Decoupled approaches to register and software controlled memory allocations

Boubacar Diouf

Institut National de Recherche en Informatique et Automatique — **Université Paris-Sud 11**
ED Informatique de Paris-Sud

Decoupled Approaches to Register and Software-Controlled Memory Allocations

THÈSE

présentée et soutenue publiquement le 15 Décembre 2011

pour l'obtention du

Doctorat de l'Université de Paris-Sud

Spécialité : informatique

par

M. Boubacar Diouf

Composition du jury

Président :	Pr. Yannis MANOUSSAKIS
Rapporteurs :	Pr. Pierre BOULET Pr. Jingling XUE
Examinateurs :	Dr. Fabrice RASTELLO Pr. Sebastian HACK Dr. Florent BOUCHEZ
Directeur de thèse :	Pr. Albert Cohen

Université Paris-Sud, 91405 Orsay Cedex, France

Mis en page avec la classe thloria.

Remerciements

L'heure est venue de dire merci à tous ceux qui de près ou de loin m'ont soutenu, aidé ou encouragé durant ces années de thèse ou durant la redaction de ce manuscrit. Je suis conscient qu'il me faudrait sans doute l'équivalent d'un autre manuscrit pour remercier tout le monde monde et n'oublier personne. De ce fait, je tiens à remercier tous ceux dont les noms n'apparaissent pas dans ce document et qui m'ont, ne serait ce qu'une fois, aidé dans ma vie, même si ce fût par un petit sourire.

La première personne que je tiens à remercier est *Albert Cohen*, mon directeur de thèse, pour avoir accepté de diriger cette thèse. Je tiens aussi a le remercier pour toute la confiance qu'il m'a accordé, sa gentillesse et ses conseils.

Mes remerciements vont conjointement et tout particulièrement à M. *Pierre Boulet*, professeur à l'université Lille 1 et M. *Xingling Xue*, professeur à l'université de New South Wales en Australie, qui m'ont fait l'honneur d'être les rapporteurs de cette thèse. Le fait d'avoir des professeurs de leur qualité comme rapporteurs constitue pour moi une motivation et une satisfaction supplémentaire.

Je voudrais également remercier doublement M. *Fabrice Rastello*, chargé de recherche Ã l'ENS de Lyon, pour tous les conseils et critiques objectives qu'il m'a prodigué durant cette thèse, mais aussi pour avoir accepté d'examiner mon rapport de thèse.

Je voudrais également remercier M. *Yannis Manoussakis* , professeur à l'université Paris-Sud, M. *Sebastian Hack*, Professeur à l'université de Saarland et M. *Florent Bouchez*, Docteur-ingénieur à Kalray, pour avoir accepté d'examiner mon rapport de thèse et de faire partie de mon jury de thèse.

J'adresse mes remerciements les plus sincères à Taj et Arame pour les maux de tête que je leur ai causé lors de la relecture et de la correction de mon rapport de thèse.

Je voudrais remercier l'ensemble du personnel de l'INRIA et plus particulièrement Valèrie, Christine et Cédric pour leur apport positif lors des difficultés administratives ou logistiques que j'ai rencontrées.

Un grand merci à tous les membres de l'équipe Alchemy avec qui j'ai partagés des moments inoubliables durant toutes ces années de thèse : *Mounira*, *Taj*, *Mouad*, *Michael*, *Ryiadh*, *Sofiane*, *Zheng*, *Walid*, *Ramakrishna*, *Cédric*, *Luidnel*, *Konrad*, *Grigori Phillipe*, *Louis Noel*, *Olivier* et *Piotr*.

Je tiens aussi à remercier le club des veilleurs avec qui j'ai passé des moments sympatiques sur le parc club et au PCRI : *Lina* la veilleuse de nuit, *Rania* son adjointe et *Vincent* le matinal.

Je remercie toute ma famille, notamment Mon père *Alioune*, ma mère *Fatoumata* et

Ta Ba, mes frères *Mao*, *Big Boss*, *Oussou*,*Moussa* et *Abib*, ma soeur *Fifi*, mon cousin *Seydina* et ma belle soeur *Fatou Diallo*, pour leur irremplaçable et inconditionnel soutien. Il ne faudrait pas non plus que j'oublie mon voisin de colocation Mathieu, pour sa compréhension durant la rédaction et la préparation de la soutenance, et Zorro, pour son aide et son assistance en vue de la préparation de ma soutenance de thèse. Votre présence m'a permis de faire face aux difficultés, aux doutes et aux obstacles que j'ai pu rencontré.

J'aimerais à remercier toutes les personnes qui m'ont assisté durant mes travaux de recherche et d'écriture de cette thèse.

☺

Merci à vous tous!

Je dédie cette thèse
à ma mère Fatoumata Traoré,
à mon père Alioune Diouf,
à ma marraine Ta Ba,
à mes nièces et neuveux, à qui j'ai donné le surnom de : The next generation,
et à mon ami Cheikh Ada, notre voyage ensemble n'a pas duré bien longtemps. Puisse Dieu nous unir de nouveau dans ses hauts jardins

Sommaire

Abstract

Register and local memory allocation are two important optimizations performed during compilation. The former optimization maps the variables of a program to either machine registers or main memory locations. The latter one maps arrays to either local memory or main memory locations. Recent work in register allocation leverages the complexity and performance benefits of decoupling its allocation and assignment phases.

In this thesis, we exploit the decoupled approach to propose a split register allocator,showing that linear complexity does not imply reduced code quality in just-in-time compilation, and to address the spill minimization problem. Considering the similarities between the register and local memory allocation problems, we study how a decoupled approach could be applied to the local memory allocation problem. We propose theoretical basis of such an approach, validate it experimentally and reset a bridge between the register and local memory allocation problems.

Résumé

Les programmes informatiques sont souvent écrits dans des langages de haut niveau et puis traduit en code machine, une forme dans laquelle ils pourront être exécutés par les ordinateurs. Cette traduction est effectuée par un logiciel appelé, *compilateur*. Globalement le rôle d'un compilateur est de traduire un programme source en un programme cible sans en modifier la sémantique; il doit aussi signaler toutes les erreurs qu'il détecte durant le processus de traduction et essayer d'optimiser le code généré. Le programme cible peut être traduit en code machine ou dans un autre langage de programmation. Les optimisations effectuées, par le compilateur, cherchent à minimiser le temps d'execution, et ou, la taille du programme cible.

Le temps d'exécution d'un programme dépend du nombre d'instructions à exécuter mais aussi de la durée que prend chacune de ces instructions à s'exécuter. La durée d'exécution d'une instruction quant à elle peut varier d'une nanoseconde à plusieurs millisecondes selon le temps pris pour accéder aux données stockées dans la mémoire.

La variance dans les temps d'accès à la mémoire est due à une limitation technologique fondamentale: il est possible de concevoir des mémoires de petites capacités à accès rapide, il est aussi possible de concevoir des mémoires de grandes capacités à accès lent, mais il est impossible de concevoir des mémoires de grandes capacités à accès rapide. Il est impossible de fabriquer une mémoire de plusieurs giga-octets accessible en une nanoseconde, ce qui est la performance des processeurs hautes performances actuels. Pour réduire cette différence de performance entre le processeur et la mémoire, les ordinateurs modernes utilisent une *hierarchie mémoire*. Cela correspond à une organisation de la mémoire en plusieurs niveaux. Les mémoires rapides et de petites capacités sont placées à coté du processeurs, ce qui accèlére les accès, alors que celles de grandes capacités et lentes en sont éloignées.

La figure 1 est un exemple de hierarchie mémoire typique utilisée dans les ordinateurs modernes. Les unités mémoires les plus proches du processeurs sont les registres (situés dans le processeur) qui sont les plus rapides d'accès dans un ordinateur. Ensuite, nous avons une mémoire RAM statique (SRAM) de petite capacité, souvent configurée en un ou plusieurs niveaux de mémoire-cache, allant de quelques kilo-octets à plusieurs mega-octets. La mémoire physique ou mémoire principale allant de quelques centaines de mega-octets à plusieurs giga-octets, faite de mémoire RAM dynamique (DRAM) représente le niveau suivant. Enfin, le dernier niveau est formé par la mémoire virtuelle avec une capacité de plusieurs gigaoctets. Lorsqu'un accès mémoire survient, l'ordinateur cherche la donnée dans la mémoire la plus proche du processeur, et si la donnée ne s'y trouve pas, elle

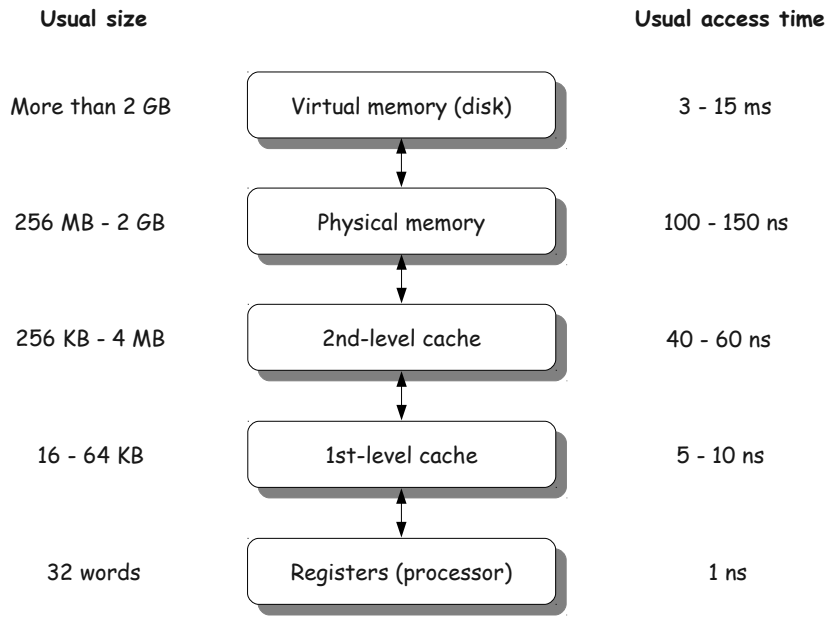

Figure 1: La hierarchie mémoire

cherche dans le niveau supérieur et ainsi de suite.

Les mémoires SRAM sont généralement configurées comme des mémoires caches gérées par le matériel. Une seconde configuration possible est de les gérer de manière logicielle. Dans le cas d'une gestion logicielle les mémoires SRAM, *mémoire(s) locale(s)*, le développeur ou le compilateur insère explicitement des intructions pour transfèrer les données de la mémoire locale vers la mémoire principale. Les registres aussi, comme les mémoires locales, sont gérés de manière logicielle. Tous les autres niveaux de la mémoire sont gérés automatiquement.

Les registres sont les mémoires plus rapides d'accès, mais ils existent en trés petite quantité. Généralement toutes les variables d'un programme ne peuvent pas être stokées dans des registres. Les variables pour lesquelles il n'ya pas de registres disponibles sont stokées dans les couches supérieures de la mémoire qui sont plus éloignées du processeur, donc plus lentes. Pour réduire les accès à ces niveaux supérieurs, il convient donc d'utiliser au mieux les registres de la machine. Dans les compilateurs actuels, ce rôle est dédié à une optimisation appelée: *allocation de registres*. L'allocateur de registres attribue à chacune des variables du programme, soit un registre, soit un emplacement en mémoire. L'allocation de registres se décompose en deux étapes: *l'allocation* qui détermine à chaque

point du programme l'ensemble des variables (variables allouées) qui seront stockées en registres et *l'assignation* qui choisit un registre spécifique pour chaque variable allouée.

De même que pour les registres, Il est important d'optimiser l'utilisation de la mémoire locale pour éviter d'accéder aux niveaux supérieurs de la mémoire, qui sont plus lents. Dans les compilateurs, l'optimisation en charge de la gestion de la mémoire locale est appelée *allocation de mémoire locale*. L'allocateur de mémoire locale selectionne l'ensemble des variables qui seront stockées dans la mémoire locale à chaque point du programme. Il choisit aussi pour chaque variable l'emplacement spécifique qu'il va occuper, soit en mémoire locale, soit en mémoire principale.

L'allocation de registres et l'allocation de mémoire locale sont deux problèmes qui sont NP-complets auxquels nous nous intéréssons dans cette thèse, qui est structurée en trois parties.

Dans la première partie de la thèse, nous nous penchons sur le problème de l'allocation de registres. Tout d'abord, nous proposons dans le contexte des compilateurs-juste-à-temps, une allocation de registres fractionnées, appelée *split register allocation*. Avec cette approche l'allocation de registres est effectuée en deux étapes: une faite durant la phase de compilation statique et l'autre pendant la phase de compilation dynamique. Ce qui permet de réduire le temps d'exécution des programmes avec un impact négligeable sur le temps de compilation. Ensuite Nous introduisons une allocation de registres incrémentale qui permet de résoudre d'une manière quasi-optimale le problème d'allocation. Cette méthode est pseudo-polynomiale alors que le problème d'allocation est NP- complet même à l'intérieur d'un "basic block".

Dans la deuxième partie de la thèse nous nous intéressons au problème de l'allocation de mémoire locale. Au vu des dernières avancées dans le domaine de l'allocation de registres, nous étudions dans quelle mesure le problème d'allocation pourrait être séparé de celui de l'assignation dans le contexte des mémoires locales. Dans un premier temps nous validons expérimentalement que les problèmes d'allocation et d'assignation peuvent être résolus séparément. Ensuite, nous procédons à une étude plus théorique d'une approche découplée de l'allocation de mémoire locale. Cela permet d'introduire de nouveaux résultats sur le "submarine-building problem", une variante du "ship-building problem", que nous avons défini. L'un de ces résultats met en évidence pour la première fois une différence de complexité (P vs. NP-complet) entre les graphes d'intervalles et les graphes d'intervalles unitaires.

Dans la troisième partie de la thèse nous proposons une nouvelle heuristique, appelée "clustering allocator" fondée sur la construction de sous-graphes stables d'un graphe d'interférence, permettant de découpler aussi bien le problème d'allocation pour les reg-

istres que pour les mémoires locales. Cette nouvelle heuristique se veut le pont qui permettra de réconcilier les problèmes d'allocations de registres et de mémoire locale.

1 Allocation de Registres

L'allocation de registres est un problème NP-complet. En effet, Chaitin et al. ont montré que le problème de l'existence d'une allocation sans spills (*sill-free*) est équivalent au problème de la coloration de graphe [CAC+81]. En plus, le problème de l'allocation de registres ne se limite pas uniquement au problème du "spill-free"; si la réponse au problème du "spill-free" est non, alors le but de l'allocation de registres est aussi de réduire l'impact des variables spillées sur le temps d'exécution du programme. Ce problème est connu sous le nom de, *minimisation des coûts de spills* ou *minimisation de spills*. Plusieurs heuristiques et techniques d'approximation au problème d'allocation de registres ont été proposé dans la litterature [CAC+81, BCT94, PS99, THS98, HGG06, PP08].

Dans cette première partie, nous proposons deux nouvelles techniques à savoir l'allocation de registres fractionnées, qui cherche à améliorer l'allocation de registres dans le contexte de la compilation *juste-à-temps* (JIT) et *l'allocation de registres optimale itérée* qui s'attaque au problème de la spill minimization.

1.1 Allocation de Registres Optimale Itérée

Les travaux en allocation de registres ont montré que lorsque les live ranges des variables sont assez finement découpés alors le problème de l'assignation devient quadratique dés lors qu'une allocation est possible. Il suffit juste que le nombre maximal de variable en vie en un point du programme soit inférieur au nombre de registres disponibles. Cependant le problème de la minimisation de spills demeure NP-complet même à l'intérieur d'un basic block [FCL00]. Les solutions proposées pour la résolution sont loin d'être optimales. Ce problème est d'autant plus important sur les machines CISC, où le nombre de registres est trés limité.

Nous nous sommes attaqués au problème de la minimisation de spills. Notre approche, appelée *allocation de registres optimale itérée*, permet de résoudre le problème d'allocation de manière quasi-optimale et ceci rapidement. Notre solution peut être utilisée dans un contexte découplé ou non. Nous avons comparé notre approche avec la coloration de graphe, le "linear scan", une nouvelle heuristique que nous avons conçu et appelé *heuristique mixte* et une allocation de registres optimale par programmation linéaire. Les résultats montrent que L'allocation de registres optimale itérée est souvent proche de

l'allocation de registres optimale et produit de bien meilleurs résultats que les autres heuristiques. Un résultat trés intéréssant de notre approche est que, pour les programmes sous SSA, nous avons une garantie de pseudo-polynomialité.

1.2 Allocation de Registres Fractionnée

Dans le contexte de la compilation JIT, la compilation fait partie du processus global d'exécution des programmes. Ceci implique un compromis entre le temps de compilation, donc des optimisations et celui du temps d'exécution du code généré. En pratique, l'utilisation d'algorithmes de compléxité linéaire reste la régle pour la compilation JIT.

Notre approche de l'allocation de registres fractionnée montre qu'une compléxité linéaire n'implique pas forcément une réduction de la qualité du code généré. Notre approche est un exemple de compilation fractionnée, ou les analyses couteûses sont effectuées en avance pour guider des optimisations quasi-linéaires. Les informations collectées dans la phase hors-ligne de compilation statique, sont transmises à la phase en ligne, de compilation juste-à-temps, au moyen d'annotations de bytecode.

Notre allocateur de registres fractionnée garantie quatre propriétés: des annotations avec un impact minimal sur la taille du code, un traitement des annotations en temps linéaire, une perte minimale en qualité de code et la portabilité des annotations lorsque le nombre de registres disponibles varient. Nous avons implanté notre technique d'allocation de registres fractionnée dans JIkesRVM, la machine virtuelle de recherche d'IBM.

2 Allocation de Mémoire Locale

Dans la majorité des systèmes embarqués, les mémoires locales sont préférées au caches, du fait des garanties qu'elles offrent en prédictibilité, en adaptabilité (passage à l'échelle), en efficience énergétique, en rapidité d'accès aux données et du faible espace occupé sur la puce [BSL+02]. La predictibilité et une bonne consommation d'energie sont souvent essentielles aux applications temps réél et embarqués. D'autres processeurs plus spécialisés, tels que les "stream-processors et les processeurs graphiques, utilisent aussi les mémoires locales [NVI08, BPMR03].

Pour bien exploiter tout le potentiel des mémoires locales, il est essentiel de les utiliser de manière efficiente. Dans les compilateurs, cette tâche est à la charge d'une optimisation appelée allocation de mémoire locale. L'allocateur de mémoire locale selectionne l'ensemble des variables qui seront stockées dans la mémoire locale à chaque point du programme. Il choisit aussi pour chaque variable l'emplacement spécifique qu'il va occuper,

soit en mémoire locale, soit en mémoire principale.

L'allocation de mémoire locale est un problème NP-complet [VWM04]. Les travaux précédants ont porté sur différents angles, ciblant aussi bien une allocation du code et des données du programme. Plusieurs heuristiques ont été proposées à la résolution du problème d'allocation de mémoire locale. [MFA01, VWM04, UDB06, LFX09, LXK11].

L'allocation de mémoire locale est connectée à l'allocation de registres depuis au moins 30 ans. En effet, dans son papier phare [Fab79], Fabri a démontré l'existence de liens forts entre les allocations de registres et de mémoire locale. Ces liens ont été sous-exploités depuis lors et par conséquant, les nouvelles avancées, réformant le design des "backend" des compilateurs, dans le domaine de l'allocation de registres ont été complétement ignorées en allocation de mémoire locale. Notre travail sur l'allocation de registres a été motivé par les récéntes avancées en allocation de registres et par la volonté de mieux comprendre le probléme d'optimisation.

2.1 Validation expérimentale

Pour valider notre intuition d'une approche découplée de l'allocation de mémoire locale, similairement à l'approche découplée en allocation de registres, nous avons exprimé l'allocation des tableaux en programmation linéaire pour résoudre le problème optimalement (en minimisant le coût des latences d'accès) en utilisant MaxSize comme critère. MaxSize est la taille maximale occupé par les tableaux en vie au même moment. Une fois le problème d'allocation résolu, la phase d'assignation se charge d'assigner aux tableaux alloués des emplacements dans la mémoire locale. Cependant, due à la possible fragmentation que la phase d'assignation peut entrainer, une solution du probléme d'allocation ne garantie pas forcément une solution au problème d'assignation. Néanmoins, nous avons expérimentalement montré, sur une série de benchmarks, que dés lors que le problème d'allocation admettait une solution alors le problème d'assignation aussi en admettait une sans spills supplémentaires.

2.2 Etude théorique

Aprés avoir expérimentalement validé une approche découplée de l'allocation de registres, nous avons pris une approche plus théorique qui a aboutit à des résultats fondamentaux dans le domaine de la gestion des mémoires locales par les compilateurs. Nous avons montré que la gestion des mémoires locales peut se modéliser comme un problème de coloration de graphes pondérés et nous avons étudié plusieurs variantes de ce problème. Ce qui a mené aux contributions suivantes:

1. L'introduction d'une nouvelle forme de coloration par intervalle que nous avons appelé "submarine-building".

2. Un nouvel algorithme linéaire et optimal résolvant le problème du "submarine-building" pour tout graphe d'intervalle propre et une preuve de NP-complétude de ce problème pour les graphes d'intervalle.

3. L'introduction d'un critère permettant de décider de la faisabilité ou non d'un problème de "submarine-building" sur les graphes d'intervalle propre. Lorsque le critère est satisfait, l'algorithme linéaire et optimal de résolution des problèmes "submarine-building" permet de résoudre le ship-building problème pour les graphes d'intervalle propre.

4. Nous exhibons pour la première fois un problème NP-complet pour les graphes d'intervalle et polynomial pour les graphes d'intervalle propre (qui sont équivalents aux graphes d'intervalle unitaire).

Les évaluations effectuées ont montré que l'approximation des graphes d'intervalle par des graphes d'intervalle propre ne fournit pas de très bons résultats. Ces résultats décevants nous ont poussé à la conception d'une nouvelle méthode heuristique de résolution du problème de la gestion des mémoire locale.

3 Allocation par Clustering

Notre étude théorique d'une approche découplée de l'allocation de mémoire locale a aboutit à des résultats encourageants. Cependant cette approche n'est pas encore assez mature pour qu'on puisse l'utiliser dans la pratique. De ce fait nous avons considéré une méthode heuristique que nous avons appelée, *clustering allocator*. Cette approche, bien qu'elle ait été conçue pour l'allocation de mémoire locale, fournit de trés bons résultats en allocation de registres. Donc après plusieurs année de séparation, le "clustering allocator" semble rétablir un pont entre les problèmes d'allocations de mémoire locale et de registres.

Le "clustering allocator" decouple les problème d'allocation et d'assignation et cherche à minimiser le coût de l'allocation par la construction de sous-graphes stables d'un graphe d'interférence. Durant la phase d'allocation, notre algorithme regroupe les variables par "clusters" qui seront alloués, ou spillés globalement.

Pour les problèmes d'allocation de registres, le "clustering allocator fournit des résultats presques optimaux et est beaucoup plus performant que les autres heuristiques

Pour les problèmes d'allocation de mèmoire locale, sur des graphes générés de manière aléatoire, nous obtenons de meilleurs résultats que l'algorithme du best-fit sur des graphes de densité moyenne à forte.

Introduction

Programs running on computers are often written in high-level programming languages and then translated into an executable code, a form in which they can be executed by these computers. This translation is done by a software program, called a *compiler*. Globally, the role of a compiler is to translate a source program into a target program without changing the semantics of the source code; it must also report any errors that it detects during the translation and try to optimize the generated target code. The target program can be an executable code or a program written in another programming language. The optimizations, done by a compiler, try to minimize the execution time, and or, the size of the target program.

When a programmer compiles a program from a source language to an executable code, he wants the compiled one to conserve the source code's semantics. He also wants the code to run in an acceptable time. The execution time depends on the number of instructions to compute, and also on the computation time of each of these instructions. The computation time of an instruction can vary significantly, from nanoseconds to milliseconds, depending on the time taken to access data in memory.

