:StructCopy: pS1=(convert)pS1
73:StructCopy: pS2=(convert)pS2
74:StructCopy: Edge from currentBB to LoopBodyBB
75:StructCopy: pS1=(convert)pS1
76:StructCopy: pS2=(convert)pS2
77:StructCopy: LoopCpt = 0
78:StructCopy: goto LoopBodyBB

80:MISC: inline of function's return
81:MISC: if statement reduction
82:MISC: if statement reduction
83:MISC: Added hwloop.c (begin)
84:MISC: Added hwloop.c (goto)
85:MISC: Added hwloop.c (end)
86:MISC: duplicated by misc.c
87:MISC: strcpy
88:MISC: do_assign
89:MISC: duplicated by MISC (assign of PureFuncCall)

90:LOOPREV: incr stmt
91:LOOPREV: init stmt
92:LOOPREV: update stmt
```

```
95:LOOPREMOVE: 'begin'created
96:LOOPREMOVE: 'end' created
97:LOOPREMOVE: 'exit' created
98:LOOPREMOVE: created by 'rewrite_loop'

100:LOOPMISC: duplicated

110:HWLOOPBREAK: restore stmt
111:HWLOOPBREAK: mirgoto stmt

120:DEADCODE: deadcode_side
```

Résumé

Les optimisations jouent un rôle majeur dans la compilation des programmes embarqués. Elles interviennent à tous les niveaux, et sur les différentes représentations intermédiaires. En effet, les systèmes embarqués imposent souvent de lourdes contraintes à la fois sur l'espace disponible en mémoire et sur la puissance de calcul utilisable. La structure habituelle d'un compilateur lui fait collecter les informations de débogage au début du processus de compilation, pour les ajouter au fichier binaire à la toute fin. De ce fait, si le programme est modifié par les optimisations, les informations de débogage présentes dans le fichier binaire sont en partie incorrectes. Or le débogueur s'appuie sur ces informations afin de répondre aux requêtes de l'utilisateur. Si elles ne correspondent plus à la réalité du programme, les informations données à l'utilisateur seront erronées. Ainsi la méthode classique de débogage de programme est de développer sans optimisation de compilation durant la phase de mise au point et quand le produit est prêt à être livré, le compiler avec le maximum d'optimisations. Cette méthode ne convient pas au cycle de développement dans le contexte des systèmes embarqués.

Notre approche est de présenter l'exécution du programme optimisé au développeur, de manière à ce qu'il comprenne aisément le lien avec le code source, malgré les transformations appliquées par le compilateur. L'idée est de ne pas émuler l'exécution du programme non-optimisé à partir de l'exécution du programme optimisé. Le développeur de programmes embarqués a des connaissances que nous allons exploiter. À partir d'une analyse de l'état de l'art du débogage de code optimisé et des outils fournis par STMicroelectronics, nous avons cherché à développer une solution viable industriellement. Notre parti est de montrer la réalité à l'utilisateur, de faire ce que P. Zellweger et J. Hennessy ont défini comme étant du *débogage non-transparent*. Il offre au développeur la possibilité de comprendre l'exécution de son programme. Afin de tracer les modifications effectuées par le compilateur, nous proposons d'étiqueter chaque instruction du code source lors de la compilation. Il s'agit ensuite pour le compilateur de maintenir de manière précise les étiquettes utilisées par optimisation et de propager cette information tout au long de la compilation. Ajoutées au fichier binaire en tant qu'informations de débogage, elles sont ensuite utilisées par le débogueur afin de répondre sans erreurs aux interrogations de l'utilisateur. L'ensemble des expérimentations est fait sur le compilateur `mmdspcc` et l'infrastructure `IDbug`.

Mots-clés : Compilation, Débogage, Optimisation de Code, MMDSP+, `IDbug`

Abstract

Optimizations play a major role in the process of compilation for embedded programs. They are to be found at every level, working on different intermediate representations. From the size of run-time memory usage to the available power of execution, embedded systems usually constrain the execution of programs they run. The common structure of a compiler is to collect debugging information at the very beginning of the compilation. It later adds it to the binary file, after having optimized the program, i.e. modified its internal structure. Thus, the debugging information is inaccurate. Since debuggers rely on it, their answers will be wrong. The standard method for development is to work without optimizations until the release. At delivery, optimizations are turned on, rarely before.

Our approach is to reveal the execution of the optimized program to the user so the latter understands the mapping to the source code in spite of transformations applied to the program. We do not emulate the execution of the unoptimized program. We make good use of the programmer's knowledge of its development platform. From the analysis of the state of the art of the debugging of optimized code, we developed an industry-oriented solution. Our goal is to reveal the reality to the user thanks to so-called *non-transparent debugging*. It allows the developer to understand the execution of the program. In order to trace optimizations, we label every single statement from the source code and then enrich this labeling during compilation. Added to the binary file, this information is read by the debugger to enable it to answer the user's queries upon the program states. We present our experiment made on the `mmdspcc` and `IDbug`, tools provided by STMicroelectronics.

Keywords : Compilation, Debug, Code Optimization, MMDSP+, `IDbug`.