The variance in memory access times is due to technological limitations: we can make small storage with fast access-time, we also can make large storage with slow access-time, but we cannot make large storage with fast access-time. It is impossible to make gigabytes of memory to be accessed in a nanosecond, the speed of high-performance processors. To reduce this performance gap between the processor and the memory, modern computers use a *memory hierarchy*. It corresponds to an organization of the memory into different levels of hierarchy. The faster and smaller levels of memory are placed closer to the processor, this speeds up access, and the slower and larger ones are placed farther from the processor.

Figure 1, taken from the "Dragon Book" [ALSU06], shows an example of a typical memory hierarchy in a modern computer. The closest memory level to the processor is composed of a few registers (located in the processor) that are accessed faster than any other kind of memory. Then, we have a small amount of on-chip *static RAM* (SRAM),

usually configured as one or many levels of hardware-managed cache, going from some kilobytes to many megabytes in size. The next level is the physical (main) memory, made of *dynamic RAM* (DRAM), going from hundreds of megabytes to many gigabytes. Finally, the last level in the hierarchy is the virtual memory with a size of many gigabytes. When a memory access occurs, the computer tries to find the data in the closest memory, and then if the data is not there, it tries in the higher level, and so on.

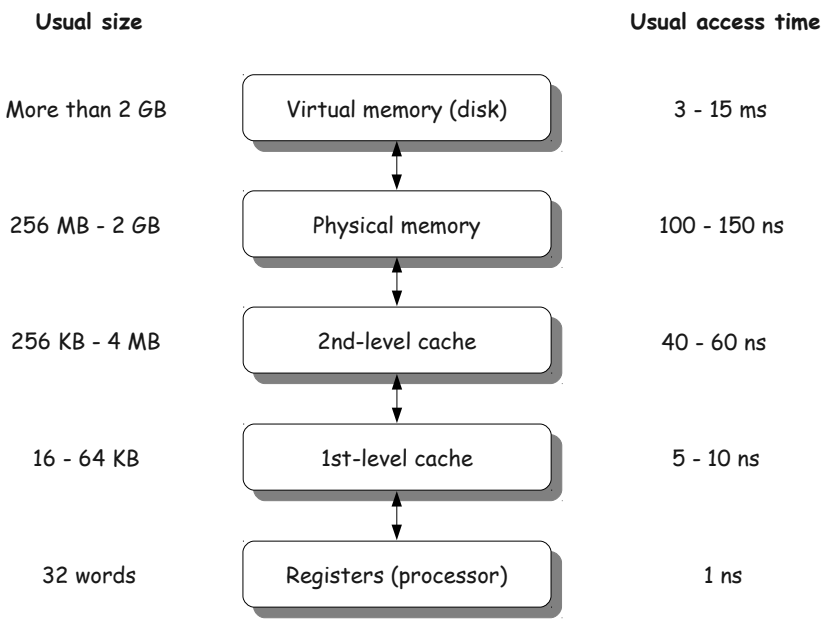

Figure 1: A typical memory hierarchy configuration, from the Dragon book [ALSU06]

The alternative approach to manage the SRAM is to configure it as a *software-controlled local memory*. In such an approach, the developer or the compiler must insert explicit instructions to transfer data between the local memory and the main memory. Like software-controlled local memories, registers also are managed by software. All the other levels of memory are managed automatically.

1 Register Allocation

Registers are the fastest type of memory within a computer. Registers are quickly accessed, but they exist in a very limited number. Usually, all the values of the executed code cannot reside in registers. The values not held in registers should reside in memory,

which is farther from the processor and therefore slower to access. To reduce access to slower memory, it is essential to have an efficient usage of registers.

In all modern compilers, there is a phase called *register allocation* which optimizes the use of registers. The register allocator, in a phase called allocation, decides at each program point which variables will be held in registers (allocated variables) and which variables will be stored in memory; it also assigns each allocated variable to a register and maps to other variables a location in memory.

The register allocation is NP-complete. Indeed, Chaitin et al. have shown that the *spill-free* problem is equivalent to the graph coloring problem [CAC+81]. Moreover, register allocation is not bound to the spill-free problem; if the answer to the spill-free problem is no, the goal of register allocation is also to reduce the impact of the spilled variables on the execution time of the program (*spill cost minimization*). Many heuristics and approximation techniques have been proposed to solve register allocation [CAC+81, BCT94, PS99, THS98, HGG06, PP08].

In this thesis we are interested in two topics: the spill minimization problem and the improvement of the register allocation in the context of Just-in-time (JIT) compilation.

1.1 Spill Minimization

Recent works in register allocation have shown that when enough live range splitting is allowed, the assignment problem can be solved in quadratic time as soon as the maximum number of simultaneously living variables is lower than the number of available registers. But unfortunately, the spill minimization problem (an allocation that minimizes the access latency) is still NP-complete even when enough live range splitting is allowed [FCL00]. The proposed solutions are far from being optimal. Thus, our aim is to propose good heuristics to solve this problem especially on systems like CISC machines where only few registers are available.

1.2 Improvements of JIT Register Allocation

In the context of *Just-in-time* compilation, the compilation time is part of the global execution process. This implies a trade-off between the time spent for compilation (optimizations) and the execution time of the produced code. In practice, (quasi-)linear complexity is the rule for JIT compilation. This severely impacts the aggressiveness of optimizations, like register allocation. We are interested in solutions that improve register allocation without lengthening the JIT compilation.

2 Local Memory Allocation

In most embedded systems, local memories are often preferred to caches due to their better performance and predictability, their power efficiency, and their smaller area cost [BSL+02]. Predictability of data access and power consumption efficiency are often essential to real-time and embedded applications. More specialized processors also utilize local memories, including stream-processing architectures such as graphical processors (GPUs) and network processors [NVI08, BPMR03].

To take advantage of all the potentials provided by local memories, it is essential to use them efficiently. Within a compiler, the optimization which performs this task is called *local memory allocation*. It selects the set of variables that will reside in the local memory at each point of the program. It also finds the specified place in the local memory or in the main memory where a variable will reside in.

The local memory allocation problem is NP-complete [VWM04]. Previous studies addressed local memory allocation from different angles, targeting both application/code and data. Many heuristics-based approaches to the problem have been proposed [MFA01, VWM04, UDB06, LFX09, LXK11].

Our work on local memory allocation is motivated by the recent advances in register allocation and the desire for a better understanding of the local memory optimization problem.

2.1 Link between Register and Local Memory Allocations

Compilation-time local memory allocation has been connected to register allocation for at least 30 years. Indeed, in her seminal paper [Fab79], Fabri reported strong links between register allocation and local memory allocation. This have been overlooked until very recently. As a result, the series of, fundamental and applied advances impacting the design of compiler backends have also been ignored in the field of local memory allocation [AG01, BDGR06a, HGG06, BDR07c, PP08].

2.2 The Local Memory Allocation Optimization Problem

While there exist many heuristics to the local memory allocation problem, theoretical foundations of the proposed heuristics are missing. More recently, Li et al. [LXK11] seriously improved the state of the art by proposing an approach to the problem, that has strong theoretical foundations. Nonetheless, little is known about the optimization problem, its complexity, and its interplay with other optimizations.

3 Outline

This dissertation is divided into three parts. The first part is devoted to register allocation, the second one tries to see in which context the fundamental advances in register allocation could be extended to local memory allocation, and the third one aims to rebuild a bridge between the domains of register and local memory allocations.

3.1 Register Allocation

In Chapter 1, we present the state of the art in register allocation. We introduce the terminology and the essential notions needed to understand register allocation. We describe the graph coloring and the linear scan approaches to register allocation and we show how separating the allocation phase from the assignment helps to ease and improve the register allocation.

In Chapter 2, we recall the approaches used to process programming languages: static compilation, interpretation, and JIT-compilation. We also introduce *split compilation* and show how it improves the quality of the code generated by JIT compilers.

In Chapter 3, we introduce the *split register allocation* which leverages the decoupled approach to improve register allocation in the context of JIT compilation. We experimentally validate the effectiveness of split register allocation and its portability with respect to register count variations, relying on annotations whose impact on the bytecode size is negligible.

Chapter 4 introduces a new decoupled approach, called *iterated-optimal allocation*, to register allocation. The iterated-optimal allocation algorithm achieves results close to optimal while offering pseudo-polynomial guarantees for SSA programs and fast allocations on general programs.

3.2 Local Memory Allocation

Chapter 5 sets the state of the art in local memory allocation. We introduce in this chapter static and dynamic methods for local memory allocation.

In Chapter 6, we explain the motivation of our work on local memory allocation. We also expose our approach to the problem, the preliminary assumptions we have made and our methodology to evaluate and compare our work.

Chapter 7 validates our intuition for decoupled approach to local memory allocation. It shows experimentally that after an optimal allocation phase relying on a generic and scalable integer linear program, the assignment phase could be achieved without any

fragmentation-induced spills.

In Chapter 8, reinforced by results of Chapter 7, we study the local memory allocation in a more theoretical way setting the junction between local memory allocation for linearized programs and weighted interval graph coloring. We design and analyze a new variant of the *ship-building* problem called the *submarine-building* problem. We show that this problem is NP-complete on interval graphs, while it is solvable in linear time for proper interval graphs. We also give a criterion to guarantee the feasibility of the submarine-building problems for proper interval graphs and then we extend it to an extension of the class of proper interval graphs interval graphs. Our results show that while our approach represents an improvement over state of the art methods, it is limited so far in its practical application.

3.3 Reconciling Register and Local Memory Allocations

In Chapter 9, we propose a heuristic-based solution, the *clustering allocator*, which decouples the local memory allocation problem and aims to minimize the allocation cost. The clustering allocator while devised for local memory allocation, appears to be a very good solution to the register allocation problem. For register allocation, the results show that the clustering allocator outperforms both graph coloring and linear scan and is often close to the optimal solution. For local memory allocation the results are not as good as those for register allocation, but we do believe that the clustering allocator can be improved.

Part I

Register Allocation

Chapter 1

Register Allocation

1.1 Introduction

When a programmer compiles a program from a source language to an executable code, he wants the compiled one to conserve the source code's semantics. He also wants the code to run in an acceptable time. The execution time depends on the number of instructions to compute, and also on the computation time of each of these instructions. The computation time of an instruction can vary significantly, from nanoseconds to milliseconds, depending on the time taken to access data in memory.

In a modern computer, it does not take more than one cycle of the CPU clock to read or write data into a register, while reading the data, from the cache or the main memory, is an order of magnitude slower. Registers are quickly accessed, but they exist in a very limited number. A 32-bit x86 architecture has only 8 general-purpose registers, ARM and PowerPC processors typically have 32 registers. Usually, all the values of the executed code cannot reside in registers. The values not held in registers should reside in memory. To reduce accesses to higher levels of memory it is essential to have an efficient usage of registers. Registers are software managed whereas all the other levels of the hierarchy are managed automatically. The management of the registers is done by the programmer, the application, or the compiler.

In all modern compilers, there is a phase called *register allocation* which optimizes the use of registers. The *register allocation* is usually one of the last phases of the compilation. It happens when all the candidate variables to register allocation are known. These candidate variables are composed of the variables already present in the source code and of the temporaries generated during preceding phases of compilation. The register allocation is defined [ALSU06] as the problem which is often subdivided into two sub-problems:

1. the *allocation* which selects the set of variables that will reside in registers at each point of the program.

2. the *assignment* which picks the specified register where a variable will reside in.

The variables mapped to registers are called *allocated variables* and the rest, stored in memory, are called *spilled variables*.

```
a := 2
b :=10
a := a * b
c := a - b
   ...
```

(a) Before register allocation

```
r1 := 2
r2 := 10
r1 := r1 * r2
r3 := r1 - r2
   ...
```

(b) After register allocation

Figure 1.1: An example of register allocation

Figure 1.1 (b) gives an example of register allocation for the code shown in Figure 1.1 (a). Assuming that three registers, $r1$, $r2$ and $r3$ are available, the variables a, b and c are assigned respectively to registers $r1$, $r2$, and $r3$.

In this chapter, we intend to present the state of the art in register allocation. Section 1.2 introduces the terms and notions necessary to understand register allocation. Section 1.3 presents some aspects of the register allocation problem, with a sub-section on its complexity. Then, Sections 1.4 and 1.5 present respectively *graph coloring* and *linear scan* as two approaches to register allocation; the former is the dominant approach, and the latter is an alternative mostly implemented in *just-in-time compilers*. Finally, Section 1.6 presents a new approach to register allocation which decouples its allocation and assignment phases.

1.2 Terminology

We set here the meaning of some terms and notions needed or used in register allocation.

Basic Block. A basic block is a maximal sequence of instructions with the following properties:

- it has a unique entry point as its first instruction. Any instruction within the basic block but this entry point cannot be a destination of a jump instruction.

- it has a unique exit point as its last instruction, meaning that no instruction but the last one can cause to leave the block.

Within a basic block, when the first instruction is executed all the other instructions of the basic block are executed.

Figure 1.2(b) shows an example of four basic blocks: BB_1, BB_2, BB_3 and BB_4.

Control-flow graph. A control-flow graph is an intermediate representation describing all the possible sequencing of instructions that may occur during the execution of the program. It is a directed graph where each node is a basic block. There is an edge from the basic block B to the basic block C, if the first instruction of C can follow immediately the last instruction of B. B is called a predecessor of C and C is called a successor of B. A control-flow graph must have a unique entry point and a unique exit point. That is why often an *entry block* and an *exit block* are added to the control-flow graph.

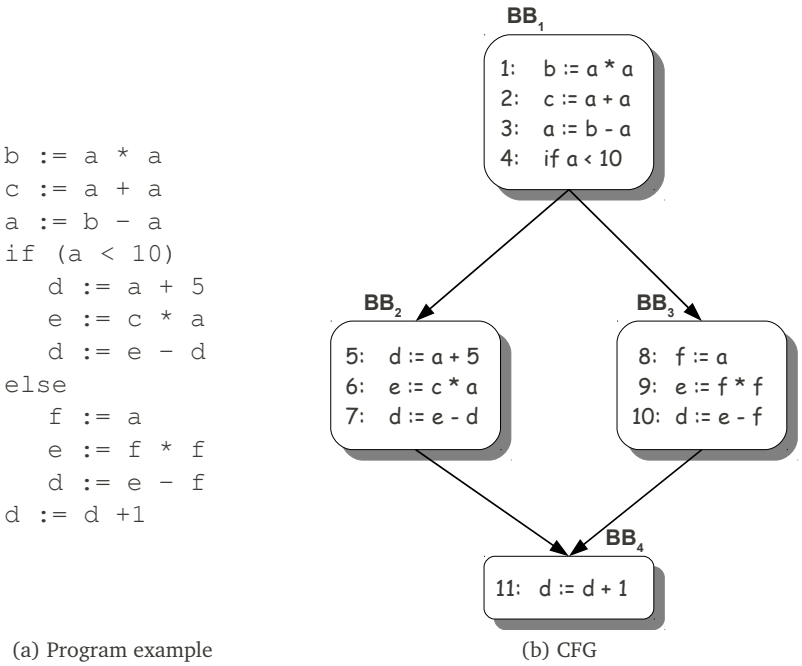

Figure 1.2: The control-flow graph of an example program.

Figure 1.2(b) shows the control-flow graph of the portion of code given in Figure 1.2(a).

Dominators. A node d of a control-flow graph *dominates* a node n if every path of directed edges from s_0, the entry point, to n must go through d [AP02]. Every node dominates itself. An *immediate dominator* $idom(n)$ of n is a basic block that dominates

n and that is also dominated by any other basic block d that dominates n. A basic block cannot be its own immediate dominator. The graph containing every node of the control-flow graph, and for every node n and edge from $idom(n)$ to n is called the *dominator tree*. It is a tree because each node has exactly one immediate dominator.

Liveness. We say that a variable v is *defined* by an instruction if it is a result of that instruction. We also say that it is *used* by an instruction if it is read by that instruction. Notice that a variable can be defined and used by the same instruction. For instance, the variable b in Figure 1.2(b) is defined by the first instruction. The variable d is re-defined and used by the instruction number 7. A variable v is said to be *live* at the point p if there exists a path in the control-flow graph from p to the exit point where v is used without being re-defined.

Live range. The *live range* of a variable v is the list of program points where it is live. Figure 1.3 shows the live ranges of the variables of the example program given in Figure 1.2(a). The variable a is supposed to be live in, which means it is live at the beginning of the basic block, it stays live until it is read, by the instruction 6 if the left branch is taken, and the instruction 8 if the right branch is taken. By abuse of terminology in this dissertation, as it is common in the literature, we sometimes use the term live range to designate the variable it refers to.

Interference. Two variables *interfere* if the intersection of their live ranges is non-empty. Two interfering variables cannot share the same register because this will cause one of them to overwrite the content of the other. The variables a and b interfere because their live ranges represented in Figure 1.3 intersect, whereas the variables d and f do not.

Spilling. An essential notion to register allocation is *spilling*. It is frequent that the number of available registers in an architecture cannot map all the variables of a program. Therefore, it is sometimes necessary to have some variables that remain in the memory. We say that these variables are spilled. In certain architectures, e.g., RISC architecture, memory can only be accessed through load and store instructions. This means that to use a variable v at address a, we need to load the value at address a (which is the value of v) into a register and then use this register as an operand. Thus, even if a variable is spilled, it must be loaded into a register from memory before each of its use, and stored from a register after each of its definition, and the added instructions are called *spill code*. In architectures like x86, an instruction's operand can be directly accessed from

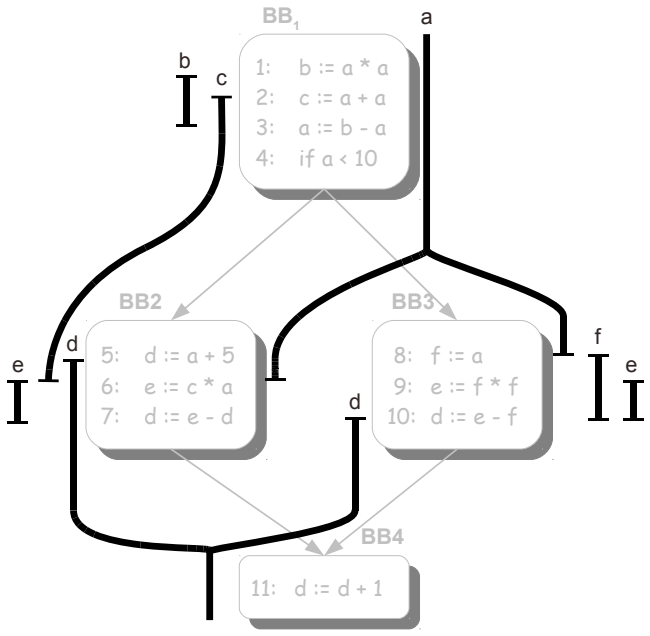

Figure 1.3: The variables's live ranges in Figure 1.2(a).

memory. It means that to manipulate a variable v at address a in the memory, we can use a as an operand to directly use or define v. Such a kind of operand is called a *memory operand*. But even for these architectures, some restrictions may exist, for instance on IA-32 architectures, move instructions cannot have more than one memory operand.

Coalescing. Two variables related by a copy instruction and which do not interfere can be assigned to the same register. That is, the copy instruction becomes useless and can be safely removed. We say that these two variables are *coalesced*. A good register allocator will also strive as most as possible to coalesce variables that can be coalesced, because it will improve the quality of the generated code. For illustration, the variables a and f in Figure 1.3 should be coalesced, by looking at instruction 8.

Live range splitting. A variable v live at two different instructions $i1$ and $i2$ can be split into two separate variables renamed v_1 live at instruction i_1 and v_2 live i_2. To avoid changing the semantics of the program, sometimes it is necessary to join instructions v_1 and v_2, with a copy instruction from v_1 to v_2. This process of splitting the variable v is called *live range splitting*. The live range of a variable v can be very long and can have a non-empty intersection with many other variables's live ranges. Splitting the live range of such kind of variables tends to reduce the interference between variables and thus

minimizes the number of registers required by a program.

Static Single-Assignment (SSA) Form. The SSA form is an intermediate representation in which each variable has only one definition in the program text [AP02, CFR+91, RWZ88]. The SSA form facilitates many other optimizations like constant propagation, dead code elimination, and register allocation, as we will see later. In straight line code (e.g. in a basic block), which is a code without control-flow, each definition of a variable v is given a new name such as $(v_1, v_2, \ldots)$, and each use of a variable v is renamed to the *most recently defined version* of the variable. In programs with control-flows, With two control-flow paths merging together, a special form of variable's definition called *ϕ-function*, is inserted.

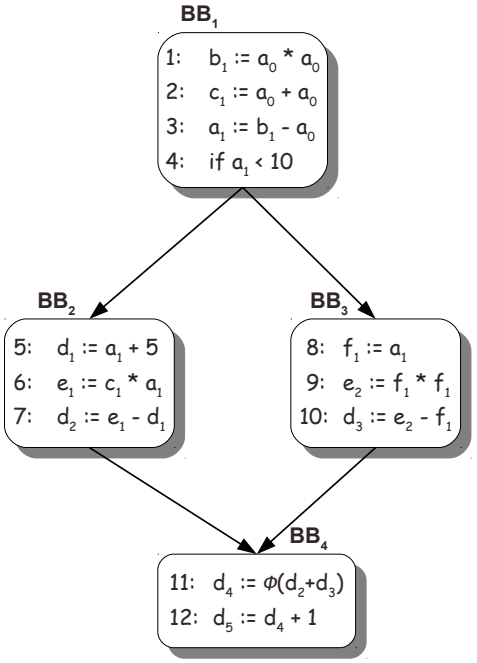

Figure 1.4: The program in 1.2(a) in SSA form.

In Figure 1.4, a ϕ-function is inserted at the beginning of the basic block BB_4. The variable d_2 and d_3 used as the operands of the ϕ-function indicate which definitions of d reach the join node. Subsequent uses of the variable d are replaced with uses of the variable d_4. The ϕ-functions do not correspond to hardware instructions and after the optimizations have been done upon it, a program in SSA form must be translated into an executable representation without ϕ-functions.

1.3 Aspects of the Optimization Problem

The scope of register allocation can be local, global, or interprocedural. It is *local* when it is restricted to a basic block, *global* when it is performed over a procedure or a method, and *interprocedural* when it is performed across multiple procedures. Register allocation has been performed since the first compiler for FORTRAN. It has been widely studied and shown to be NP-complete. It has also been shown to have strong interactions with the instruction scheduling optimization.

Depending on the trade-off between the difficulty of the register allocation problem and the quality of the solution of the problem (memory-access-latency minimization). We can identify at least three types of problems that can be considered:

Spill everywhere. It corresponds to the coarser grain of the register allocation problem, where the live range of each variable is viewed as an atom [PS99], there is no live range splitting. In this context, a variable cannot be spilled at some part of the code and allocated at some other part: it is spilled everywhere. This approach does not give best results but it is simple and when compile time is in concern like in JIT compilers, it can be well suited.

Register allocation with live range splitting. This approach is finer than the first one. The live range of a variable can be split at some points, for instance, at points where variables are redefined or spilled (to avoid loosing previous assignment). In this context, a variable can be assigned to a register r_1 sometimes and spilled or allocated to another register r_2 at a different moment. This second approach is harder but superior to the first one and SSA-based techniques and recent work on register allocation focus on it[THS98, HGG06, BDR07a].

Load-store optimization. The third approach of the finest granularity. The goal is to optimize each load and store separately in order to minimize the memory access time. In this context, the variables's live ranges can be split between every two instructions that can be consecutive [AG01]. This approach gives much better results than the two preceding level of granularity, but it is also harder to solve.

1.3.1 Complexity

Register allocation is a NP-hard problem. Indeed, Chaitin et al. have shown that the *spill-free* problem is equivalent to the graph coloring problem [CAC+81]. Moreover, register allocation is not only bounded to the spill-free problem; if the answer to the spill-free

problem is no, the goal of register allocation is also to reduce the impact of the spilled variables on the execution time of the program (*spill minimization*). In addition to this, register allocation usually removes useless copy instructions in order to minimize the execution time of the program (coalescing problem). Register allocation also needs to handle some irregularities in the underlying architecture, like register *aliasing* and variable's *pre-coloring*.

Spill-free problem. Given k available registers and n variables that are candidates to register allocation, the spill-free problem answers the question of whether or not it is possible to assign the n variables to registers without any spilling. For general programs, this problem is equivalent to the graph coloring problem and is hence NP-complete [CAC+81].

Spill-cost (or spill) minimization problem. When all or part of a variable v in a program is spilled, it may impact on the execution time of the program because of the spill code insertion. If we assume that a store has a cost and a load has a cost too, then it is possible to compute the cost of the spilled variables. The spill minimization is the problem of minimizing the cost of the spilled variables for a register allocation problem. This problem becomes NP-complete even for local register allocation as soon as stores have positive cost [FCL00, LFCK99].

Coalescing problem. The coalescing reduces the cost of move instructions between variables that do not interfere. The problem of looking for an allocation of minimal cost (where each pair of coalescable variables has been assigned a cost) is NP-complete [BDR07b].

Aliasing problem. When an assignment to a register name can affect the content of another register name, such register names are said to alias. As shown in Figure 1.5, in the IA-32 architecture, the two lower bytes of the 32-bit general-purpose registers EAX and EBX can be referenced with names AX and BX. The first byte of AX and BX can be respectively referenced with AH and BL (high bytes) and their last bytes can be referenced with AL and BL (low bytes). In this example the register names EAX and AX, EAX and AH, or AX and AL alias. Lee et al. [LPP08] show that aliased register allocation is NP-complete even for straight-line programs, which are very simple programs without control-flows.

Pre-coloring problem. We say that a variable is pre-colored if it must be assigned to a register due to some architectural constraints. For instance, in many architectures,

Figure 1.5: An example of aliasing registers's names

registers are used to pass parameters during function and procedure calls. Such kind of conventions force the first k arguments of the called function to be assigned to the k registers reserved for that purpose. Register allocation with pre-colored variables is equivalent to the coloring extension problem. Biro et al. [BHT92] demonstrate that the coloring extension problem is NP-complete for interval graphs, and thus, for register allocation with pre-colored variables.

1.3.2 Register Allocation with Fixed Scheduling

Register allocation is not the only optimization performed by the compiler. It is part of a flow of optimizations which may impact on each other. In particular, it has strong interactions with *instruction scheduling* optimization. The instruction scheduling is used to increase the *Instruction Level-Parallelism* (the number of simultaneously executed instructions during a computer cycle). It looks for an ordering capable of improving the instruction level-parallelism.

Instruction scheduling can be done before or after register allocation. If it is done before register allocation, it can increase the number of needed registers and make the code harder to color. If it is done after register allocation, some ordering opportunities may then turn impossible due to the performed register allocation. The interplay between these two problems have been studied and many approaches that consider both of these two problems have been proposed in the literature [TE04b, BEH91, NP95]. Register allocation with instruction scheduling considerations is beyond the scope of this dissertation: we assume that we have a fixed scheduling.

1.4 Graph Coloring

The graph coloring is the dominant approach to register allocation. The idea of abstracting the register allocation problem to a graph coloring problem dates from the early 1960s [SL62]. The first graph coloring framework for register allocation has been implemented

by Chaitin et al. [Cha82, CAC+81]. We need first to introduce some terms about graphs before presenting the graph coloring approach to register allocation.

A *graph* $G = (V, E)$ consists of two sets, V the set of vertices or nodes, and E the set of edges. Every edge (v_1, v_2) of E has two end points $v_1 \in V$ and $v_2 \in V$. We say that v_1 and v_2 are *adjacent(s)* or are *neighbor(s)* if $(v_1, v_2) \in E$. The number of neighbors of a vertex v is called the *degree* of v. Here, We only consider *undirected* graphs, i.e., we do not make difference between the edges (v_1, v_2) and (v_2, v_1).

A *clique* is a set of vertices, where every pair of distinct vertices is adjacent. A clique is maximum if there is no clique of G of larger cardinality. The number of vertices in a maximum clique of G is denoted $\omega(G)$ and is called the clique number of G.

A *vertex coloring* (or coloring) of a graph G is a function C that maps each vertex v of G into a color c_v such that adjacent vertices are mapped to different colors. We say that I is a k-coloring of G, if the number of colors used to color G is equal to k. The chromatic number $\chi(G)$ is the smallest k for which it is possible to find a k-coloring of G. A proper coloring of a maximum clique A of G requires at least $\omega(G)$ colors, because otherwise there will be at least two vertices of A having the same color. Hence, for any graph G we always have: $\omega(G) \leq \chi(G)$.

The technique presented here abstracts the register allocation to an *interference graph* coloring problem. The interference graph G of a given program is the intersection graph of the variables's live ranges in that program. Each variable is represented by a vertex in G and the vertices of two interfering variables are adjacent in G. If we consider each register as a color, thus for a given program, the register allocation problem corresponds to a graph coloring problem.

Chaitin et al. use the Kempe's heuristic to color the interference graph. Given a graph G and k available colors, the heuristic tries to find a k-coloring of G. Assume that there exists a node n in G of *low degree*, that is, a node which has less than k neighbors. From a k-coloring of $G - \{n\}$, it is easy to find a k-coloring of G, it suffices to assign to n a color that differs from the colors of its neighbors. Thus, n is always colorable and G is k-colorable if $G - \{n\}$ is k-colorable. The node n can be removed (with its edges) from G and placed on a stack. When n is removed from G, the dregrees of its neighbors are lowered and this may turn some of its neighbors into low degree nodes in the new graph and prove that they are colorable. If this procedure can be iterated until the graph becomes empty, then G is showed to be k-colorable. To color G, it suffices to pop nodes from the stack, to insert them back into the graph and to map them to colors (the reverse order of their removal). A node n popped from the stack is inserted into the graph and the edges removed at the moment of its removal are restored. It is mapped to a color

different from all its current neighbors, though it can have some other neighbors not yet re-inserted into the graph.

The allocator proposed by Chaitin et al. handles both spill code insertion and coalescing. The principal phases of this allocator are:

Renumber performs live ranges splitting. Each variable v is renamed every time it is defined. It creates a new sub-variable v_i for each definition of v. At each use point of v, it merges together the v_i of v that reach the use. At the end of this process, each variable v is represented by a set of sub-variables called *names*.

Build constructs the interference graph where each node represents a name.

Coalesce removes unnecessary move instructions. Two names that do not interfere and that are related by a move instruction are coalesced into one node adjacent to the neighbors of the nodes being replaced. When no more coalescing is possible, the graph is re-build to trigger new opportunities for coalescing. The *Build* and *Coalesce* phases are repeated until no more coalescing is possible.

Spill costs computes the cost of spilling each name. The cost of spilling a name is an estimation of the impact of spilling it on the execution time of the program.

Simplify finds the set of nodes to be colored and order them. It first constructs an empty stack and looks repeatedly for a low degree node:

1. if such a node exists, it removes it from the graph and pushes it on the stack.
2. otherwise, all nodes are of significant degree ($degree \geq k$) and it chooses according to the spill costs a node to spill and removes it from the graph.

This procedure is repeated until the graph becomes empty.

Spill Code inserts loads and stores instructions if some nodes have been spilled during the simplification phase. For each spilled name n, it adds a load instruction before every use of n and a store instruction after every definition of n. Thus, n will be replaced into a collection of new names with tiny live ranges. Hopefully, due to their tiny live ranges, these new names will not interfere with several other names. After this step, the whole procedure is restarted from *Renumber*. These iterations from *Renumber* to *Simplify* are repeated until no node is spilled. But, in practice one or two iterations often suffice.

Select assigns colors to the nodes of the graph in the reverse order of their removal. It pops a node n from the stack and inserts it back into the graph. The node is assigned a color different from the colors of all its neighbors.

Figure 1.6 taken from Brigg's paper [BCT94] shows the inter-connexion between the different steps of the Chaitin's allocator.

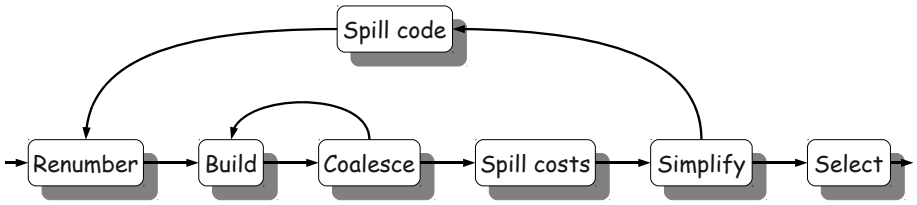

Figure 1.6: The Chaitin et al.'s Allocator

The Chaitin et al. allocator works very well in practice even if it is not flawless and subsequent works have improved it.

Briggs et al. have shown that deferring the *spill code* phase to the *selection* can reduce the number of spilled nodes [BCT94]. This is possible because the condition for finding a low degree node is sufficient but not necessary: a node can have more than k (k being the number of colors) neighbors and be colorable. This improvement is called *optimistic coloring*.

Briggs et al. also pointed out that when the coalescing is done too aggressively, it can make the interference graph uncolorable and lead to excessive spilling. They introduced the notion of *conservative coalescing* which ensures that whenever two nodes are coalesced, the resulting graph remains colorable, if it was the case for the original graph. They used the following criterion to conserve this colorability property: two nodes are coalesced only if the resulting node will not have more than k neighbors [BCT94].

After the improvements of Briggs et al., George and Appel have proposed the *iterated register coalescing* that was mostly focused on coalescing [GA96]. They noticed that the Briggs's allocator was too conservative and left in the program too many move instructions that could be removed. They have shown that interlacing the *simplify* phase with the *coalesce* phase permits to be much more aggressive in the coalescing without turning the graph uncolorable. The basic idea is that, when conservative coalescing is performed before simplification, the move-related nodes of significant degree might not be coalesced, while they sometimes can be turned into low-degree node after simplification. Hence, in the described allocator, the *simplify* phase removes non-move-related nodes one at a time.

Then, when no more simplification is possible, the *coalesce* phase merges move-related nodes in Brigg's style. If a resulting node is no longer move-related, it will be simplified in the next round of *simplify*. The *simplify* and *coalesce* phases are repeated until there remains only significant-degree nodes or move-related nodes in the graph. When such a case is reached, the *freeze* phase looks for a move-related node and mark it as non-move-related. All the remaining moves involving this node will not be removed. Then, the *simplify* and *coalesce* phases are resumed. Figure 1.7, taken from "Modern compiler implementation in Java" [AP02] shows the flow chart of the allocator proposed by George and Appel.

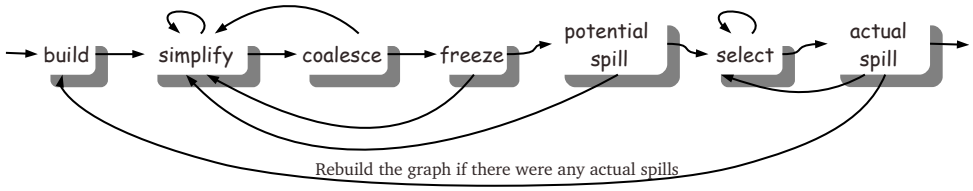

Figure 1.7: Iterated Register Coalescing

1.5 Linear Scan

The linear scan has been introduced by Poletto and Sarkar as an alternative to the graph coloring approach for fast global register allocation [PS99]. It is suitable for applications where compile time is a concern, such as just-in-time compilers or dynamic compilation systems.

The linear scan is based on the notion of *live interval* which is a conservative approximation of a variable's live range. Given some numbering of a code's intermediate representation, $[i, j]$ is said to be the live range of the variable v, if there is no instruction with number $i' < i$, such that v is live at that instruction, and there is no instruction with number $j' > j$ such that v is live at that instruction. There may exist some sub-ranges of $[i, j]$ where v is not live, this inaccuracy in the live ranges approximation is ignored for the sake of fast computation. Live interval information can be computed with one pass through the intermediate representation from live variable information. Figure 1.8 shows the live intervals of variables in the example program in Figure 1.2(a).

The interferences among variables are captured with their live intervals. Two variables interfere if their corresponding live intervals overlap. The linear scan assigns as many live intervals (variables) as possible to a register so that two overlapping live intervals are

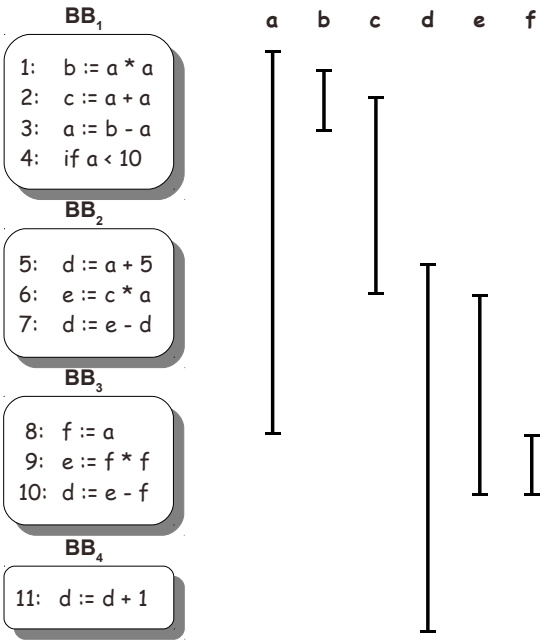

Figure 1.8: Live intervals of the variables in Figure 1.2(a).

assigned different registers.

Algorithm 1 ClassicalLinearScan

Require: *list*: the list of basic intervals ordered by increasing start point
Require: R: the number of available registers
Ensure: *active*: the list of currently live intervals ordered by increasing end point

```
1: active ← ⊥
2: for all i ∈ list do
3:    ExpireOldIntervals(i)
4:    if length(active) = R then
5:       SpillAtInterval(i)
6:    else
7:       register[i] ← r an available register
8:    end if
9:    add i to active
10: end for
```

Algorithm 1 shows the different steps of the linear scan register allocation. The number of registers available on the architecture is called R. It is assumed that live intervals have been computed and *list* is a list that contains all the live intervals sorted by increasing start point. At each step of the algorithm, the list called *active* maintains the live intervals placed in registers that overlap the current point; *active* is sorted by increasing

end point of live intervals. For each starting live interval i, ExpireOldIntervals(i) removes from *active* all the live intervals that do not overlap i's start point. If the size of *active* is lower than R then i is assigned a register and is added to *active*. Otherwise, SpillAtInterval(i) spills the interval that goes further in the future between i and *last*, the last interval of *active*. If *last* is spilled, i is assigned to its register and added to *active*.

The linear scan algorithm is interesting because it is fast, simple and produces codes of relatively good quality. This makes it a serious candidate for register allocation in just-in-time compilers, where compilation time is of the utmost importance. That is why the linear scan register allocation is implemented in the Java Hotspot and the LLVM compilers.

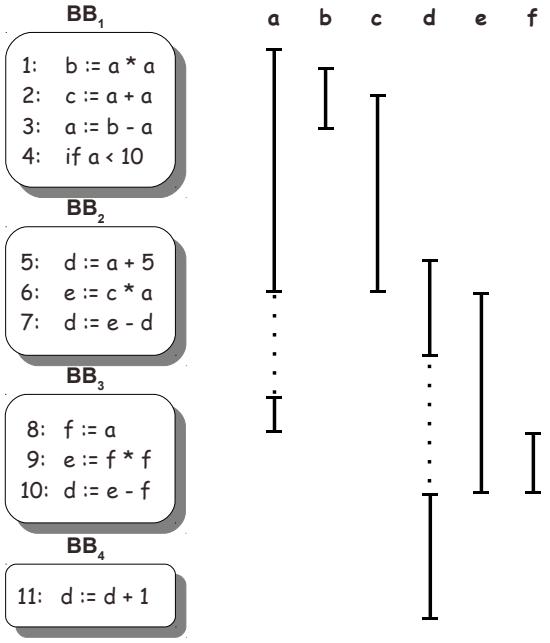

Figure 1.9: Live intervals with holes of the variables in Figure 1.2(a).

A weakness of the original version of linear scan is the inaccuracy of live intervals. Indeed, the live interval characterizing the entire live range of variables may contain some *idle holes*. Those holes correspond either to program points dominated by a redefinition of the variable (the variable is effectively dead at those points), or to a hole resulting from the order in which the basic blocks are numbered (a control-flow artifact). For instance, in Figure 1.9, the variable d is still live when the hole within its live interval starts. Wimmer et al. make use of these holes when implementing an optimized version of the linear

scan algorithm for Java HotSpot compiler[WM05]. Mössenböck et al. also improved the quality of the linear scan thanks to the live interval holes and the SSA form which tends to produce short live intervals [MP02]. The most recent work that we will discuss in the next section, called extended linear scan, has been done by Sakar et al. [SB07].

1.6 Decoupled Register Allocation

We present the new approaches to register allocation that decouple its allocation and assignment phases.

The intuition for decoupled register allocation derives from the observation that live range splitting is almost always profitable if it allows to reduce the number of register spills, even at the cost of extra register moves. The decoupled approach focuses on spill minimization only, pushing the minimization of register moves to a later register coalescing phase [AG01, BDR08].

1.6.1 A first two-phase approach

In 2001, Appel and George have proposed a register allocator adopting a two-phase approach [AG01]. In the allocation phase, they allowed live range splitting at each program point (between every two consecutive instructions), and they formulated an *integer linear program* that was able to find rapidly (tenth of milliseconds) the optimal sets of split points and spills. To ensure that the resulting interference graph was k-colorable, they copied every variable to a freshly named temporary to decrease the register pressure. As a result, they needed to have special care on coalescing, and thus, their assignment phase was dedicated to coalescing and coloring. They used a variant of Park and Moon's optimistic coalescing algorithm and obtained code of good quality [PM98]. Even if there was no theoretical reason showing that solving the spilling and coloring problem separately leads to an optimal solution of the original problem, Appel and George's paper has the virtue of showing that such an approach can give very good results.

1.6.2 SSA-based Register Allocation

In 2003, Anderson tested a huge number of interference graphs from the SML/NJ of Appel and George [And03]. He found that for every interference graph G he tested, $\chi(G) = \omega(G)$. In the same way, Pereira and Palsberg found that 95% of the methods in the Java 1.5 library have chordal interference graphs when compiled with the JoeQ compiler [PP05]. Based on that observation they proposed a greedy algorithm which can optimally color the

chordal interference graphs in linear time according to the number of edges. They also gave good heuristics for coalescing and spilling. Comparing their algorithm with the iterated register coalescing, they produced better results for configuration with few registers and comparable results for configuration with many registers. The works of Anderson and Pereira et al. experimentally validated that many of the interference graphs could be colored optimally and in polynomial time.

In 2005, three teams, namely Bouchez et al., Brisk et al., and Hack et al., independantly discovered that the interference graph of a program in SSA form is chordal. We will report here how Hack et al. show this property of interference graphs of programs in SSA form.

A vertex v of a graph G is *simplicial* if its neighbors form a clique in G. An ordering $v_1, v_2, \ldots, v_n$ of the vertices of a graph G is a *perfect elimination order* (PEO) if each v_i is a simplicial vertex in $G_{\{v_i, v_{i+1}, \ldots, v_n\}}$, the graph remaining from G when all the vertices preceding v_i in the ordering have been removed. Given a PEO of a graph G, it is possible to color G with the following procedure [Gol04]:

1. Remove all the vertices of G in the order they appear in the PEO and push them into a stack.

2. Pop the vertex v on top of the stack, re-insert it in the graph, and assign it to a color not used by any of its neighbors. Since the neighbors of v already present in the graph form a clique, the number of colors used currently is bound by the size of the maximum clique present in the graph.

3. Repeat the step 2 until the stack becomes empty.

This procedure will use exactly as many colors as the size of the largest clique in G and thus gives an optimal coloring of G. It is well known in perfect graph theory that a graph for which there exits a PEO is chordal and therefore perfect [Gol04]. For a perfect graph G the maximum clique number $\omega(G)$ is equal to $\chi(G)$ the smallest k for which it is possible to find a coloring.

Hack et al. have shown that a variable v of an interference graph G of a program in strict SSA-form, where every use of a variable is dominated[1] by its definition, can be added to a PEO, if all variables, whose definitions are dominated by the definition of v have already been added to the PEO. The intuition is that, whenever v is added to a PEO, the only neighbors of v remaining in G are those dominating v; since they dominate

[1] An instruction i dominates another instruction j if all paths, in the control-flow graph, form the entry point to j contains i.

v, they are all live when v is defined and hence form a clique. A PEO of the vertices of G can be obtained by a post-order walk over the program's dominance tree and thus in quadratic time according to the number of vertices of G. Hack et al. also demonstrate that for every clique in the interference graph, there exists a point in the program where all the variables in the clique are live. Thus, a spilling algorithm can make an interference graph k-colorable by reducing the number of live variables at each program point down to k. This shows that the register allocation problem can be decoupled into two phases that can be solved separately:

1. If MaxLive, the maximum number of simultaneously live variables, is greater than k, use a spilling algorithm to decrease MaxLive down to k.

2. Color the chordal graph optimally and coalesce as much as possible.

For register allocation again, SSA permits to ease the optimization problem. Specifically, the SSA-based register allocation techniques collapse the register coalescing with the hard problem of getting out of SSA [HGG06, BDdD+09], as one of the last backend compiler passes.

1.6.3 A Decoupled Linear Scan

Sarkar and Barik have introduced *Extended Linear Scan* [SB07], a new version of linear scan which adopts a decoupled approach to register allocation. Extended Linear scan considers separately the sub-intervals which compose the live intervals (a live interval can be composed of many sub-intervals interleaved with holes). The allocation decisions are performed at the end points of these sub-intervals; we call them decision points. In the allocation phase, at each decision point where *count*, the number of simultaneously living variables, exceeds k, the number of available registers, $count - k$ variables are spilled. In the assignment phase, non-spilled variables are assigned to registers, notice that a non-spilled variable can be assigned to a different register for each of its sub-intervals. This can lead to some inconsistencies in the generated code. For instance, in Figure 1.9, if the variable d is assigned to the register r_1 during the first sub-interval of its live interval and then is assigned to r_2 during the second sub-interval of its live interval, r_2 may hold a wrong value of d if the first branch of the control-flow is taken. Extended Linear Scan eliminates these inconsistencies at the cost of inserting some move instructions.

1.7 Conclusion

In this chapter, we have introduced the background of register allocation upon which our dissertation is based. We have presented the graph coloring and the linear scan approaches to register allocation and we have shown how separating the allocation phase from the assignment helps to ease and improve the register allocation.

Chapter 2

Split Compilation

Traditionally, there are two approaches to translate programs, written in programming languages, into a form that is executable (machine code or native code) on a target machine: compilation and interpretation.

A *compiler* is a software-program that translates a source program, usually written in a programming language, into a target one written in machine code or another programming language. If the target program is executable, it can then be run and fed with inputs to produce outputs. Figure 2.1 illustrates the process of compilation and Figure 2.2 depicts how an executable target program is called for execution. When the compilation is performed *during* the execution process, it is called *dynamic compilation*. Otherwise, it is called *static compilation*.

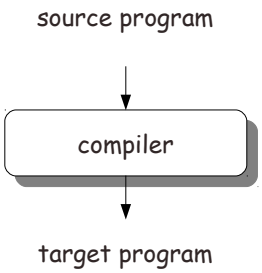

Figure 2.1: Compilation process, from the Dragon book [ALSU06]

Figure 2.2: Execution of a target program, from the Dragon book [ALSU06]

Interpretation is an alternative to compilation. An interpreter directly executes the operations of the source program on the given inputs. Unlike a compiler, an interpreter does not produce a target program. Figure 2.3 describes the process of interpretation.

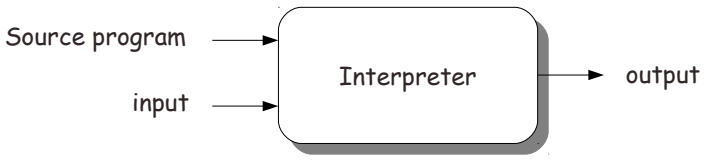

Figure 2.3: Interpretation, from the Dragon book [ALSU06]

Another approach used to translate programs is to combine both compilation and interpretation. In this approach, the source program is generally compiled into an intermediate form, called *bytecode* (e.g. the Java .class files), that: is independent of any particular hardware, cannot be run directly, and is close to machine instructions. This first compilation is usually called *off-line compilation*. The bytecode is then interpreted by a virtual machine, which is a software implementation that emulates the functioning of a physical machine or computer. For instance, such an approach is used for programming languages like Smalltalk [GR83] and Java [GJSB05].

The advantage of the above-mentioned strategy is that the bytecode is portable across many architectures and operating systems for which a virtual machine exists. Indeed, the bytecode can be compiled on a computer and run on another one, or over the network. The imported bytecode is checked for errors before it is executed. This is to prevent misuses which may be deliberate or not. In addition, if the bytecode is at a higher level than machine code, it becomes more compact: it is smaller in size and carry, implicitly, much more semantic information [Ayc03]. This compactness is crucial in environments, like embedded systems, where the code size is an important issue.

2.1 Just-In-Time Compilation

To speed up the execution of programs, some virtual machines use *just-in-time* (JIT) compilers, which dynamically compile (*online compilation*) the bytecode into machine code (native code) just before it is executed. JIT compilation leverages advantages of both static compilation and interpretation. These advantages are listed below:

Execution speed. The machine code produced by a JIT compiler, like one produced with a static compiler, runs usually faster than an interpreted code.

More optimizations. During the execution, some information that is target-dependent or that was unavailable prior to execution becomes accessible. For instance, input parameters, target machine specifics and other types information are often unaccessible before runtime. This additional information enables more optimizations for JIT compilers compared to static ones.

Program analysis and optimizations performed during static compilation can be very expensive in time. Usually, JIT compilers cannot afford expensive program analysis and optimizations since the compilation time is part of the global execution process. Thus, there is a trade-off between the time spent for compilation (optimizations) and the execution time of the produced code.

2.2 Annotations-Enhanced JIT Compilers

In order to reduce the online compilation (optimizations) time and thus the global execution time, some annotation-based techniques have been proposed. These techniques annotate the bytecode (namely Java bytecode) with analysis information that are time-consuming to collect. The works of Pominville et al. [PQVR+01] and Chrintz et al. [KC01] are good examples of such annotations-enhanced JIT compilation.

In order to guarantee the safe execution of Java programs, Java virtual machines must check if the array index exceeds the range and throw an exception (notice of an error) if it is the case. For array-based application, array bounds checks may incur severe overhead at runtime. Pominville et al. [PQVR+01] proposed a method to eliminate array bounds and null pointer checks that can be proven unnecessary by static analysis. They convey information, on usefulness of checks or not, through bytecode annotations.

Following a similar approach, Krintz and Calder [KC01] proposed an annotation framework that reduces the online compilation time overhead of Java programs. They devised four kinds of annotations. The first one was used to transmit to the dynamic compiler statically collected data that are needed for some optimizations (e.g. global register allocation) and that precomputed offline (before the online compilation). The second kind of annotations indicate to the compiler which data should be generated once and stored for reuse instead of being regenerated. The third annotation marks a method for optimization or not, or selects the level of optimizations to perform on a method. This helps to avoid wasting time optimizing methods that will not improve the execution performance. The last kind of annotation guide the selection of profitable optimizations to perform on a method.

The techniques that are presented here reduce the compilation time and thus the global execution time. However, they do not improve the performance of the produced code.

2.3 Split Compilation

The aim of split compilation is to reduce both the compilation and execution times.

The traditional approach to process bytecode language is to distribute the roles among offline and online compilers. Verification and code compaction are typically assigned to offline compilation, while target-specific optimizations are performed by online compilation.

Split compilation reconsiders this notion: it allows *a single optimization algorithm to be split into multiple compilation steps*, transferring the semantic information between different moments of the lifetime of a program through carefully designed (bytecode) language annotations. Split compilation has the potential to combine the advantages of offline and online compilations: running expensive analyses offline to prune the optimization space, deferring a more educated optimization decision to the online step, when the precise execution context is known. Many JIT compilation efforts tried to leverage the accuracy of dynamic analysis to outperform native compilers; but split compilation is a concrete path to get the best of both worlds.

One of the first split compilation works is the AJIT (annotation-aware Just-In-Time) framework of Azevedo et al. [ANH99]. In order to improve the quality of the code generated by the JIT compiler, the authors removed some runtime checks, like array accesses checks, and mostly focused on splitting the register allocation optimization into two steps:

- The first step performed offline implements a variant of priority based graph coloring algorithm. A priority-based coloring algorithm uses heuristics and cost analysis to sort variables. The most frequently accessed variables have highest priority and are assigned to colors first. Based on this priority list and assuming an infinite number of registers, called *virtual registers*, they assign each variable to a virtual register (the most important variables are assigned to virtual registers of lowest numbers). Since real machines have small number of registers, the authors try to maintain the number of used virtual registers as small as possible. That is, many non-interfering variables are assigned to the same virtual register. This information on variables mapped to their virtual registers are transmitted to the online step through annotation.

- The second step of the algorithm, performed online, uses the annotations to retrieve the assignment of variables to virtual registers. It first replaces virtual registers of lowest numbers with physical ones. The variables assigned to virtual registers for which there is no physical register available are spilled.

Azevedo et al. implemented this approach with the public domain JIT compiler system *Kaffe* [Wil96]. They showed that on average the code they produced runs two times faster than the code originally produced by Kaffe. This work, while being interesting, does not handle the live range effect of the calling conventions which sometimes leads to extra spilling and performance degradation. In addition, by moving some array bounds checks and in the absence of analysis in the AJIT virtual machine implementation to verify such accesses, some safety constraints of the Java virtual machine design are violated [Jon02].

Following a similar approach, Jones [Jon02] extended the idea of virtual register annotation with the swap annotations. The swap annotations leverage the non uniform use of variables throughout a method to improve register allocation. For instance, a variable v assigned to a register r cannot be accessed within a loop l. Thus, a variable v' that is frequently accessed within l may be locally assigned to r. Swap annotations were generated to locally increase the priority of a virtual register into the priority of a higher one. In addition, Jones forces that the values held by a given virtual register are of the same type. This permits to verify easily the virtual registers annotations with a simple extension of the standard Java virtual machine verification procedure. These verification guarantees that every virtual register access is compatible with all other accesses. This work also addressed the calling convention issues and portability to deal with architectures that differ in number of registers.

Split compilation is not restricted to register allocation. Indeed, the *split compilation* term was first coined in the context of JIT vectorization [LCC+07].

2.4 Conclusion

This chapter reviews the different approaches to process programming languages. It emphasizes on how split compilation improves the quality of the code generated by JIT compilers.

Chapter 3

Split Register Allocation

3.1 Introduction

Just-In-Time (JIT) compilers rely on continuous, feedback-directed (re-)compilation frameworks to select hot functions (frequently executed) for online optimizations. These online optimizations must make important trade-offs in terms of reducing compilation time for decreased generated code performance. Reducing compilation overhead has two main benefits, low-complexity algorithms simultaneously increase the amount of code being optimized while reducing the compilation time for hot functions. In practice, (quasi-)linear complexity is the rule for JIT compilation. This severely impacts what kind of optimizations are admissible and how aggressive they may be.

3.1.1 A Case for Split Compilation

Traditional bytecode language tool chains distribute the roles among offline and online compilers. Verification and code compaction are typically assigned to offline compilation, while target-specific optimizations are performed by online compilation. *Split compilation* reconsiders this notion: it allows *a single optimization algorithm* to be split into *an offline and an online stage*, transferring the semantic information between those stages through carefully designed bytecode annotations.

Split compilation has the potential to combine the advantages of offline and online compilation: running expensive analyses offline to prune the optimization space, deferring a more educated optimization decision to the online stage, when the precise execution context is known. Many JIT compilation efforts tried to leverage the accuracy of dynamic analysis to outperform native compilers; but split compilation is a concrete path to get the best of both worlds.

To make a concrete case for split compilation, we selected the (spill-everywhere) register allocation problem [CAC+81, BCT94]. Register allocation is an ideal candidate to demonstrate how split compilation impacts the design of future bytecode languages and compilers, and how it differs from plain annotation-enhanced JIT compilation [KC01]. Indeed:

- the principles of register allocation are reasonably well understood;
- it is one of the most important components of all JIT compilers;
- it is challenging to design an offline analysis that would improve online register allocation, while ignoring the exact register count of the target.

3.1.2 Outline of the chapter

This chapter makes two important contributions.

1. We design bytecode annotations enabling a linear-time online algorithm to achieve high-quality register allocation, with negligible impact on the size of the bytecode.
2. We demonstrate how such annotations are robust to variations in the number of registers. With additional provisions in the offline stage, it is even possible to accommodate radical changes in the instruction set target architecture.

Our method is implemented in the JikesRVM open source JIT compiler for Java [Aea05], and evaluated on x86. We do believe that it would be easy to port it to multi-language JIT frameworks like the ECMA-335 CLI standard.[2]

The chapter is organized as follows. Section 3.2 presents the split register allocation flow and algorithms. Section 3.3 evaluates split register allocation, with coverage of performance improvements as well as annotation compaction and portability. Section 3.4 explores more complex compilation scenarios. Finally, Section 6.3 discusses related work on annotation-enhanced just-in-time compilation.

3.2 Split Register Allocation

We first introduce some terminology. An interval characterizing the entire lifetime of a local variable or temporary may contain some *idle holes*. The live range of a variable x is the set of program points where x is live; it corresponds to a union of basic intervals.

[2] http://www.ecma-international.org/publications/standards/Ecma-335.htm

When linearising the control flow (e.g., when generating code), the basic interval of a given live range are interleaved with holes. Those holes correspond either to program points dominated by a redefinition of the variable (the variable is effectively dead at those points), or to a hole resulting from the order in which the basic blocks are numbered (a control-flow artifact). *Register pressure* refers to the amount of locally live variables. Considering that a variable is not alive during its idle holes can help in reducing the register pressure. Jikes RVM takes advantage of this.

3.2.1 Optimization Problem and Baseline Algorithm

Since our primary focus is to illustrate the split compilation concept, we limit ourselves to the most basic register allocation and assignment problem:

- Spill everywhere allocation: spill the whole live range.
- Single-color assignment: when such a live range is allocated, all its basic intervals must be assigned to the same register. Some live ranges may be preassigned due to function call conventions and operand restrictions of some target instructions;

Throughout the chapter, we handle register allocation in different register classes separately (e.g., general purpose, floating point), and call R the number of registers in the current class of interest.

Algorithm 2 recalls the main steps of the linear scan algorithm, as implemented in JikesRVM. Every time a basic interval i becomes active, Algorithm 2 calls the function AssignOrSuggestSpillCandidate($V(i)$), where $V(i)$ is the live range corresponding to i. According to the allocation that has been performed up to this point, function AssignOrSuggestSpillCandidate($V(i)$) returns, either a live range or $\perp$ (bottom): if it returns a live range, it is the one to be spilled in order to continue allocation; if it returns $\perp$ it was possible to assign $V(i)$ without spilling. These algorithms are the basic framework upon which the offline and online phases of our split register allocation are constructed.

3.2.2 The ILP Model

Here, we discuss our formulation of spilling in register allocation as an ILP problem. We obtain spilling decisions offline and pass this information to the online compilation phase using annotations. Considering a set S of live ranges, a *spill set* of S is any subset S' of S such that $S \setminus S'$ can be allocated over the R registers (without spilling). We also consider

Algorithm 2 LINEARSCAN

Input: *list*: the list of basic intervals ordered by increasing start point

```
1: foreach: i ∈ list do
2:    toSpill ← ASSIGNORSUGGESTSPILLCANDIDATE(V(i))
3:    if toSpill ≠ ⊥ then
4:       if toSpill ≠ V(i) then
5:          Assign V(i) to the register freed by toSpill
6:       end if
7:       Spill toSpill
8:    end if
9: end for
```

Return: sets of spilled live ranges and register assignments

Algorithm 3 ASSIGNORSUGGESTSPILLCANDIDATE

Input: v: a live range

```
1: if v was previously assigned to a register r then
2:    if r is free then
3:       Continue with this assignment
4:       Return ⊥
5:    else if v can be assigned to another register r' then
6:       Assign v to r'
7:       Return ⊥
8:    else
9:       Let v' be the live range assigned to r
10:      Return the live range with the minimum cost among v and v'
11:   end if
12: else if v can be assigned to a free register r then
13:   Assign v to r
14:   Return ⊥
15: else
16:   Return v' with the lowest cost among v and the other live ranges at the current
      point
17: end if
```

Return: a live range to spill or $\perp$

a function which assigns to each live range in S the cost of spilling it. The cost of a spill set is the sum of the costs of live ranges within that set. An optimal register allocation is associated with a spill set with the minimal cost.

We build an ILP model that is optimal among spill-everywhere, single-color allocations, for a given cost model.

We model register allocation as a $\{0, 1\}$ linear program, the objective function being the cost of the spill set. We support multiple classes of registers, each register class

is further decomposed into 2 subclasses: caller-saved (scratch register) and callee-saved (non-scratch register). Live ranges are partitioned according to register classes, and can be of the *volatile*, *non-volatile* or *preassigned* kinds: a *non*-volatile live range can only be assigned to some callee-saved register, and a preassigned live range can only be assigned to a specific physical register.

We create a $\{0, 1\}$ variable l_r for each live range l and register r that l may be assigned to (considering class and volatility constraints):

$$l_r = 1 \text{ if and only if } l \text{ is assigned to } r.$$

These variables are constrained by 3 kinds of (in)equalities.

1. At most one register per live range (single color assignment):

$$\sum_{1 \leq r \leq R} l_r \leq 1$$

.

2. Interfering live ranges cannot be assigned to the same register: $l_r + l'_r \leq 1$.

3. The third constraint states that if a live range l interferes with a live range l' preassigned to r, then $l_r = 0$.

3.2.3 Annotation Semantics

The offline stage generates annotations that can be used by an online stage to characterize important properties of some live ranges. The online stage may run on a target that may not match what was used to generate the annotations in the offline stage. This triggers portability problems: we address register count variations in this section, and defer the discussion of other problems to Section 3.4.

In the context of register allocation, the most specific portability issue is related to variations in the number of registers. To define portable annotations, it would be ideal to prove a general result about the inclusion of an optimal spill set for a given number of physical registers into one of the optimal spill sets for a lower number of registers. Unfortunately, this is not true in general. Figure 3.1 shows a counter example on the allocation of 5 live ranges — the horizontal bars. Every number on top of a horizontal bar denotes the cost of spilling the corresponding live range. Dashed black lines correspond to spilled live ranges. For the left graph, we assume $R = 2$ registers. For the graph on the right, $R = 1$ register only. When $R = 2$, we may optimally spill i_3 to assign i_1 and

i_4 to one register and to assign i_2 and i_5 to the another one. When $R = 1$, the single optimal allocation is to spill i_2 and i_4 and to assign i_1, i_3 and i_5 to the single register. In this example we see clearly that an optimal spill set for two registers is not included in the optimal spill set for one register.

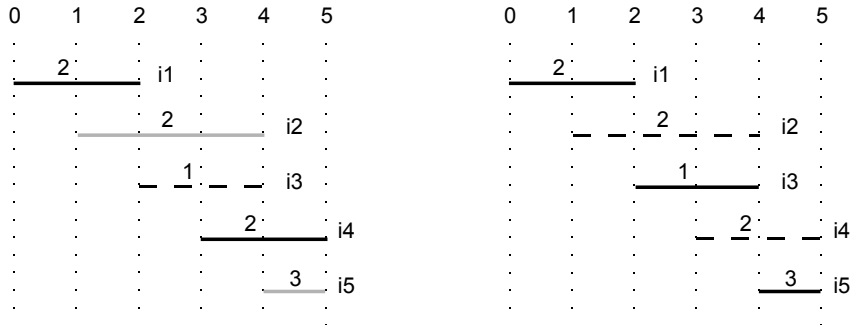

Figure 3.1: Counter example to spill set inclusion

Although such an inclusion property does not always hold, we experimentally validated that only few live ranges should be spilled for $R + 1$ registers but allocated for R registers. For example, considering the x86 instruction-set architecture, when moving incrementally by one register from the minimum number of registers, to a spill-free[3] number of registers for each method, inclusion property was violated for only 0.13% of the live ranges over the whole SPEC JVM suite. This validates the intuition that the semantics of an allocate/spill-oriented annotation is portable across variations in the register count.

3.2.4 The Offline Procedure

Our split register allocation procedure derives from three key observations.

1. First, once the ILP solver finds an optimal spill set, it would be possible to directly annotate the code with the best spill set. This can lead to annotation bloat (although linear), with total annotation size potentially larger than the bytecode itself. Jones and Kamin do not address the problem [JK00].

2. Second, the more detailed the annotation, the more sensitive it is to low-level decisions on instruction selection and scheduling that may happen after register allocation. To make the annotation portable, it is important to focus it on semantic properties that preserve the essence of the offline optimization while maximizing independence w.r.t. post-pass optimizations in the online compilation stage. The

[3] until we reach a number of register for which allocation can be done without spilling

idea here is to focus the annotation on long live ranges whose interferences do not vary much w.r.t. post-register allocation instruction selection and scheduling. Indeed, short live ranges are likely to be allocated due to their limited interferences and high-rate register usage.

3. Third, notice that a greedy allocation algorithm is typically too conservative, allocating a live range that should have been spilled or assigning an inappropriate register/color. This means that annotations should only pertain to "must-spill" information.

With those three observations in mind, we devised Algorithm 4. The intuition behind this algorithm is natural: why store annotations for live ranges on which a greedy, linear procedure can readily make the right decision?

The algorithm uses an oracle-driven version of the linear scan. Every time the greedy heuristic wishes to spill a live range which does not belong to the annotations, the algorithm forces it to spill a live range which is currently active and which belongs to the annotations. By doing so, we discover live ranges in the optimal spill set that the linear scan *cannot* find on its own.

Considering Algorithm 4, at a step where live range $V(i)$ is active (according to the allocation performed since the beginning of the method being allocated), function ASSIGNORSUGGESTSPILLCANDIDATE($V(i)$) returns, either a live range or $\perp$ (bottom): if it returns a live range, it is the one to be spilled in order to continue allocation; if it returns $\perp$ it was possible to assign i without spilling. Function FINDACTIVELIVERANGE(*optimalSpills*) returns a currently active live range that is in the set *optimalSpills*, and set *annotation* records live ranges that will not be found by the linear scan.

The algorithm returns live ranges that will not be optimally allocated by the linear scan and keeps those as the constituents for the compressed annotations.

The final step consists of pairing the live ranges returned by Algorithm 4 with a "must-spill" tag. This pairing should be as economical as possible to represent, but it should also make sense across different targets and carry relevant allocation information. For each live range l, we compute the maximal value of R for which l must be spilled, denoting it as $R_{\max}(l)$. We do not care much about offline compilation time in this study: the computation thus boils down to iterating the ILP model over decreasing values of R, pre-spilling live ranges spilled at the previous step (for $R+1$ register) to guarantee inclusion.

Finally, annotated live ranges need to be stored in a compact persistent format, together with the bytecode program. Rather than storing every pair $(i, R_{\max}(l))$, we cluster

Algorithm 4 compressAnnotation

Input: *list*: the list of basic intervals ordered by increasing start point
Input: *optimalSpills*: the set of live ranges to be spilled as decided by the optimal allocator

```
1:  annotation ← ∅
2:  foreach: i ∈ list do
3:    toSpill ← assignOrSuggestSpillCandidate(V(i))
4:    if toSpill ≠ ⊥ then
5:      if toSpill ∉ optimalSpills then
6:        toSpill ← findActiveLiveRange(optimalSpills)
7:        annotation ← annotation ∪ toSpill
8:      end if
9:      if toSpill ≠ V(i) then
10:       Assign V(i) to the register freed by toSpill
11:     end if
12:     Spill live range toSpill
13:   end if
14: end for
```

Return: *annotation*: the compressed annotations

live ranges with the same value of $R_{\max}(l)$, sort those clusters, and serialize the list of live ranges in every cluster, prepending each cluster's list with the corresponding value of $R_{\max}(l)$. We end up with separate strings, one for each size s of the register set, listing the live ranges that must be spilled for s registers and that were *not* already listed in a string associated with size s' greater than s. This way, most of the space is used to store live range names, for which we conservatively count up to 4 bytes per live range.

3.2.5 The Online Procedure

The online stage performs allocation based on a compact spill set collected by the offline stage, and carried as bytecode annotations.

Our online algorithm follows the steps of Algorithm 2. In addition, at the beginning of every basic interval , it checks whether the corresponding live range is present in the annotation. *If so, then spill it* (if the live range was not previously spilled).

This algorithm takes its roots in the decoupled allocation/assignment approach. As our experiments will confirm, the annotation-enhanced linear scan algorithm results in a much better quality allocation. Yet it does not optimally preserve the information available in the annotation and may yield spurious spill code. The reason is simple: register *assignment* on a colorable (spill-free) graph is equivalent to a graph coloring decision problem, which is NP-complete on live ranges [CAC+81]. It is *not* NP-complete with sufficient

Algorithm 5 ONLINEALLOCATION

Input: *list*: the list of basic intervals sorted in increasing start point
Input: *annotation*: a set of annotated live ranges

```
1: foreach: i ∈ list do
2:   if V(i) is not spilled then
3:     if V(i) ∈ annotation then
4:       Spill V(i)
5:     else
6:       ASSIGNORSUGGESTSPILLCANDIDATE(V(i)
7:     end if
8:   end if
9: end for
```

Return: sets of spilled live ranges and register assignments

live-range splitting: linear complexity can be achieved on SSA form following a perfect elimination order — a greedy reverse post-order traversal of the SSA graph [BDR07c]. It is clearly the way to go for optimality preservation, but it also implies a major engineering endeavor that has not yet been undertaken in a full-scale JIT compiler. Fortunately, the interference graphs that arise in non-SSA code are "mostly" chordal [PP05], which guarantees the existence of a perfect elimination order in most cases; this motivates the decoupled approach and explains the observed quality of our online algorithm.

3.3 Experimental Evaluation

We implemented split register allocation in JikesRVM version 3.0.1 [Aea05], relying on CPLEX[4] for the offline resolution of optimal allocation problems.

3.3.1 Methodology

To assess the cost of a spill, we need to define the optimal solution we are aiming for. The cost model of the spill-everywhere problem is implemented in Jikes RVM; it combines dynamic edge profiling, static use count and instruction type.

We illustrate split register allocation on SPEC JVM benchmarks. Experiments on the DaCapo benchmarks [Bla06] could not be included at the time of the submission, but we are working hard on it. We target a 2.67GHz Intel Core 2 Quad, running in 32-bit mode, in a PC platform. This configuration is favorable to register allocation experiments due to the low number of registers, although the cost of spilling is often marginal due to

[4] `http://www.ilog.com/products/cplex`

out-of-order execution and to the sophisticated memory hierarchy.

Each figure was obtained from 100 individual runs of the benchmark, eliminating the 10% best and 10% worst performing points. We did not conduct a systematic statistical study of the performance distribution. Instead, we eliminated the largest source of variation by selecting a non-adaptive, aggressive (maximal optimization), profile-directed strategy (with embedded replay), using the following compilation flags:

```
-Xmx1024M -Xms1024M -X:irc:O3 -X:aos:enable_recompilation=false
-X:aos:initial_compiler=opt -X:aos:enable_replay_compile=true
-X:vm:edgeCounterFile=my_edge_counter_file
```

Split compilation is of course compatible with adaptive optimization. This methodology differs from the standard practices in that we do not run an adaptive compilation scheme [Bla06, GEB08]. We claim our methodology is relevant in the context of split compilation:

- it eliminates the instability triggered by monitoring-based decisions, allowing to focus on the effect of the register allocation itself;
- an adaptive execution methodology is needed to compare the relative contributions of JIT-compilation, monitoring, garbage collection, and the effect of the optimizations themselves [GEB08]; our methodology allows for a fair comparison nonetheless, since the online stage of the split allocation does not introduce significant overhead w.r.t. the original linear scan implementation.

Thanks to its Java API, it was easy to connect CPLEX to our framework. The total resolution time for the optimal register allocation of all SPEC JVM benchmarks — running with aggressive optimization including inlining and unrolling — takes less than 4 minutes on a Core 2 Quad processor at 2.67GHz with 4GB of RAM.

3.3.2 Performance Results

Benchmark	check	compress	jess	raytrace	db	javac	mpegaudio	mtrt	jack
Live ranges (number)	86672	86870	181396	122993	93055	406348	127847	122755	220871
Annotations (number)	77	105	214	191	98	685	315	195	236
Compression %	0.09%	0.12%	0.12%	0.16%	0.11%	0.17%	0.26%	0.16%	0.11%
Optimal spill set (number)	2950	2984	6408	3765	3210	16821	3830	3877	6400
Remaining spills %	2.60 %	3.51%	3.34%	5.07%	3.05%	4.07%	8.23%	5.03%	3.69%
Bytecode %	0.9%	6.9%	0.9%	6.9%	3.4%	0.5%	0.9%	1.1%	0.6%

Table 3.1: Annotation compression

Benchmark	check	compress	jess	raytrace	db	javac	mpegaudio	mtrt	jack	average
Original Jikes RVM	1.31	1.38	1.16	1.19	1.59	1.41	1.39	1.14	1.27	1.32
All live ranges Annotation	1.02	1.30	1	1.17	1.01	1.25	1.03	1.19	1.03	1.11
LIR live ranges Annotation	1.02	1.30	1	1.17	1.01	1.25	1.03	1.19	1.03	1.11
Java local variables Annotation	1.25	1.44	1.02	1.19	1.59	1.36	1.32	1.13	1.18	1.28

Table 3.2: Allocation cost normalized to optimal

Benchmark	check	compress	jess	raytrace	db	javac	mpegaudio	mtrt	jack	average
All live ranges Annotation	0%	12.0%	-1.0%	0.9%	-0.4%	-0.6%	7.5%	1.2%	0.2%	2.2%
LIR live ranges Annotation	0%	12.1%	0.2%	1.0%	-0.3%	-0.7%	5.1%	1.1%	0.2%	2.1%
Java local variables Annotation	0%	5.1%	0.8%	0.0%	-0.3%	-0.2%	-1.4%	1.1%	-0.3%	0.4%

Table 3.3: Wall-clock speedups of split register allocation

Table 3.1 illustrates the effectiveness of the annotation compression scheme: it shows the total number of live ranges (*Live ranges*); the effective number of live ranges within the annotations (*Annotations*); the *Annotations/Live ranges* ratio (*Compression*, in percentage); the number of live ranges within the optimal spill sets (*Optimal spill set*); the *Annotations/Optimal Spill set* ratio (*Remaining spills*, in percentage); and the size overhead w.r.t. the bytecode itself (*Bytecode*, in percentage, counting 4 bytes per annotation).

Preserving the information collected in the offline stage requires at most 0.26% of the live ranges to be annotated. This is several orders of magnitude more effective than state-of-the-art approaches [JK00], and even comes with a formal guarantee about optimality. The addition compression row reports the benefits of Algorithm 4, and confirm its important role in making the annotation size negligible w.r.t. the bytecode size.

Table 3.2 Considers the analytical cost model of Jikes RVM as a metric. *All live ranges annotation* correspond to annotation produced by Algorithm 4; *LIR*[5] *live ranges annotation* correspond to the intersection between the set of live ranges present in the LIR and *all live ranges annotation*; *Java local variables annotation* correspond the set of Java local variables present in *all live ranges annotation*. Table 3.2 shows the penalty (allocation cost / optimal cost) of using the *Original Jikes RVM* (linear scan), *All live ranges annotation*, *LIR live ranges annotation* and *Java local variable annotation* methods in terms of percentage of the optimal spill cost achieved by the ILP model. The Jikes RVM linear-scan misses the optimal cost by 32% on average, whereas the split allocation only incurs a 11% average penalty. The case for annotation portability is validated by the very close figures for the full annotation (All live ranges) and the LIR-only annotation (*LIR live ranges*). However, when only annotating Java variables, the annotation loses its effectiveness. Using the LIR-only annotation appears as the best performance/portability trade-off.

[5]Low-level Intermediate Representation of JikesRVM, which does not include yet all the characteristics of the target architecture.

Considering wall-clock execution time as a metric (JIT compilation plus execution time), Table 3.3 shows the speedup of split register allocation w.r.t. original JikesRVM's allocation algorithm. In most cases, the speedup is consistent between the optimal and split approaches. Nevertheless, the annotation does not help much on some benchmarks like `javac`. The strong improvement in the corresponding column in Table 3.2 indicates that the cost model itself misses the complex interplay between optimizations and important components of the target architecture.

3.3.3 Portability Across Variations of the Register Count

We showed there is no formal inclusion property among optimal spill sets in general. Nevertheless, for every method and among millions of live ranges, we varied R from a minimum equal to the number of pre-allocated physical registers for the method to the spill-free number of registers. Through all these allocation problems only 0.13% of the intervals spilled for $R+1$ registers did not belong to the optimal spill set for R registers.

To make the annotation portable across variations in the register count, the compression algorithm must not eliminate a live interval that may be useless for a given number of registers but useful for a smaller number of registers. We thus run Algorithm 4 on $R = R_{\min}$ registers, where $R_{\min}$ is the minimal number of registers to enable code generation on the target.

3.4 Looking Forward

So far, we ignored important issues related with the practical applicability of split register allocation.

3.4.1 Portability of the Annotation

Let us first consider the portability of annotation names. The names of the annotated live ranges must remain consistent between the two stages. Some annotations may be missing or extraneous, but an annotation designating a live range during the offline stage must designate to the same live range during the online stage. There are practical solutions for most portability scenarios.

1. The majority of live ranges correspond to Java variables, locations in the operand stack, and other live ranges synthesized in the intermediate, target-independent passes of JikesRVM (the LIR). For those live ranges, a non-ambiguous name can

be crafted that is independent of the execution context when the JIT compiler is triggered.

2. A fraction of live ranges are synthesized along the target-dependent compilation flow: address computation temporaries, conditional predicates, etc. We discard annotations regarding those live ranges when compiling for another instruction-set architecture (ISA).[6] Fortunately, besides representing a small minority, these live ranges also feature a very short temporal locality and a low degree of interference with other live ranges. This reduces the chances of impacting an important allocation decision that would result in a significant performance difference. Indeed, we showed that annotation associated with target-dependent live ranges have negligible impact on performance.

Besides the live range names, annotation properties themselves need to be portable over multiple targets: liveness properties may vary significantly over the targets if no assumption is made on the optimization flow. To achieve portability, we thus make one important assumption: optimizations selected by different JIT compilers must not vary significantly *before* the pass where annotations are loaded and attached to the intermediate representation. This restriction does not impact target-specific, post-register allocation passes like instruction selection and local scheduling.

This restriction does not solve all portability problems: reusing annotations across ISAs remains an issue. There are multiple reasons to be optimistic. Some of these are due to the context in which JIT compilation is employed, and some to the nature of the optimizations being performed before register allocation:

- Embedded system designs value the code compression and safety benefits of bytecode languages, but do not stress portability to the extreme. Although many processors and hardware configurations may exist, Java or CLI applications are likely to run on some variant of the ARM instruction set. Varying the number of registers is important to support the ARM's compact instruction encoding options, and to support extensions like vector instructions of ARM NEON. On general-purpose platforms, an analogous situation holds, with portability issues from the 32 and 64 bit variants of the x86 instruction set, different vector instruction sets and sizes, etc.

- Bytecode languages are important for link-time optimization. Complex software architectures built of thousands of independently designed components bring many

[6] Such annotations remain usable when varying the register count (or the calling convention) for a given ISA.

opportunities for inter-module optimization at link-time. Again, the ISA portability issue is only secondary to many of these applications.

- Beyond ISA portability, bytecode languages are used for operating system portability. In this case, the JIT compiler is minimally impacted, and annotations are expected to be robust to changes to the underlying OS.
- Eventually, the software provider may easily specialize the offline stage to generate annotations for a particular family of targets and for a particular optimization flow, tagging the annotated bytecode accordingly. This consists of constructing a (lossless) union annotation considering all live ranges that occur when compiling to the different targets. Since many live ranges will remain the same (e.g., those associated with Java local variables and constant pool, as opposed to operand stack or target-specific temporaries), the union will not significantly increase the size of the annotation.

3.4.2 Separate Compilation

Realistic compilation scenarios will run the offline stage separately on the different modules of the application and on its library dependences. This raises a modularity problem for any annotation-based online compilation approach.

In the context of object-oriented and functional languages, function inlining is of utmost importance to reach performance levels on par with lower level imperative implementations. It raises the following dilemma:

- what is the point of annotating code in functions that will later be inlined, since the effective interference graph will only be known after inlining;
- what is the point of annotating functions whose calling context heavily influences the internal control flow, hence the spill costs?

Our approach to modular split compilation is twofold.

No performance regression. First of all, if one module depends on a module without annotations (such as a package form the Java Development Kit), only the code in the annotated module will benefit from split compilation. This is not ideal, but not worse than the usual penalty of separate compilation in offline, static compilers. Conversely, when optimizing a "library" module, it is always possible to run a *context-insensitive* split-compilation flow, relying on a representative execution profile; this again is consistent with the traditional way of optimizing libraries in static compilation.

Multiversioning for cross-boundary optimization. Nevertheless, JIT compilation opens many opportunities for *link-time optimization*, and JIT compilers for object-oriented and functional languages do implement such advanced techniques, effectively optimizing across module boundaries (e.g., across application-library boundaries). Split register allocation is possible in this context.

First of all, a *context-sensitive annotation* of the callee can be tuned according to the most frequent calling context(s). This is only impactful when the costs of the live ranges depend on the calling context, which may be the case when the callee contains complex, data-dependent control-flow.

A more aggressive approach consists in generating multiple versions of the annotations for the most frequent call trees. For example, if a library method m_2 is frequently called from an application method m_1, the offline stage of the split register allocation may inline m_2 into m_1, optimize the resulting new method, and generate the annotation for it. This specialized version of the inlined methods can later be checked for consistency with the dynamic execution context (indeed, the library code may have changed in the mean time, or dynamic class loading may have occurred), and used directly in favor of performing all the optimizations online and dropping the (irrelevant) per-method annotation. Practical ways to implement this scheme have been proposed in the QuickSilver project [SBMG00]. This scheme has all the benefits of running a JIT compiler offline (better optimizations, lower overhead) while preserving modularity (up to dynamic class loading) and the effectiveness of split compilation.

3.5 Related Work

Annotations are an optional part of the Java bytecode specification from the start and are part of the *class file attributes*. They have been used in debugging and integrated development environments. Syntactic support has been added in recent versions of Java. The same applies to the ECMA-335 CLI.

Interestingly, annotation-driven JIT compilation was first directed to register allocation, with the pioneering work of Azevedo et al. [ANH99]. This work demonstrated how to achieve performance competitive with native priority-based graph coloring allocation. Jones and Kamin [JK00] extended their virtual register allocation approach, dealing with correctness, calling conventions and portability (addressing variations of the number of physical registers only).

The *split compilation* term was first coined in the context of JIT vectorization [LCC+07]. Split register allocation improves on Jones and Kamin's annotation-driven approach by

leveraging the decoupled allocation (spilling) and assignment (coloring) phases of register allocation. Decoupled register allocation is the key to the compactness and the portability of our annotation. The intuition behind decoupled register allocation is that the assignment problem (mapping of variables to registers with no additional spill) is very easy, as long as the cost of live-range splitting (the introduction or register moves) is neglected. This intuition is backed by the important property that spill-free assignment is always possible if the maximal number of simultaneously live variables (MaxLive) is lower than the number of available registers. The online stage can rely on the colorability guarantee inherited from the offline stage through the annotation: these strong ties between the offline and online stages are specific to *split compilation* algorithms, as opposed to classical annotation-driven JIT compilation.

A fully decoupled approach has been used by Appel and George [AG01], and studied in the context of SSA-based register allocation [PP05, HGG06, BDGR06b]. Notice that recent versions of the linear scan algorithm are capable of live range splitting [WM05, SB07]; they are implicitly based on this decoupled approach. This is not the case for the linear scan implemented in JikesRVM, and leads in practice to spurious spills (to our disadvantage), as we confirmed in our evaluation.

Pominiville et al. [PQVR+01] used annotations to mitigate the performance penalty of Java pointers and arrays, and designed a generic annotation-driven compilation framework (Soot). Eventually, Krintz and Calder [KC01] proposed a comprehensive method to reduce the compilation time overhead through bytecode annotations, enabling rapid method selection and optimization selection, and precomputing simple method statistics.

Several papers address two additional important questions related to register allocation in JIT compilers: is there any room for performance improvement, and is it important to use a linear-time allocation algorithm? Cavazos provides an original answer relying on adaptive optimization [CMB06]. Annotation-enhanced versions of this method would be worth investigating.

When using annotations for optimization, safety issues immediately arise because of incorrect or malicious uses. Solutions can be found in proof-carrying code [Nec97], encryption, or correct-by construction annotation designs. We choose the latter approach, relying on annotations whose misuse can at worst lead to performance degradations.

3.6 Conclusion

We designed a split compilation framework dedicated to register allocation. We experimentally validated the effectiveness of split register allocation and its portability with

respect to register count variations, relying on annotations whose impact on the bytecode size is negligible. This combination of results is a strong improvement over the state of the art. It was made possible by revisiting the decoupling of the spilling and coloring (a.k.a. assignment) phases.

Nevertheless, the approach still depends on the stability of the upstream optimization flow in the JIT compiler. Although this restriction is acceptable in a majority of use cases, it would be useful to design a split register allocation framework that would be more robust to changes in the optimization flow. One direction of work consists in revisiting the context of pre-pass allocation to control register-pressure by inserting additional constraints in the data dependence graph [TE04a]. This would accommodate for scheduling (local and global) changes, and possibly for code motion, redundancy elimination and hoisting as well.

Chapter 4

Iterated-Optimal Register Allocation

Recent works on register allocation have shown that when enough live range splitting is allowed, the assignment problem can be solved in quadratic time as soon as the maximum number of simultaneously living variables is lower than the number of available registers. However, the spill minimization problem (an allocation of minimal cost) is NP-complete even when enough live range splitting is allowed [FCL00]. Thus, good techniques are needed to solve this problem and especially on systems like CISC machines where only few registers are available.

In this chapter we address the spill minimization problem. Our aim is to achieve fast allocations that are close to optimal ones. We present an iterative register allocation algorithm, called *iterated-optimal allocation*, which can be used to perform allocation in both decoupled and non-decoupled contexts. We compare it with the graph coloring, the linear scan, a newly devised heuristics called mixed heuristic, and an optimal ILP-based register allocation algorithm. The results show that the iterative ILP-based algorithm outperforms the other heuristics and is often close to the optimal. An interesting result of our approach is that, for SSA programs, the iterated-optimal allocation method has a pseudo-polynomial time complexity.

4.1 The Approach

In the rest of this chapter we address both the spill everywhere and the register allocation problem with live range splitting. We assume that an estimated spill cost has been computed for each variable. A spill cost represents the access frequency of a variable, it is high when the variable is frequently accessed and low when it is not. We denote R the number of available registers and $R_{\min}$ the minimal number of registers to enable code generation on the target architecture. We assume that $R_{\min}$ is small, in our experiments

with JikesRVM running on x86 machine, $R_{\min}$ is equal to 2.

The spill minimization problem is NP-complete for programs in SSA form [BDR07c, YG87] and thus for general programs. An important result, proven independently by Yannakakis et al. [YG87] and Bouchez et al. [BDR07c], is that the spill minimization problem for SSA programs is pseudo-polynomial. That is, it is solvable, by dynamic programming, when R is fixed to a small number. The intuition of our solution derives from this result. Our approach is to solve the spill minimization problem for general programs with R registers by iteratively optimally solving the spill minimization problem with few registers. Even if, the spill minimization problem is only pseudo-polynomial for SSA programs, we validate in our experiments that, for general programs, iteratively solving the the spill minimization problem with few registers gives results close to optimal while being much faster than directly solving the optimal problem with many registers (with R registers, where R is not constrained to be small).

Algorithm 6 ITERATED-OPTIMAL ALLOCATION

Input: *var_list*: The list of variables
Var: *allocated_list*: The list of so far allocated variables

```
1:  result ← OptimalAllocation(var_list, R_min)
2:  add every variable of result to allocated_list
3:  remove every variable of result from var_list
4:  count ← R_min
5:  while var_list ≠ ⊥ ∧ (count + step) ≤ R do
6:     result ← OptimalAllocation(var_list, step)
7:     add every variable of result to allocated_list
8:     remove every variable of result from var_list
9:     count ← count + step
10: end while
11: if count < R then
12:    result ← OptimalAllocation(var_list, R − count)
13:    add every variable of result to allocated_list
14:    remove every variable of result from var_list
15: end if
16: return allocated_list
```

Algorithm 6, which depicts our approach, receives as input *var_list*, the list of variables that are candidate to register allocation. It then returns as result *allocated_list*, the list of variables that have been allocated with R registers. In its first step (lines 1 to 4), Algorithm 6 calls the function OPTIMALALLOCATION which returns the optimal set of allocated variables minimizing the spill cost (*optimal allocation set*), with a register count of $R_{\min}$. It adds the returned variables to *allocated_list*, and then removes them

from *var_list*. In its second step (lines 5 to 10), Algorithm 6 iteratively finds the *optimal allocation set*, with a register count of *step*, among the variables that have not yet been allocated (currently in *var_list*). Like $R_{\min}$, the parameter *step* is a small number (we used 2 in our experiments) which guarantees that the function OptimalAllocation will be polynomial in the case of SSA programs and fast in the case of general programs. In its last step (lines 11 to 15), Algorithm 6 finds the set of variables that minimizes the spill cost among the variables remaining in *var_list* if $R - count$ registers are available. We can notice that $R - count$ is smaller than *step*. The function OptimalAllocation optimally solves the spill minimization problem, e.g. by dynamic programming, integer linear programming (ILP), or logic programming.

In Algorithm 6, the number of registers available at each iteration is limited to *step* registers because we do not want to increase the complexity of the problem, with registers that have been considered and allocated at the previous step.

Close to optimal. Our method described in Algorithm 6 does not produce an optimal allocation. This is for the same reason as the spill set inclusion does not always hold as shown in Figure 3.1. Let us assume that R is equal to 2, $R_{\min}$ is equal to one, *step* is also set to one, and we run Algorithm 6 on the five variables: i_1, i_2, i_3, i_4 and i_5. In its first step, Algorithm 6 will find the optimal allocation set composed of variables i_1, i_3 and i_5, and mark them as allocated. In its second step, it will find an optimal allocation composed of either i_2 or i_4. Finally, Algorithm 6 will return either $\{i_1, i_2, i_3, i_5\}$ or $\{i_1, i_3, i_4, i_5\}$ of spill cost 2 as result, while the optimal allocation set is $\{i_1, i_2, i_4, i_5\}$ of spill cost 1.

Pseudo-polynomial on SSA programs. For SSA programs, the complexity of the function OptimalAllocation is polynomial, if it is solved by dynamic programming and for a small register count. Thus, Algorithm 6 is also polynomial if $R_{\min}$ and *step* are chosen to be small numbers.

4.2 Experimental Evaluation

4.2.1 Methodology

In order to evaluate our approach, we have dumped all the register allocation problem instances obtained during the JIT compilation of the methods of the SPEC JVM 98. The methods have been compiled with the following options of JikesRVM:

```
-Xmx1024M -Xms1024M -X:irc:O3 -X:aos:enable_recompilation=false
```

```
-X:aos:initial_compiler=opt -X:aos:enable_replay_compile=true
-X:vm:edgeCounterFile=my_edge_counter_file
```

In our experiments, the function OPTIMALALLOCATION, which returns the optimal allocation set, is implemented by ILP. $R_{\min}$ is fixed to 2 because, on x86 architecture JikesRVM performs precoloration with two registers. The variable *step* is set to 2.

We also considered different configurations of register count going from 2 to 16 registers. For each instance of the register allocation problem and for each configuration, we compared our approach with the following algorithms:

1. Default linear scan: the linear scan algorithm which takes advantage of holes.
2. Belady linear scan: the linear scan algorithm which takes advantage of holes and which uses the furthest first strategy to choose between two variables to spill if their costs are close enough according to a chosen threshold.
3. Mixed heuristic: an algorithm which chooses the algorithm which has the smaller spill cost among the linear scan and an algorithm which spills as a priority the most constrained variables.
4. GC: the graph coloring algorithm.
5. Optimal: an optimal ILP-based allocation algorithm which computes the optimal allocation.

4.2.2 Results

Figure 4.1 shows the allocation costs of all the benchmarks for each register count. The allocation costs have been normalized using the optimal allocation's cost. The iterated-optimal allocation algorithm is close to the optimal except for register counts 14 and 16, it always outperforms the others.

Figure 4.2 reports the allocation costs for each benchmark, normalized using the optimal allocation's cost, when the register count is 6, the register count of x86 machines. We see that here again, the iterated-optimal allocation algorithm performs close to optimal allocation and outperforms all the other allocation algorithms.

Table 4.1 shows, for all benchmarks, the time spent in milliseconds to solve the ILP-programs of the optimal and the iterated-optimal allocations. We see that when the register count is 2, the time spent to solve optimal ILP-programs and iterated-optimal are almost the same. But for all the register counts, we notice important speedups when

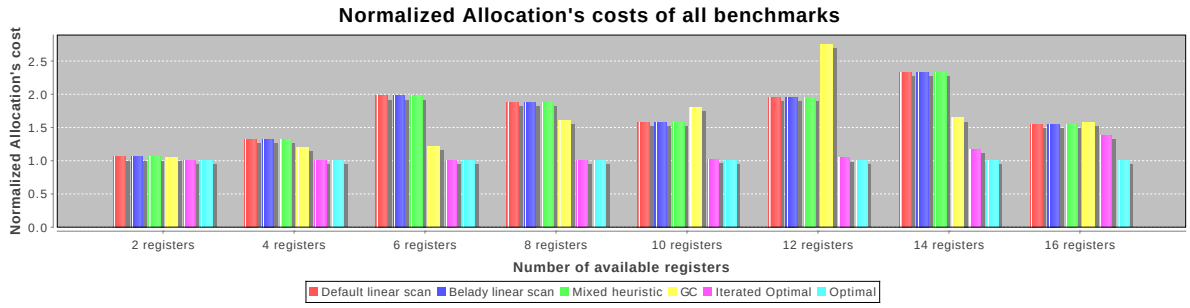

Figure 4.1: Iterated-optimal compared to other allocators for different register counts

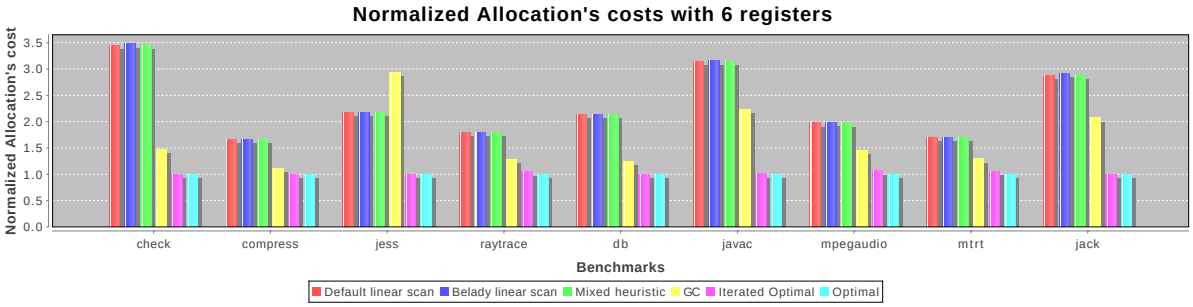

Figure 4.2: Iterative-optimal compared to other allocators when the register count is 6

using the iterated-optimal allocation algorithm compared to using the optimal one, while producing approximately the same allocation. With 6 registers we have a speedup of 8x and when we exceed 8 registers the speedups reach 100x. This shows great improvements and shows that it is possible to achieve fast allocations that are close to optimal.

Register count	Optimal (ms)	Iterated-Optimal (ms)	speedup (Iterated-optimal/Optimal)
2 registers	2810	2880	0.98
4 registers	22998	7372	3.12
6 registers	74561	8846	8.43
8 registers	381755	9768	39.08
10 registers	1194311	10477	113.99
12 registers	3231582	11120	290.61
14 registers	4147764	11688	354.87
16 registers	4879200	12281	397.3

Table 4.1: Time spent in milliseconds (ms) to solve ILP-programs

4.3 Related Work

Register allocation algorithms often rely on spilling algorithms to perform spill minimization.

In static compilation the dominant approach to register allocation is the graph coloring in which the spilling and coloring (assignment) algorithms are interleaved. During the *Simplify phase*, whenever all the remaining nodes have at least k degrees, a node needs to be marked as spilled or pushed onto the stack (optimistic coloring) and removed from the graph. A natural intuition is to choose a node that has a low spilling cost and which interferes a lot. Many graph-coloring-based allocators are based on this intuition and use the quantity $cost(v)/deg(v)$ to choose the variables to spill [Cha82]. Thus, the spilling algorithm uses a global information over the whole program that combines the interference degree and the spilling cost.

In the context of just-in-time compilation, the compilation time is part of the global execution time and (quasi-)linear complexity remains a driving force in the design of optimization algorithm. Moreover, when embedded systems are addressed, the limited memory resources is also an important issue. The linear scan which is one of the most used register allocation algorithm on JIT compilers has a worst case complexity of $\mathcal{O}(n \times k)$, where n is the number of variables in the program and k is the number of available registers on the target architecture. The original spilling heuristic used in linear scan [PS99] is based on the Belady's furthest first algorithm [Bel66]. This algorithm relies on *local information* to perform spilling: "At a point p where registers are not enough to hold all the live variables, spill the variables whose live ranges go farther in the future". May be the variable chosen for spilling will not interfere a lot in the future and will be accessed many times or may be it will only be accessed once after this point. But, since we are using a local information we cannot answer this question at the point p. Recent versions of linear scan use more elaborate algorithms which are based on variables's spill cost estimation and even sometimes use the same spilling heuristics used by graph coloring [SB07]. This gives more global information on which to rely to make spilling decisions in linear scan.

The idea of improving the spill minimization in a decoupled approach, where the allocation is decoupled from the assignment, has been explored by Proebsting and Fischer [PF92], and by Braun and Hack [BH09]. Braun and Hack generalized the Belady's furthest first algorithm, which works very well on straight-line code, to control-flow graphs. Their approach, while being applicable as a pre-spill phase in any compiler, is more adapted to a use in SSA-based register allocation. In their evaluation, they reported that with their approach they reduced the number of reload instructions by 54.5% compared to the linear scan and by 58.2% compared to the graph coloring. Like the approach of Braun and Hack, the iterated-optimal allocation algorithm is fast and can be applied as a pre-spill phase or a non-decoupled allocation for general programs and as a decoupled

allocation phase for SSA programs. However, unlike Braun and Hack, we experimentally show how the iterated-optimal allocation algorithm is close to optimal allocations.

4.4 Conclusion

We have presented the iterated-optimal allocation algorithm which is a new approach to solve the spill minimization problem. It is fast and produces allocations that are close to optimal ones. The iterated-optimal allocation algorithm is pseudo-polynomial (polynomial, when *step* and $R_{\min}$ are fixed to small numbers) on SSA programs. It can be used in a non-decoupled context for general programs, in a decoupled context for SSA programs, and as a pre-spill phase in any compiler.

Part II

Local Memory Allocation

Chapter 5

Local Memories and Allocation Techniques

5.1 Introduction

In Chapter 1, we saw that to reduce the performance gap between the processor and the memory, modern computers use a *memory hierarchy*. It corresponds to an organization of the memory into different levels based on their size and speed of access. The faster and smaller levels of memory are placed closer to the processor and the slower and larger ones are placed farther from the processor. Typically, modern computers, have a small number of registers, followed by a small amount of on-chip *static RAM* (SRAM) going from some kilobytes to many megabytes in size. The next level is the physical main memory (off-chip memory) going from hundreds of megabytes to many gigabytes of *dynamic RAM* (DRAM). Usually, in desktop computers, the on-chip SRAM is configured as a hardware cache. The hardware mechanism automatically stores frequently used instructions and data into the cache memories.

The alternative approach to manage on-chip SRAM is to configure them as software-controlled local memories, also called local memories or scratchpad memories. In such an approach the developer or the compiler must insert explicit instructions to transfer data between the local memory and the main memory. Local memories are often preferred to caches due to their better performance, their power efficiency, and their smaller area cost. Indeed, in a detailed study Banakar et al. [BSL+02] make a comparison between local memories and caches. The authors measured 18% runtime improvements in cycles for a configuration using local memory over one using a cache, when the allocation was performed with a simple knapsack-based algorithm. They also found that on average,

compared with a cache of same capacity, a local memory occupies an area 34% smaller and reduces the power consumption by about 40%. Moreover, local memory guarantee better time predictability for real time systems. Indeed, the access time of each memory object can be predicted, since the compiler or the developer exactly knows where it is located, either in the local memoryor the main memory.

Given all the above-mentioned advantages of software-controlled local memories over caches, one may ask why caches are still used in desktop computers? The reason is that with caches it is possible to guarantee the binary code portability [UDB06], while local memories do not provide such a guarantee. It is possible for the same program to be executed across different computers with different cache sizes. In contrast, a code compiled for a local memory of a given size is not portable across architectures of different local memory sizes. The binary portability is not the main issue in the embedded world because the hardware is known and software is often loaded in it within the factory and is rarely modified afterwards.

Most ARM processors have an on-chip local memory [ARM98], and more generally, it is typical for DSPs and embedded processors to have local memories [Mot98, Ins97]. More specialized processors also utilize local memories, including stream-processing architectures such as graphical processors (GPUs) and network processors [NVI08, BPMR03]. Most processor(s) may directly access the *main memory*, but few exceptions exist. The IBM Cell broadband engine's synergistic processing units (SPU) [KDH+05] which rely exclusively on Direct Memory Access (DMA) for instruction and data transfers with main memory.

To take advantage of all the potentials provided by local memories, it is essential to use them efficiently. This is the goal of an optimization problem called *local memory allocation*. The local memory allocation is defined as comprised of the following two sub-problems which are solved either together or separately:

1. the *allocation* which selects the set of variables that will reside in the local memory at each point of the program.

2. the *assignment* which finds the specified place in the local memory where a variable will reside.

Previous work on local memory allocation addressed it from different angles, targeting for both application code and data placement. All the existing allocation methods can be classified in two categories: static allocation methods and dynamic ones. Static methods place variables in the local memory only once in the beginning, and throughout the entire execution the contents of local memory remains invariant. In contrast, dynamic methods

place data/code in the local memory at a certain moment during the execution and in the main memory at a different moment depending on its access frequency. Dynamic methods reflect the dynamic behavior of the program and generally outperform static ones except when code size is extremely constrained [UDB06]. As a result, recent research mainly focuses on dynamic methods. The two next sections present respectively the main works that have been done in static and dynamic allocation strategies.

5.2 Static Allocation Methods

Early techniques for local memory allocation were mostly static. Among these static allocation methods, we mention the works of Steinke et al. [SWLM02], Sjodin and Von Platen [SvP01], and Avissar et al. [ABS02].

The method proposed by Steinke et al. analyses the application and decides which part of both data and code to place in the local memory. They formulated the problem as an integer linear programming (ILP) knapsack problem and proved an improvement between 12% and 43% in energy consumption over a cache solution.

Sjodin et al. used an ILP formulation to solve the static local memory allocation problem [SvP01]. In their work, the authors handled global variables and focused on the fact that it is very common to have different native pointers that are used to access the different kinds of memory available in embedded systems. For instance, the Intel 8051 has three native pointer types: an 8-bit pointer to access the internal memory, a 16-bit pointer to access the external memory, and a 16-bit pointer to access the code memory. They optimized the use of native pointer types to improve the execution speed and to reduce the code size.

Avissar et al. also proposed an ILP formulation to perform a static allocation over different types of memory (local memory, main memory) [ABS02]. In contrast to Sjodin et al. who only managed global variables, Avissar et al. considered also stack variables in their allocation. Moreover, they used a distributed stack technique [BLAA01] which distributes stack variables among different memory units (local memory, main memory). The results show 44% reduction in runtime when using the distributed stack strategy compared to using a unified stack. They also reported a reduction of 11% in runtime when using ILP compared to using a greedy strategy consisting of sorting the variables according to their size and then trying to allocate the variables of smallest size to the fastest memory.

Static allocation methods are however limited compared to dynamic ones because they do not handle the dynamic behaviours of the program being executed. That is, they do

not maintain the set of frequently accessed data in the local memory, at each time. This issue is addressed by dynamic methods presented in the next section.

5.3 Dynamic Allocation Methods

The first dynamic allocation methods were restricted to software-caching techniques [MFA01, HR00]. Moritz et al., manage the local memory as a cache. They substitute the tag-memory and the cache controller, present on architectures with caches, with a compiler managed tag-like data structure, address translation and tag checks. As a result, this incurs software overhead which is, in some cases, optimized by the compiler. They showed that even without any hardware support, their approach outperforms hardware caches by improving caching effectiveness. The approach of Moritz et al., like other software-caching techniques [BCZ90, Iye99, CR95], incurs significant overheads in runtime, code and data size, energy consumption, and has unpredictable execution times [UDB06]. These drawbacks led to other dynamic methods which do not emulate the functioning of caches.

Verma et al. proposed an ILP-based approach to solve the dynamic local memory allocation problem [VWM04]. In their approach, they solved separately the allocation and the assignment problems. The first step of their solution identifies the candidate variables which are composed of global variables, non-scalar local variables and frequently executed code segments. In the second step, a liveness analysis is performed. The control-flow graph [7] (CFG) edges are chosen to be the points where allocation decisions and data transfers between the local memory and the main memory will be performed. The third step optimally finds the set of allocated variables and the final step finds an address in the local memory for each allocated variable under the restriction that a variable is assigned to the same address each time it is allocated. This technique, compared with a static allocation technique, shows improvements of 34% in energy consumption and 18 % in execution time. However, this technique is limited by two practicability issues. First, for large programs the solution times can be exponential. Second, due to intellectual property issues it is rare to find an industrial compiler which integrates an ILP solver.

An interesting work that does not suffer from the above-cited problems has been introduced by Udayakumaran et al. [UDB06]. The authors propose an approach for the compiler-assisted dynamic allocation of global and stack data, and explain how to extend this approach with some minor modifications to also allocate program code. They split the program into different regions, which are defined as the code between successive program points. Each program point starts before a procedure call or before loop beginning. At

[7] A definition of CFG can be found in Chapter 1.

every program point, they keep track of timestamps, where a timestamp refers to one path beginning from the entry to a parent node of the current point. They allocate data onto the scratch-pad memory between program regions. More specifically, they allocate based on the access frequency-per-byte of a variable in a region, collected from the profile data. For each program point visited in the timestamp order, they choose the variables to place in the local memory among the ones that are going to be accessed in the next region. The allocation remains fixed within a region. In a separate phase, the method performs an assignment for each region as follows:

1. The method collects the list of free holes in the local memory. These holes are due to de-allocation of variables that are either dead or spilled.

2. It attempts to allocate the incoming variables to the available free holes. The largest variables are placed first in a best-fit fashion.

3. When, a hole of adequate size cannot be found for a variable, compaction, which is the process of moving variables in order to reduce fragmentation, is considered. It is performed if it is more profitable than spilling the variable.

4. If the compaction is not profitable, the method tries to find a cheaper variable to spill among the already assigned variables in the local memory. If such a variable cannot be found, it is spilled to off-chip memory.

This method is quite successful and in their experiments the authors reported on average improvements of 39% in runtime and 31% in energy consumption depending on the local memory size used, when comparing with an optimal static allocation.

Li et al. introduced another approach which uses an existing graph coloring technique to perform memory allocation for arrays [LGX05, LFX09]. There are three main parts in the presented approach, where in the first phase, local memory is partitioned into a pseudo-register file with interchangeable and aliased registers. This phase is followed by a live-range splitting phase, where suitable program points are generated based on the live-ranges of the arrays. The copy statements between the local memory and the off-chip memory are inserted at these points. Third, memory coloring is performed within a register allocation framework, to remove unnecessary copy statements during live-range splitting. The advantages of this work are that it uses a very mature framework (the graph coloring) and leverages live-range splitting to optimize the exploitation of the local memory. The weakness is that the partitioning of the local memory into different register class can lead to excessive fragmentation.

In a recent work, Li et al. take a more theoretical approach and proposed a way to effectively decouple the local memory allocation with optimal guarantees in the assignment phase [LNX07, LXK11]. The authors observed that in many embedded applications most arrays present a specific live range behavior. Specifically, for any two arrays, their live ranges (the list of program points where they are live[8]) are either disjoint or one of them is contained by the other one (containment property). They showed that, for the tested benchmarks, it is extremely rare to have two live ranges which interfere with one another without containment. When this happens, they extend the live range of one of the arrays to contain the other. Authors proved that the interference graph of an application with such a property is a comparability graph which is a superperfect graph and hence optimal interval-coloring for these array interference graphs is possible. Based on this observation, they rely on the maximum weighted clique to guarantee the optimal colorability of generated interference graph. When the maximum weighted clique exceeds the size of the local memory, they use heuristics to spill or split some of the live ranges until the resulting graph is optimally colorable. While this work is interesting, it is restricted to applications where most arrays satisfy the containment property.

5.4 Conclusion

In this chapter, we have seen that in the embedded world, local memories are often preferred to caches due to their better performance, their power efficiency, their smaller area cost and their better time predictability. We have presented first the static techniques to local memory allocation and then the dynamic methods which are often superior to static ones.

[8] see Chapter 1

Chapter 6

Motivation and Approach to Local Memory Allocation

Software-controlled local memories are widely used to provide fast, predictable, and power efficient access to critical data. Compilation-time local memory allocation has been connected to register allocation for at least 30 years. Indeed, in her seminal paper [Fab79], Fabri reported strong links between register allocation and local memory allocation. The author also presented fundamental results and practical insights about the interplay with loop transformations. Much of this has been ignored until very recently. As a result, the series of tremendous, fundamental and applied advances redefining design of compiler backends have also been ignored in the field of local memory allocation [AG01, BDGR06a, HGG06, BDR07c, PP08].

In this chapter we explain what has been motivating our work on local memory allocation, what is our approach to tackle the problem, and how we compare it to previous works.

6.1 Motivation

Our work on local memory allocation is motivated by recent progress in register allocation and the question of optimality in local memory allocation.

6.1.1 Decoupled Allocation

Recent progress in register allocation leverages the complexity and performance benefits of decoupling its allocation and assignment phases [AG01, BDR07c]. The allocation phase decides which variables to spill and which to assign to registers. The assignment phase

chooses which variable to assign to which register.

The allocation phase relies on the maximal *number* of simultaneously living variables, called MaxLive, a measure of *register pressure*[9]. When "*fine-grain enough*" live-range splitting is allowed, it is sufficient, for all the live ranges to be allocated, that MaxLive is less or equal to the number of available registers to guarantee that the forthcoming assignment phase can be done without further spill. In many cases, assignment can even be achieved in linear time [BDR07c]. If at some program point the pressure exceeds the number of available registers, needs to be reduced through spilling.

This decoupled approach permits to focus on the hard problem, namely the spilling decisions. It also improves the understanding of the interplay between live-range splitting and the expressiveness and complexity of register allocation. This is best illustrated by the success of SSA-based allocation [BDGR06a, HGG06, BDR07c, BDR07a].

The intuition for decoupled register allocation derives from the observation that live-range splitting is almost always profitable if it allows to reduce the number of register spills, even at the cost of extra register moves. The decoupled approach focuses on spill minimization only, pushing the minimization of register moves to a latter register coalescing phase [AG01, BDR08]. Here again, SSA-based techniques have won the game, collapsing the register coalescing with the hard problem of getting out of SSA [HGG06, BDdD+09], as one of the last backend compiler passes.

The domain of local memory allocation tells a very different story. Some heuristics exist [UDB06, KRI+01, LFX09] but little is known about the optimization problem, its complexity and the interplay with other optimizations. The burning hot question is of course: does the decoupled approach hold for the local memory allocation problem? Surprisingly, the state-of-the-art of local memory allocation completely ignores all the advances in register allocation. When focusing on arrays, the similarity between register and local memory allocation is obvious nonetheless:

Local memory allocation. Deciding which array blocks to spill to main memory and which array blocks to allocate to the local memory. Spilling is typically supported by DMA units.

Local memory assignment. Deciding at which local memory offset to assign which allocated array block. When reconciling the offsets across control-flow regions or over multiple incoming paths, there is no unique equivalent of register moves:

- most papers assume a local copy operation with a much lower cost than loading or storing to main memory;

[9] Not the sum of array block sizes.

- one may also address each local-memory-allocated array block with its own dedicated pointer; this increases register pressure but brings down the cost of offset reconciliation to a plain pointer copy.

In the context of local memory management, the maximum size of simultaneously living arrays [10], called MaxSize, gives a measure of *local-memory pressure*. Again, like for register allocation, live-range splitting helps to reduce the local-memory pressure. Since arrays are frequently accessed inside loops, local memory allocation algorithms often split arrays at loop-entry points , we call these points: decision points. Decision points can also be chosen in a finer manner, after loops or before array accesses. Local memory pressure can also be reduced by loop-transformations like strip-mining, and tiling, which reduce the portion of accessed arrays. Moreover local memory offset reconciliation can be implemented with pointer copies. With such an assumption, the array move/spill ratio becomes negligible w.r.t. the already low register move/spill ratio. For all these reasons, the study of a decoupled approach in the local memory allocation context seems very appealing.

6.1.2 Example

```
// Nested within outer loops
for (i=0; i<N; i++)
   for (j=0; j<N; j++)
    C[i][j] = /* ... */;

F[0][0]=1; F[0][1]=2; F[0][2]=1;
F[1][0]=2; F[1][1]=4; F[1][2]=2;
F[2][0]=1; F[2][1]=2; F[2][2]=1;
```

Figure 6.1: Example: `Edge-Detect`

Figure 6.1 shows a code fragment from the UTDSP benchmarks [Lee98]. This fragment has been slightly simplified for the sake of the exposition, preserving the original array data flow. Generally, arrays are frequently accessed inside loops. As a result, local memory allocation algorithms often (re)consider allocation decisions at loop entry points; a form of *live-range splitting*. On this code fragment, if live-range splitting is only considered at loop entry points, accesses to `C` will artificially coincide with accesses to `F`. Zooming to the grain of individual program statements, the live ranges of these two arrays are in fact disjoint. Finer grain decision points would be beneficial. It is interesting to take allocation

[10] Not the number of simultaneously living arrays.

```
for (i=0; i<N; i++)
  // Outer strip-mined loop
  for (jj=0; jj<N+B-1; jj+=s)
    // Inner strip-mined loop
    for (j=jj; j<N && j<jj+s; j++)
      C[i][j] = /* ... */;

F[0][0]=1; /* ... */ F[2][2]=1;
```

Figure 6.2: Homogeneous blocks

```
for (i=0; i<N; i++)
  // Outer strip-mined loop
  for (jj=0; jj<N+B-1; jj+=s)
    STORE(C[i][jj..min(jj+B-1,N-1)]);

STORE(F[0..2][0..2]);
```

Figure 6.3: Abstract model

decisions for each array at the points where it is going to be frequently accessed (if its usage justifies memory transfers). Of course, the cost of (un)loading whole array blocks to the local memory is too high to authorize live-range splitting at every instruction (a.k.a. load/store optimization [AG01, PP08]). More advanced strategies would integrate loop transformations such as loop distribution and strip-mining [Wol95].

6.2 Our Approach to the Problem

The previous example shows the importance of the selection of decision points (live-ranges splitting) to take advantage of the dynamic behavior of the program for local memory allocation.

In the following, *main memory* refers to the main — typically off-chip DRAM — memory resources; *local memory* refers to — typically on-chip SRAM — low-latency, high-bandwidth memories to exploit temporal locality and hide the cost of accessing the main memory.

We will only consider *single-threaded* code running on a *single local memory*. To extend our results to single-threaded code running on multiple local memories with different characteristics, the gap is expected to be very narrow, analogous to register allocation on multiple register classes and register files.

Having set the context of the optimization problem, we present in the following the preliminary analyses and transformations we assume already performed and we present three live range splitting schemes that can be adopted to tackle the problem.

6.2.1 Preliminary Analyses and Transformations

We make several assumptions while formulating the local memory allocation and assignment problem. We currently restrict our approach to uniform array accesses only.

Non-uniform accesses may also be exploited through the use of symbolic expressions, approximations or pre-pass loop and data-layout transformations [Kan01, DLLK04, KAP97]. Those techniques are complementary to our work, and we consider loop nests that have been previously tiled for locality, together with data layout compaction and uniformization. As a rule of thumb, those transformations should favor the emergence of "homogeneous" array blocks. Here, "homogeneous" means that for a given array `A`, the loop transformations must strive to strip-mine the computation such that inner loops traverse a fixed-size block of `A` in a dense manner. Most numerical and signal-processing codes exhibit a vast majority of dense, uniform — sum of loop iterator plus a constant — access patterns. In such cases, homogenization is trivial: it is sufficient to set a constant strip-mining factor for all loops operating over a given array `A` [UDB06]. The last step is to abstract the transformed code, isolating array block operations into atomic regions. Considering the motivating example, these two steps are illustrated on Figures 6.2 and 6.3; s is the homogeneous block size for array `C`.

This abstraction is sufficient to collect profile information on the homogenized version of the loop nest, to solve the allocation and assignment problems. When generating the code for a real target, one needs to insert back the array block load, store, and pointer moves. Again, standard techniques exist to handle this, at least when array accesses are uniform [UDB06, KRI+01].

On the abstracted program, array blocks play the role of scalar variables in a register allocation problem, with the exception of cost modeling. We keep track of the access frequency of each execution point to capture the number of accesses for each array block.

To benefit from its algorithmic properties, we extend the SSA form to operate on array blocks. This extension differs from Array SSA proposals [KS98, RHAR06], in that it does not attempt to model the data flow of individual array elements. In this form, array blocks are fully renamed, and name conflicts at control-flow points are handled with Φ functions following the rules of strict SSA form.

From array blocks, one need to extract *live ranges* which will be the subject of the allocation and assignment decisions. Live ranges generalize live intervals in basic blocks to arbitrary control flow; in SSA form, extraction of live ranges can be done in linear time [HGG06]. Live ranges and detailed profiling of the application is then used to generate the frequencies for dynamic allocation. Using these live ranges, for each decision point (as will be explained in more detail in Chapter 7) and for each array block alive at this point we keep track of the access frequencies. From these access frequencies and from the size of each array block we compute an estimated of the cost of accessing an array block in the local memory or in main memory. The cost of accessing array blocks in main

memory depends on the access pattern of the considered architecture. On systems that rely exclusively on DMA, for data transfers, accessing a single element of an array block is as expensive as accessing the whole array block. In contrast, on systems where the main memory can be accessed directly, accessing a single element is much cheaper than accessing the whole array block.

Note that, moving array blocks within the local memory at every decision point may add severe overheads, limiting the effective benefits of live-range splitting. To avoid such penalties, we only move the pointers of these array blocks rather than moving the chunk of data.

In fact, this low-cost solution may seem so appealing to the reader that she may wonder why it was not considered in the state-of-the-art techniques. The main reason is of course that a decoupled allocation/assignment scheme is necessary to make sure that *all array block moves* induced by the assignment phase are associated with *offset reconciliation across different control-flow paths*. When resorting to a unified allocation-and-assignment algorithm with live-range splitting [UDB06], there is no such guarantee. Some moves are associated with real needs for array block displacement. Another reason is the increased pressure on the registers: we consider that a register spill is a lot cheaper than local memory block-copying, and even if repeated register spills occur in an inner loop (which is very unlikely on RISC or VLIW processors), it will not be more expensive than an array block displacement; further experimental analysis and tradeoffs should be explored in the future.

Following Udayakumaran et al. [UB03], we extend every basic block with a final program point. This extra point will be used to capture any live range load or eviction at the end of the basic block, isolating the formalization of this decision from the reconciliation of the decisions associated with incoming/outgoing control-flow arcs. Thanks to this extra point, the only cross-basic-block equations will be reconciliation equations.

6.2.2 Allocation Schemes

Live-range splitting can be implemented at different decision points depending on a customizable grain/aggressiveness. One objective could be to optimize local memory usage at every instruction execution, whereas another one could be to statically allocate the arrays for the whole execution of the program. This is analogous to the options in the register allocation problem, where the compiler designer needs to trade latency minimization for complexity. This tradeoff can be instantiated into three typical schemes:

Scheme 1. One could make decisions at fine granularity points, where a point indicates

an — abstract array block — instruction. This will exploit the array allocation and assignment at instruction granularity, providing a full latency minimization. However, modeling this scheme as an ILP-program may incur excessive complexity, due to the number of decision points.

Scheme 2. The second approach is similar to the SSA-based register allocation approach, where one may only take allocation decisions at points where a live range becomes alive.

Scheme 3. The third approach is to make an allocation decision per array block, without any splitting. This approach is called *static* in the context of LM management, and corresponds to the spill-everywhere register allocation problem. In this case, there are no program/decision points, leading to a much simpler optimization problem. However, the connection between MaxSize and colorability is lost, and the execution latency will be higher compared to the previous schemes.

6.3 Related Work

While previous studies addressed local memory allocation from different angles, targeting both code and data, we are especially interested in data allocation [KRI+01, IBMD07]. We target dynamic methods which are superior to static ones except when code size is extremely constrained [UDB06].

We elaborate on three recent series of results targeting stack and global array allocation in local memories, embracing the analogies with register allocation. The first approach [LGX05, LFX09], uses an existing graph coloring technique to perform memory allocation for arrays. The second approach [UB03, UDB06, ABS02] allocates data onto the scratch-pad memory between program regions separated by specific program points. The closest work to ours is the third approach [LXK11], where authors observed that in many embedded applications it is extremely rare to have two live ranges which does not respect the containment property. While this work is interesting, it is restricted to applications where most arrays satisfy the containment property. Compared to these three approaches, our work leverages the decoupled allocation/assignment approach, allowing scalable and more effective algorithms. Moreover, it offers much more flexibility in terms of integration of architecture constraints and performance models. We are able to analyze the tradeoffs involving live range splitting, like SSA-based techniques, as well as the effects of loop transformations.

6.4 Conclusion

We presented in this chapter the reasons that motivate our work on local memory allocation. We then presented the assumptions we made and the approach we chose to solve this problem and we finally show how our work is related to previous ones.

Chapter 7

Experimental Validation

In this chapter we intend to experimentally validate our intuition of a decoupled approach to the local memory allocation problem. First, we express the allocation of array blocks as an ILP-program (the decoupled ILP-program), which minimizes the cost of access-latency (execution time). A solution to this ILP-program can then be used in a subsequent assignment stage, to set the offsets of the allocated array blocks into the local memory. Because of the fragmentation of the local memory into uneven regions, this is not sufficient to guarantee that the assignment stage will succeed without further spilling. This is a major difference with the classical register assignment problem. A solution consists in adding fragmentation-avoidance constraints to the ILP-program (the integrated ILP-program). We then compare, for each of the used benchmarks, the cost returned by the decoupled ILP-program to the one returned by the integrated ILP-program. If the costs are equal, we can envisage a theoretical study of a decoupled approach to the local memory allocation problem.

7.1 Allocation

This section addresses the execution time minimization problem using integer linear programming. All the variables in the ILP-program are associated with decision points within a given program. These points represent live range splitting points where allocation decisions are taken and memory transfer instructions may be inserted. There are various outcomes for each live range, each one being characterized by the ILP-program: (i) evict the live range from the local memory, (ii) load the live range from main memory to the local memory, (iii) keep the live range in the local memory, (iv) leave the live range in the main memory. In this regard, ILP variables closely match those introduced by Appel and George [GA96].

Let us formalize the ILP-program. The size of the local memory is denoted as S; the size of live range v is denoted as s_v. The access frequency of a live range is defined as the frequency per byte of its corresponding array block; the access frequency per byte of live range v at point p is denoted as $fpb_{v,p}$.

For each point p and for each live range v alive at p, we define the following variables:

- $r_{v,p}$ is a $\{0,1\}$ variable, set to one when v is allocated to the local memory at point p. Otherwise, it is set to zero. Since $r_{v,p}$ is a $\{0,1\}$ variable, $1 - r_{v,p}$ captures the live ranges residing in main memory.

- $l_{v,p}$ is a $\{0,1\}$ variable, set to one when v is loaded into the local memory at point p. Otherwise, it is set to zero. This variable is related to $r_{v,p}$ and $r_{v,p-1}$: if v is not in the local memory at point $p-1$, i.e., if $r_{v,p-1} = 0$, and if it is in the local memory at point p ($r_{v,p} = 1$), then v is loaded at point p.

- $e_{v,p}$ is a $\{0,1\}$ variable, set to one when v is evicted from the local memory at point p. Otherwise, it is set to zero.

- $d_{v,p}$ is a $\{0,1\}$ variable, set to one when v is dirty, i.e., when v is updated and needs to be written back to the main memory in case of an eviction. Otherwise, it is set to zero.

- $u_{v,p}$ is a $\{0,1\}$ variable, set to one when v needs to be updated. That is, if v is evicted and dirty, it will be written back to the main memory. More specifically, it is the logical *AND* of $e_{v,p}$(evict) and $d_{v,p}$(dirty).

In a basic block, we capture the values of $l_{v,p}$ and $e_{v,p}$ using $r_{v,p}$ at successive points $p-1$ and p:

$$l_{v,p} \geq r_{v,p} - r_{v,p-1}. \tag{7.1}$$

This constraint makes sure that live range v is loaded if it was not in local memory at point $p-1$ whereas it is in local memory at point p. At points where there is an explicit assignment to v, we set $l_{v,p}$ to 0 since the array block is being modified and a local copy would be useless. Notice that we are dealing with partial updates of array blocks, and assignment points are not necessary kill points for live ranges.

Similar to a load, a live range v is evicted if it is in the local memory at point $p-1$ but it is not anymore in the local memory at point p:

$$e_{v,p} \geq r_{v,p-1} - r_{v,p}. \tag{7.2}$$

However, this is not enough to characterize every eviction since we only set a $\geq$ inequality and we do not have any term that (in)directly minimizes eviction in the objective function. Indeed, eviction can be free, such as when the data is being evicted but it has not changed. Setting an equality constraint would not be correct either. To prevent this, we add the following constraints to provide additional upper-bounds on $e_{v,p}$:

$$e_{v,p} \leq r_{v,p-1}. \tag{7.3}$$

If v is in the local memory at point p or if it is not in local memory at point $p-1$, then $e_{v,p}$ will be forced to 0. However, if we do not add these constraints, it could be either 0 or 1 depending on the update.

Every assignment statement will mark the target live range as dirty. We formalize this as:

$$\forall p_i, d_{v,p_i} = 1, \text{ where variable } v \text{ is modified.} \tag{7.4}$$

Moreover, if v in the local memory is dirty, it should be continuously dirty until it is written back to the main memory. We enforce the following constraint at every point except those where live range v is modified. For those cases, Equation 7.4 will be used:

$$d_{v,p} \geq d_{v,p-1} - u_{v,p-1}, \text{ where variable } v \text{ is not modified.} \tag{7.5}$$

This constraint makes sure that if v is dirty at point $p-1$ and it is still in the local memory at point p, then it should be dirty at point p as well. This, in a sense, operates as dirty bit in a cache coherence protocol.

Also, $d_{v,p}$ will be forced to be set as 0, if v is not in the local memory at point $p-1$. This will ensure that, if v is loaded at point p it will not be dirty:

$$d_{v,p} \leq r_{v,p-1}, \text{ where variable } v \text{ is not modified.} \tag{7.6}$$

Deriving from dirty variables, update variables $u_{v,p}$ are set to 1 if v is evicted at point p and is marked as dirty. We formally express this boolean *AND* as follows:

$$u_{v,p} \geq e_{v,p} + d_{v,p} - 1 \quad \wedge \quad u_{v,p} \leq e_{v,p} \quad \wedge \quad u_{v,p} \leq d_{v,p}. \tag{7.7}$$

We also need to make sure that the available local memory space is not exceeded. At every point p we add the following constraint:

$$\forall p, \sum_{v} r_{v,p} \times s_v \leq S. \tag{7.8}$$

Finally, we need to make sure that the local memory remains in the same state as we iterate through the basic blocks of the application CFG. To ensure this, we add entry and exit points to basic blocks. This way, we check the states at every merge/split point whether they are in the local memory or not. More specifically, for such pairs of points p and p':

$$\forall v, r_{v,p} = r_{v,p'}. \tag{7.9}$$

The objective function to be minimized is the following:

$$\sum_{v,p} r_{v,p} \times lr_{v,p} + l_{v,p} \times ll_{v,p} + u_{v,p} \times le_{v,p} + (1 - r_{v,p}) \times lm_{v,p}.$$

It models the cost of array block load, eviction, access in the local memory, and access in the main memory. Since we follow a decoupled allocation/assignment approach, it does not account for reconciliation costs, implemented as negligible-cost register moves. This function depends on the following additional notations:

- $lr_{v,p}$ is the sum of all latencies incurred by an access to v between p and $p+1$, the following point, where v was in the local memory at p:

$$lr_{v,p} = latency\_LM \times fpb_{v,p} \times s_v.$$

- $ll_{v,p}$ is the sum of all latencies incurred by an access to v between p and $p+1$, where v was in the main memory before p, and in the local memory after p:

$$ll_{v,p} = latency\_reload(s_v) \times point\_frequency_p.$$

- $le_{v,p}$ is the sum of all latencies incurred by an access to v between p and $p+1$, where v was in the local memory before p and in the main memory after p:

$$le_{v,p} = s_v \times latency\_spill(s_v) \times point\_frequency_p.$$

- $lm_{v,p}$ is the sum of all latencies incurred by an access to v between p and $p+1$, where v was in the main memory before, and after p:

$$lm_{v,p} = latency\_MM \times fpb_{v,p} \times s_v.$$

Each update variable acts as a switch to turn on or off the cost of the eviction, depending on the dirtiness of the corresponding live range.

This linear program is flexible w.r.t. the tradeoffs between live-range splitting and complexity, and adapts to the three proposed schemes, mentioned in Chapter 6. Of course, the p subscript will disappear for Scheme 3. For Scheme 1, $r_{v,p}$, $l_{v,p}$ and $e_{v,p}$ will be defined for all program points. They will be defined at live range beginnings for Scheme 2.

7.2 Assignment

Due to fragmentation, the colorability result, when MaxSize is below the local memory size (for the first three schemes), is not sufficient to guarantee an atomic allocation of array blocks. In practice, it is known that classical fragmentation-avoidance heuristics perform very well, even on rather constrained problems [UDB06]. Those heuristics derive from dynamic, heap memory allocation algorithms like the buddy system. This pragmatic approach leads to a very simple, greedy assignment algorithm like Belady's furthest first method [BDR07c, Bel66]. Of course, when implementing Scheme 1, one must be careful to always *assign the same offset* to a given (local-memory-allocated) live range *within a given basic block*. Since live range splitting at SSA definition points is sufficient to guarantee the colorability, this restriction does not induce spurious spills.

If fragmentation-induced spills must be avoided at all costs, we need to define additional $\{0,1\}$ variables to handle the assignment directly. Of course, this breaks the decoupling of the allocation and assignment phases. The immediate, expected impact is a dramatic increase in algorithmic complexity.

We define a pair of variables for each point p and live range v alive at p:

- $o_{v,p}$ a $\{1, S\}$ variable, which represents the offset of the variable v in the local memory at p; this variable is valid only if $r_{v,p} = 1$, that is, if v is not in the local memory, we set this variable to 0.

- $m_{v,p}$ a $\{0,1\}$ variable, which represents the displacement of variable v in the local memory at p; this is measured by the offsets at point p ($o_{v,p}$) and at point $p-1$ ($o_{v,p-1}$).

To account for the relative placement of live ranges in the local memory, we also need a very large set of variables $b_{v,v',p}$ in $\{0, S-1\}$. Each variable indicates whether v comes before v' in the local memory at point p (and how far it does).

In addition to these new variables, we introduce additional constraints and a modified objective function which captures both the assignment and allocation costs. First, we

modify the previous local memory size constraint with multiple constraints operating on the variable offsets. Specifically, we change (7.1) into:

$$o_{v,p} + s_v \leq S + 1 + (1 - r_{v,p}) \times S. \tag{7.10}$$

This constraint ascertains that at each point, the size of allocated variables does not exceed the size of the local memory. Moreover, we set the offset of any variable, $o_{v,p}$, to 0 if that variable is not within the local memory. Note that, $r_{v,p}$ will be equal to 0 for such variables forcing $o_{v,p}$ to 0:

$$o_{v,p} \leq r_{v,p} \times S. \tag{7.11}$$

In addition to the upper bound on the offset, we ensure the lower offset bound of a local memory resident variable. If a live range is in local memory, offset will be set to be greater than or equal to 1, otherwise it will be greater than or equal to 0:

$$o_{v,p} \geq r_{v,p}. \tag{7.12}$$

As indicated above, we use $b_{v,v',p}$ to express whether v comes before v' in the local memory, through a comparison between the offsets of v and v'. To characterize this variable, we add the following constraints:

$$b_{v,v',p} \leq r_{v,p} \quad \wedge \quad b_{v,v',p} \leq r_{v',p} \tag{7.13}$$

These constraints make sure that variable $b_{v,v',p}$ is only valid for local-memory-allocated live ranges: if the variable is not residing in local memory it will have an offset of 0.

$$b_{v,v',p} + b_{v',v,p} \leq 1. \tag{7.14}$$

On the other hand, to force the ordering between live ranges, we set an upper bound of 1 to the above sum. Moreover, if both v and v' are in the local memory, then either $b_{v,v',p}$ or $b_{v',v,p}$ must be set to 1 to capture the relative position of the two live ranges. This is ensured by an *AND* operation between $r_{v,p}$ and $r_{v',p}$ which will set one of $b_{v,v',p}$ and $b_{v',v,p}$ to 1 through a $\geq$ inequality:

$$b_{v,v',p} + b_{v',v,p} \geq r_{v,p} + r_{v',p} - 1. \tag{7.15}$$

Next, for all v and v', offsets should be assigned such that array blocks are not intersecting.

More specifically,

$$o_{v',p} - o_{v,p} \geq s_v \times b_{v,v',p} + (b_{v,v',p} - 1) \times S. \tag{7.16}$$

As mentioned earlier, we capture the array block displacements through the offsets. Specifically, we take the difference of offsets at point p ($o_{v,p}$) and at point $p-1$ ($o_{v,p-1}$):

$$m_{v,p} \times S \geq |o_{v,p} - o_{v,p-1}| + (r_{v,p} - 1) \times S + (r_{v,p-1} - 1) \times S. \tag{7.17}$$

Note that, in the above constraint, we force $m_{v,p}$ variable to the offset distance ($|o_{v,p} - o_{v,p-1}|$) only if v is in the local memory at both points $p-1$ and p. This is enforced through the terms $(r_{v,p} - 1) \times S$ and $(r_{v,p-1} - 1) \times S$. If variable v is not in the local memory at any of these points, then it will add $-S$ to the right part of the constraint which will set $m_{v,p}$ to 0 since the offset distance cannot be larger than the local memory size S and the objective function is *minimized* in the optimization.

We modify (7.9) which ascertains that local memory remains in the same state throughout the flow between basic blocks. We now need to keep the offsets same in order to preserve the same state. That is, we check the states at every merge/split point via the variable offsets. More specifically, for such points p and p':

$$\forall v, o_{v,p} = o_{v,p'}. \tag{7.18}$$

The objective function also needs to be updated, to include movements between different execution points. This is the purpose of the last term in the following objective function:

$$\sum_{v,p} r_{v,p} \times lr_{v,p} + l_{v,p} \times ll_{v,p} + u_{v,p} \times le_{v,p} + (1 - r_{v,p}) \times lm_{v,p} + \sum_{v,p} m_{v,p} \times moving\_cost(v,p), \tag{7.19}$$

where $moving\_cost(v,p)$ is the function giving the latency incurred by the displacement of v at point p within the local memory:

$$moving\_cost(v,p) = latency\_move(s_v) \times point\_frequency(p).$$

7.3 Experimental Results

Experiments on real hardware are out of the scope of this work. This section aims at validating the fundamental hypotheses and the relevance of the decoupled approach for

Benchmark	Brief description	Suite	Data size	arrays /blocks
`Edge-Detect`	Edge detection in an image	[Lee98]	196644	4/385
`D-FFT`	256-point complex FFT	[Lee98]	2032	7/7
`Bmcm`	Water molecular dynamics	[ea88]	125240	10/310
`MxM`	Matrix multiplication	n.a.	120000	3/300

Table 7.1: Application codes.

Constant	Latency
$latency\_LM$	8
$latency\_MM$	128
$latency\_move(s_v)$	$8 + 2s_v$
$latency\_spill(s_v)$	$128 + 4s_v$
$latency\_reload(s_v)$	$128 + 4s_v$

Table 7.2: Model parameters.

local memory allocation.

7.3.1 Setup

We validate our approach on three array-intensive benchmarks from the UTDSP [Lee98] and Perfect Club [ea88] suites, and on matrix multiplication, see Table 7.1. Table 7.2 lists the model parameters. We use Scheme 2 for all experiments, a good tradeoff between expressiveness and complexity.

7.3.2 Results

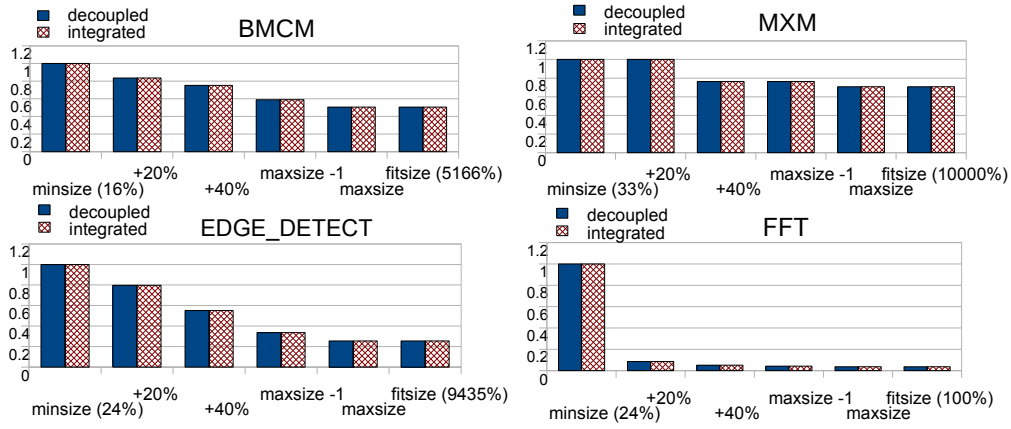

Figure 7.1: Experimental results for decoupled and integrated approaches.

Figure 7.1 shows the cost model results for each benchmark. The left bar (dark) refers to the cost achieved by the allocation phase alone, selecting which live ranges will reside in the local memory at which point. The one on the right (light) refers to the cost achieved by the integrated allocation and assignment, a much more complex and inflexible optimization taking offsets and fragmentation into account. The second bar can only be identical or higher than the first, since more constraints are taken into account.

We consider different local memory sizes: *minimal_size* corresponds to the the biggest array block in the program; *fitting_size* is the total size of all array blocks; *maxsize_size* is the maximum of the total size of array blocks simultaneously alive at any given point; the minimal size plus 20% of the *maxsize_size*; and the minimal size plus 40% of the *maxsize_size*. We also state *minimal_size* and *fitting_size* in percentage of *maxsize_size*. The "size" qualifier has been omitted from the figure.

These experiments validate the two main insights motivating our approach:

1. For all benchmarks, and all local memory sizes, the decoupled and integrated algorithms compute the same cost. This is a strong confirmation that decoupled local memory allocation behaves as good as decoupled register allocation, despite the fragmentation constraints.

2. For all benchmarks, the cost is optimal — same as for *fitting_size* — as soon as the local memory size reaches *maxsize_size* while the cost immediately increases for smaller local memory sizes (*maxsize_size-1*). This confirms that MaxSize is the relevant criterion.

7.4 Conclusion

This chapter opens a window on exciting research and applications about local memory allocation. Despite strong progress in the recent years, the state-of-the-art ignores the tremendous advances in decoupled and SSA-based register allocation. We set up a new bridge between the two optimization problems. Our experiments validate the decoupling of the allocation and assignment stages in the context of local memory allocation: after an optimal allocation phase relying on a generic and scalable integer linear program, we demonstrate a total absence of fragmentation-induced spills, during the assignment-phase.

Chapter 8

Decoupled Local Memory Allocation for Linearized Programs

In Chapter 7, we experimentally validated that a decoupled approach could be adopted to solve the local memory allocation problem. We consider here such a decoupled approach, but with a more theoretical point of view. We show that the local memory allocation for linearized programs, where the live ranges of variables or arrays are represented as intervals, is equivalent to a weighted interval-graph coloring problem that we call the submarine-building problem. The submarine-building problem differs slightly from the classical ship-building problem [Gol04] by allowing a color interval to "wrap around". We show that the submarine-building problem is NP-complete, while it is solvable in linear time for weighted proper interval graphs.

8.1 Weighted Graph Coloring and Local Memory Allocation

This section sets the terminology and definitions used in the rest of the chapter.

8.1.1 Weighted Graphs

A graph $G = (V, E)$ consists of two sets, V the set of vertices, and E the set of edges. Every edge (v_1, v_2) of E has two end points $v_1 \in V$ and $v_2 \in V$. We consider undirected graphs only, i.e., we do not differentiate between the edges (v_1, v_2) and (v_2, v_1).

A graph G is called an interval graph if its vertices can be put into one-to-one correspondence with a set of intervals I of a linearly ordered set such that two vertices are

connected by an edge of G if and only if their corresponding intervals have a nonempty intersection.

Assuming each vertex v of $G = (V, E)$ is associated with a non-negative number $w(v)$, the weight of a subset $S \subset V$ is expressed as:

$$w(S) = \sum_{v \in S} w(v).$$

The graph G associated with the function w is called a weighted graph and denoted G_w. Moreover, G_w is a weighted interval graph if G is an interval graph.

An interval coloring of a weighted graph G_w is a function I mapping each vertex $v \in V$ onto a (topologically) open interval I_v of $w(v) + 1$ consecutive integers of the real line, such that adjacent vertices are mapped to disjoint intervals; that is, $(v_1, v_2) \in E$ implies $I_{v_1} \bigcap I_{v_2} = \emptyset$. We say that I is a k-coloring of G_w if $I_v \in \{0, \ldots, k\}$, $\forall v \in V$. The chromatic number $\chi(G_w)$ is the smallest k for which we can find a k-coloring of G_w.

Figure 8.1 shows two colorings of a weighted graph shown in Figure 8.1(a). The first coloring given in 8.1(b) is a 6-coloring of the weighted graph and the second coloring shown in 8.1(c) shows a 5-coloring of the weighted graph. The chromatic number of this graph is 5.

8.1.2 Linearized Programs

Given an intermediate representation of an arbitrary program on which live range splitting has already been applied, the intermediate representation pseudo-instructions can be numbered according to some order. We define a *linearized program* as a program for which such kind of numbering [PS99] has been performed and for each variable v in this program, we represent its live range as the live interval $[i, j[$, i being the number of the first instruction where v is first defined and j being the number of the instruction where v is last used. There can be some some pseudo-instructions between i and j where v is not live, but with a successful live range splitting this problem can become marginal. In the context of just-in-time compilation where compilation time is critical, linearizing programs can pay off because it is fast to linearize a program and potentially the produced code could be of relative quality [SB07].

The linearization flattens the program's control-flow graph and the generated live intervals form an interval graph.

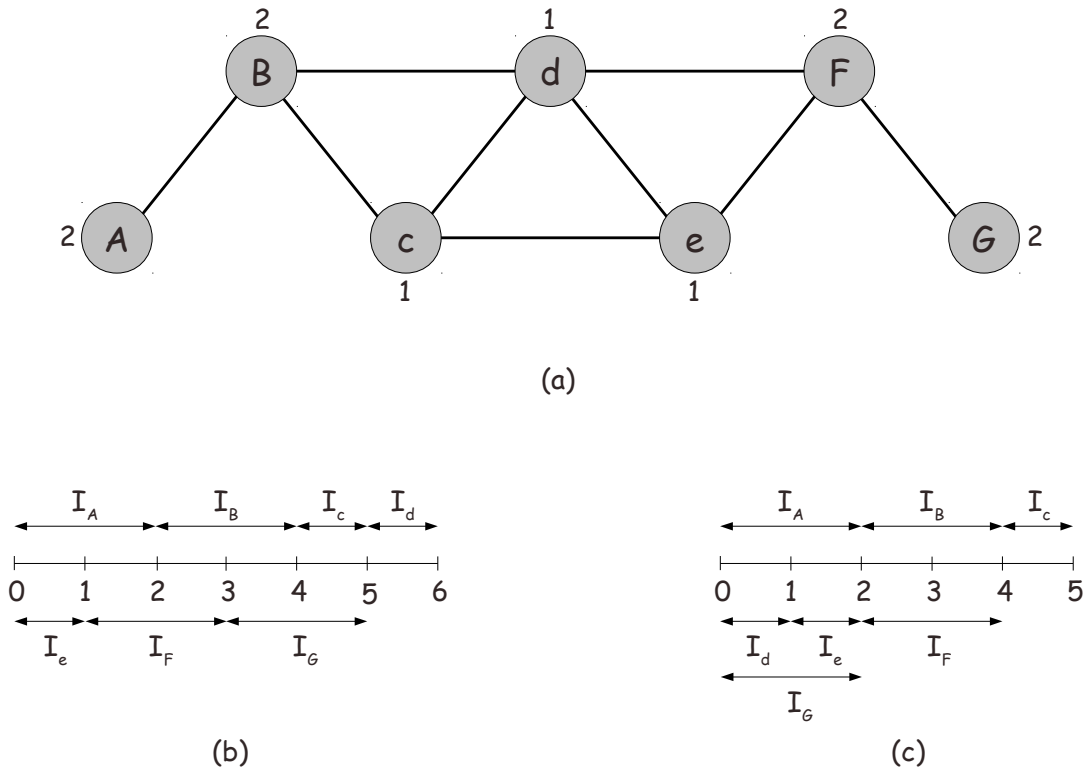

Figure 8.1: Two colorings of a weighted graph

8.1.3 Two Equivalent Classes

From Local Memories to Weighted Interval Graphs. From a linearized program we construct a corresponding weighted graph called *interference graph*. For each variable in this program, we create a vertex and associate the size of the variable to this vertex. We create an edge between two vertices if there is a point in the program where the two variables are simultaneously live. Thus, an edge connects a pair of vertices if and only if the variables are simultaneously alive. The constructed weighted graph is a weighted interval graph because each vertex corresponds to an interval defined by the definition point and the end point of the variable.

From Weighted Interval Graphs to Local Memories. We use a method similar to the one presented by Lee et al. [LPP08] to show that, for any weighted interval graph, we can find a corresponding linearized program.

Chen [Che92] and Saha [SPP07] et al. have shown how to convert an interval graph with q intervals to an isomorphic *program like* interval graph in $\mathcal{O}(q \log q)$ time. An

interval graph is program-like if the intervals representing the vertices of the graph have start points and end points that are all different, and the start points and end points of the intervals form a set $\{1, \ldots, 2q\}$, where q is the number of intervals.

From a program-like weighted interval graph G_w, we construct in $\mathcal{O}(q)$ time the following straight-line program (with pseudo-C syntax) which consists of a set of $2q$ statements:

$$\forall i \in \{1, \ldots, 2q\} \begin{cases} \texttt{type}_I \ \texttt{v}_I \ \texttt{=} \ \cdots & \text{where } \texttt{sizeof(type}_I\texttt{)} = w(I), \\ & \quad \text{if the interval } I \text{ of weight } w(I) \text{ begins at } i \\ \cdots \ \texttt{=} \ \texttt{v}_I & \text{if the interval } I \text{ ends at } i \end{cases}$$

8.2 Weighted Graph Coloring

Thirty years ago, in her seminal paper [Fab79], Fabri already envisaged to model the so-called problem of "automatic storage allocation" as a weighted graph coloring problem. She mentions the investigation of special subclasses of weighted graphs that are likely to occur. We construct a weighted graph G_w from a given linearized program. Finding an allocation for variables of the linearized program within a local memory of size k corresponds to finding a k-coloring of G_w.

This section introduces the *ship-building* problem which is related to weighted interval graph coloring. It also defines a new variant, called the *submarine-building* problem, very well suited to the local memory allocation problems on modern processors, and exhibiting interesting complexity results and potentially good approximation heuristics.

8.2.1 The Ship-Building Problem

We report here the Ship-Building Problem as presented in the book of Golumbic [Gol04].

In certain shipyards the sections of a ship are constructed on a dry dock, called the welding plane, according to a rigid time schedule. Each section s requires a certain width $w(s)$ on the dock during construction. Can the sections be assigned space on a welding plane of total width k so that no spot is reserved for two sections at the same time?

Let the sections be represented by the vertices of a graph G and connect two vertices if their corresponding sections have intersecting time intervals. Thus G_w is a weighted interval graph. An interval coloring of G_w will provide the assignment of the sections to spaces, of appropriate size, on the welding plane. This assignment will be consistent with the intersecting time restrictions. The reader must be careful to distinguish between the time intervals which produced the edges of G_w and the color intervals which provide a solution to the assignment of space on the dock.

A weighted graph built from a linearized program associated with a number k (corresponding to the size of the local memory) is an instance of the ship-building problem. For a graph G_w and a number k, we call $ship(G_w, k)$ an instance of the ship-building problem.

Determining whether $\chi(G_w) \leq k$ is an NP-complete problem [Gol04, LPP08][11], even if G is an interval graph and the weight function w is restricted to the values 1 and 2. It follows that the ship-building problem is also NP-complete.

8.2.2 The Submarine-Building Problem

Since the local memory size is generally power-of-two, it is common to mask the addresses (in software or hardware) to let loads and stores wrap around to the local memory transparently. The submarine-building problem is a new variant of the ship-building problem. Like in the ship-building problem, a vertex must occupy a contiguous color-interval, but a circular allocation scheme can be adopted permitting to a color-interval to wrap around. It extends the ship-building problem's interval coloring to circular interval coloring. It follows that a solution of the ship-building problem is a solution of the submarine-building problem, but the converse is not generally true. Figure 8.2 shows an example of submarine coloring for the weighted graph in Figure 8.1. The inner circle represents the colors and each circular arc I_v represents a color interval assigned to the vertex v. The color interval I_F wraps around.

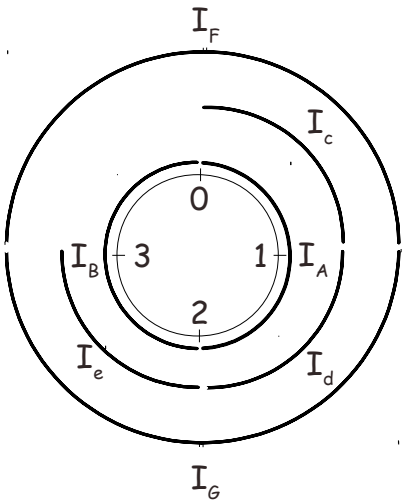

Figure 8.2: An example of a 4-submarine-coloring

For a weighted graph G_w and a number k, we call $submarine(G_w, k)$ an instance of

[11] This has been previously proven by Stockmeyer, but to the best of our knowledge, the proof of Lee et al. is the first publicly available one [LPP08].

the submarine-building problem. For the rest of the chapter we say that G_w is k-ship-colorable, if $ship(G, k)$ has a solution, and we also say that G_w is k-submarine-colorable, if $submarine(G_w, k)$ has a solution.

To the best of our knowledge, this variant of the ship-building problem has never been carefully studied, and it has not been applied to the decoupling of the spilling and assignment problems in local memory management. Many open questions about fragmentation, optimality, complexity and feasibility are tied to this new variant of the ship-building problem.

Unfortunately, the submarine-building problem is also NP-complete on weighted interval graphs as we demonstrate below.

Theorem 8.2.1. *The submarine-building problem is NP-complete.*

Proof. The submarine-building problem is in NP because we can easily see that a solution of the problem can be verified polynomially. To show that the submarine-building problem is NP-hard, we will build from an instance $ship(G, k)$ of the ship-building problem an instance $submarine(G_w, k + 1)$ of the submarine-building problem.

A problem in NP. The submarine-building problem is in *NP* because a solution of the problem can be verified polynomially.

Reduction. From an instance $ship(G_w, k)$ of the ship-building problem, we build an instance $submarine(G_w, k + 1)$ of the submarine-building problem. Let f and ℓ be respectively the minimum of the startpoints of all intervals in G_w and the maximum of the endpoints of all intervals in G_w. The graph G'_w consists of all intervals of G_w and the interval β: $[f, \ell[$ of weight one.

Let θ be a solution of $ship(G_w, k)$; θ maps each interval of G_w to a color interval between 0 and k. We define θ', a function mapping each interval α of G'_w to a color interval between 0 and $k + 1$:

$$\forall \alpha \in G'_w \begin{cases} \theta'(\alpha) = \theta(\alpha) & \text{if } \alpha \in G_w \\ \theta'(\alpha) = [k, k+1[& \text{if } \alpha \notin G_w \end{cases}$$

It follows that θ' is a solution of $submarine(G_w, k + 1)$.

Now, we study the converse case. Let θ' be a solution of $submarine(G_w, k + 1)$. We

define for a color interval $[s, e[$, an integer k, and the functions δ and mod:

$$\begin{aligned} \delta([s, e[, d) &= [s + d, e + d[\\ \text{mod}([s, e[) &= [s \bmod (k + 1), e \bmod (k + 1)[\end{aligned}$$

Let $\theta'(\beta) = [s, s + 1[$ (β is of weight one). We define for each interval α of G_w the function θ:

$$\theta(\alpha) = \text{mod}\,(\delta(\theta'(\alpha), k - s))$$

The interval β of G'_w is live from f to ℓ, therefore there is no other interval of G'_w that occupies the color interval $[s, s + 1[$. $\theta([s, s + 1[) = [k, k + 1[$, and the value on which function θ is equal to $[k, k + 1[$ is $[s, s + 1[$. Thus, θ assigns to each interval α of G_w an interval of some color between 0 and k. If two interfering intervals α and α' have two non-overlapping color intervals c and c' then $\theta(\alpha)$ and $\theta(\alpha')$ are non-overlapping too. It follows that θ is a solution of $ship(G_w, k)$ if θ' is a solution of the $submarine(G_w, k + 1)$.

8.3 Weighted Proper Interval Graph Coloring

We study the properties of weighted proper interval graphs, a subclass of weighted interval graphs. This class is interesting because we will show the submarine-building problem is solvable in linear time for this class: any instance G_w of this class is colorable with $\omega(G_w)$ colors and in linear time. For this subclass, we also have a sufficient criterion permitting to decouple the ship-building problem.

8.3.1 Proper Interval Graph

An interval graph G is a proper interval graph if it is constructed from a family of intervals such that no interval properly contains another [Gol04]. An interval graph is a unit interval graph if all of its intervals have the same length. It has been shown that the classes of proper interval graphs and the unit interval graphs coincide. A weighted graph G_w is a weighted proper interval graph if G is a proper interval graph. Figure 8.3 shows properly ordered weighted-intervals of the real line and their corresponding weighted proper interval graph.

8.3.2 Proper Ordering

Let us consider the representation of G_w, a weighted proper interval graph, on the real line, where the vertices of G_w correspond to intervals on the real line. Let us sort these

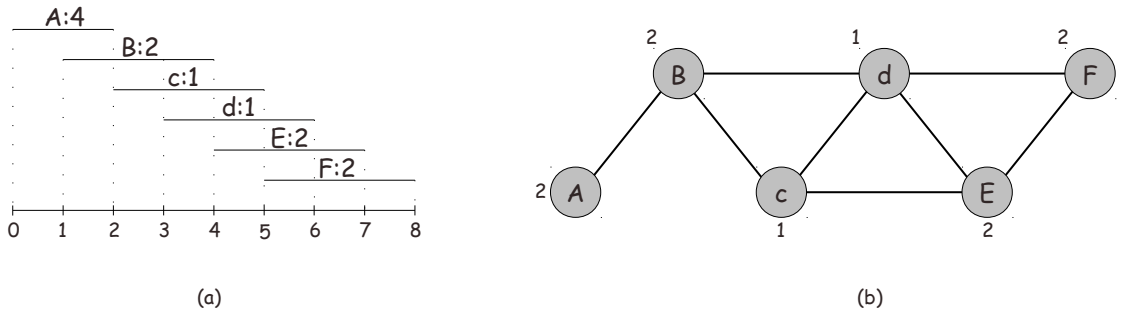

Figure 8.3: An example of a weighted proper interval graph

intervals according to their start points. If two intervals i and i' start at the same point, we can place either i before i' or i' before i. This kind of ordering can be found for any weighted proper interval graph, and is called *proper ordering* in our approach. A proper ordering of the graph in Figure 8.3 is: A, B, c, d, E, F. Based on this ordering, we say that $i \prec i'$, if i is before i'.

Lemma 8.3.1. *If $i \prec i'$ then, either i ends before i' or i and i' start and finish at the same time.*

Proof $i \prec i'$ implies that either i starts before i' or i andi' starts at the same time:

- i starts before i'. Since i cannot properly contain i', then i ends before i'.
- i and i' start at the same time. Since, none of these two intervals cannot properly contain the other, then they end at the same time. □

8.3.3 Decoupled Submarine-Building Problem

Algorithm 7 performs a k-submarine-coloring of intervals of a weighted proper interval graph G_w. It takes as input a sequence of intervals of G_w sorted according to a proper ordering. It assigns to each interval a color interval contiguous to the color interval assigned to the previous interval (according to the proper ordering) in a clockwise manner. Finally, it gives a k-submarine-coloring of the graph as output.

Theorem 8.3.1. *For any weighted proper interval graph G_w, Algorithm 7 guarantees a k-submarine-coloring if and only if $\omega(G_w) \leq k$.*

Proof.

Algorithm 7 SUBMARINE_ASSIGNMENT_ALGORITHM

Input: *intervals*: a list of properly ordered intervals
Var: *map*: an array associating to each interval an offset
1: $offset \leftarrow 0$
2: **foreach:** $i \in intervals$ **do**
3: $map[i] \leftarrow index \bmod k$
4: $offset = offset + \text{WEIGHTOF}(i)$
5: **end for**
6: **return** *map*

Direct. k-submarine-coloring of $G_w \Longrightarrow \omega(G_w) \leq k$.

Any k-submarine-coloring of G_w must assign color intervals that do not overlap to the intervals of a clique of weight $\omega(G_w)$ and this is only possible if $\omega(G_w) \leq k$.

Reciprocal. $\omega(G_w) \leq k \Longrightarrow k$-submarine-coloring of G_w.

We will call a *point*, the moment an interval starts. Let P be the following property: "at point n, the live intervals $i_j, i_{j+1}, \ldots, i_n$ (sorted according to the proper ordering) are assigned to contiguous color intervals that do not overlap in a clockwise manner, in this order: $color(i_j), color(i_{j+1}), \ldots color(i_n)$". The property P is an invariant at every point of Algorithm 7. If the graph contains m nodes, we have consequently m intervals and m points. The proof will be done inductively on points.

Just before the point 1, where the first interval i_1 starts, none of the color intervals are used. At point 1, algorithm 7 assigns to i_1 a color interval starting at 0 and property P is trivially satisfied.

Suppose that property P is satisfied from point 1 to point n, and let us see if property P is satisfied at point $n+1$ (we assume that we have at least $n+1$ intervals in the graph). We call d the number of dead intervals between n and $n+1$ (d can be zero, or $n-j$), we prove four claims successively:

1. i_{j+d} is live. Indeed, if i_{j+d} was dead then all intervals preceding it would also be dead. Therefore, we would have $d+1$ intervals that are dead, which contradicts the definition of d.

2. All the intervals between i_{j+d} and i_n are live too. If an interval i_k between i_{j+d} and i_n is dead then i_{j+d} is also dead because $i_{j+d} \prec i_k$; this leads to a contradiction with the first claim.

3. From the two first claims and the satisfaction of proposition P at point n, we deduce that all live intervals $i_{j+d}, i_{j+d+1}, \ldots, i_n$ are assigned to contiguous colors that do not

overlap, in a clockwise manner, in this order: $color(i_{j+d}), color(i_{j+d+1}), \ldots color(i_n)$.

4. Algorithm 7 assigns to the new interval i_{n+1} a color interval contiguous to the last used color interval (the color of i_n) in a clockwise manner. Therefore, the color intervals assigned to live intervals are contiguous in a clockwise manner, in this order: $color(i_{j+d}), color(i_{j+d+1}), \ldots color(i_n), color(i_{n+1})$. The colors do not overlap because they are all contiguous and they do not exceed $\omega(G_w)$ which does not exceed k.

From the fourth claim, we conclude that at the point $n+1$ the property P is again verified.

Hence, using Algorithm 7 guarantees that at every point, all the live intervals are assigned to contiguous color intervals that do not overlap, and the next starting interval will be assigned to a color interval. Thus, a k-submarine-coloring can be found for G_w if $\omega(G_w) \leq k$.

As far as we are aware, this is the first time a decision problem is shown to be NP-complete on interval graphs and polynomial on proper interval graphs.

8.4 Weighted Not-So-Proper Interval Graphs

We say that two intervals A and B *properly interfere* if A interferes with B such that A strictly starts before B and B strictly ends after A or vice versa.

We define a weighted Not-So-Proper (NSP) interval graph as a weighted interval graph, where each pair of properly interfering intervals A and B, is such that A and B must not be contained in any interval of the graph.

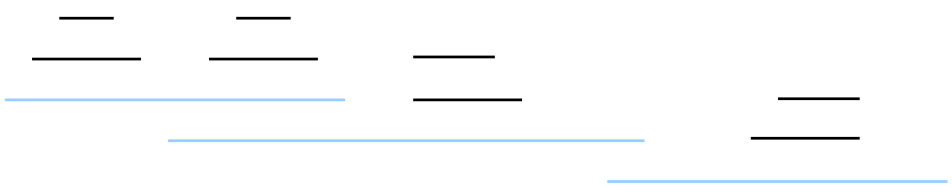

Figure 8.4: An example of weighted NSP graph

Figure 8.4 shows an example of a weighted NSP graph (weights have been omitted in the figure), whereas Figure 8.5 illustrates two weighted graphs that are not weighted NSP graphs. The light blue lines represent intervals that properly interfere, the solid black lines represent intervals that are contained, and the red dashed lines represent the intervals we do not want to have in weighted NSP graphs.

Figure 8.5: Two graphs that are not weighted NSP graphs

The weighted NSP graphs are the class of graphs that includes the weighted proper interval graphs and the superperfect graphs defined by Li et al. [LXK11]. Thus, when the submarine assignment is allowed the weighted NSP interval graphs are guaranteed to be MaxSize-colorable.

Algorithm 8 NSP_Assignment_Algorithm

Input: *intervals*: a list of intervals sorted by increasing start point
Var: *map*: an array associating to each interval an offset
Var: *stack*: a stack used to keep track of contained intervals

1: $offset \leftarrow 0$
2: $container \leftarrow \perp$
3: **foreach:** $i \in intervals$ **do**
4: **if** $container = \perp \vee \neg(\text{Contains}(container, i))$ **then**
5: $container \leftarrow i$
6: **end if**
7: **while** $stack \neq \emptyset$ **do**
8: **if** $\neg(\text{Contains}(\text{Peek}(stack), i)$ **then**
9: $contained \leftarrow \text{Pop}(stack)$
10: $offset \leftarrow (offset + \text{MaxSize} - \text{WeightOf}(contained))$ mod MaxSize
11: **else**
12: break out of the loop
13: **end if**
14: **end while**
15: **if** $container \neq i \wedge (\text{Contains}(container, i))$ **then**
16: $\text{Push}(stack, i)$
17: **end if**
18: $map[i] \leftarrow index$ mod k
19: $index = index + \text{WeightOf}(i)$
20: **end for**
21: **return** *map*

Algorithm 8 performs a submarine assignment for a weighted NSP graph on a local memory of size MaxSize. It receives as input *intervals*, a list of intervals sorted by increasing start point, and returns at the end *map*, a map that associates to each interval an offset into the local memory. Algorithm 8 makes difference between intervals that are

not contained in any other interval, called *containers* and those contained in an interval. The contained intervals are stocked into *stack*, a stack which keeps track of them. The variable *container* keeps track of the last starting container. When a new interval, i, starts, it is verified for containment. If it is not contained in the currently live intervals, it is set as the new container. The function CONTAINS($container, i$) returns true, if i is contained in *container* and false otherwise. Then all the dead intervals are removed from *stack* and the offset is updated. The function WEIGHTOF(i) returns the weight of the interval i. If i is contained into another interval, it is pushed on *stack*. Finally, i is assigned to the current offset, which is then updated.

8.5 Extension to Weighted Interval Graphs

As explained in Section 8.2, from a linearized program it is possible to construct a corresponding weighted interval graph. If the resulting graph is a weighted proper or NSP interval graph, when the submarine assignment is allowed, it is always possible to use MAXSIZE as a criterion to ensure that the assignment phase is feasible without spills. Thus, the allocation algorithm can be decoupled thanks to MAXSIZE criterion. For arbitrary weighted interval graphs, the problem is NP-complete and a heuristic-based solution must be envisaged.

We devised a solution that takes advantage of our submarine assignment algorithm. This solution, that decouples the allocation and assignment, performs the two following steps:

1. it approximates an arbitrary weighted interval graph into a weighted NSP interval graph through spilling and splitting.

2. it performs the assignment with Algorithm 8

We have implemented our approach to approximate weighted interval graphs into weighted NSP interval graphs. We have evaluated it on generated weighted interval graphs, but so far the approximation algorithm has shown modest results and does not perform very well. We are trying to improve it and we do not report the results here, since it is still being studied.

8.6 Conclusion

In this chapter we designed and analyzed a new variant of the ship-building problem called the submarine-building problem. We showed that this problem is NP-complete on

interval graphs, while it is solvable in linear time for proper interval graphs. We also give a criterion to guarantee the feasibility of the submarine-building problems for proper interval graphs and then we extended it to weighted NSP interval graphs. These results offer a general solution (more general than the solution of Li et al. [LXK11]) to decouple spill code generation from local memory assignment. Such a decoupling has been missing for a long time, limiting the transfer to local memory management of the recent wave of algorithmic and experimental successes in register allocation. We believe our results will enable the design of simpler and more robust compilation-time algorithms for local memory management. However, while our approach represents an improvement over state of the art methods, it is limited so far in its practical application.

Part III

Reconciling Register and Local Memory Allocations

Chapter 9

The Clustering Allocator

In Chapter 8, we intended a theoretical study of the decoupled local memory allocation. This study showed encouraging results that can be theoretical foundations of a decoupled approach for local memory allocation, but it is limited so far in its practical application. In this chapter, we propose a heuristic-based solution, the *clustering allocator*, which decouples the local memory allocation problem and aims to minimize the allocation cost. The clustering allocator while devised for local memory allocation appears to be a very good solution to the register allocation problem. After many years of separation, this new algorithm seems to be a bridge to reconcile the local memory allocation and the register allocation problems.

For the sake of exposition, we will first present the version of the clustering allocator devoted to the register allocation and then we will present the general version for the local memory allocation.

9.1 The Clustering Register Allocator

Here, we present a new decoupled register allocation algorithm called clustering allocator. The algorithm has three steps. It first packs the variables by clusters. Then, it performs allocation on the clusters, allocating all the variables of clusters or spilling all of them. Finally, it assigns each allocated cluster to a physical register.

9.1.1 Clustering of Variables

We assume here that an estimated spill cost has been computed for each variable. This spill cost represents the access frequency of a variable, it is high when the variable is frequently accessed and low when it is not.

We define a *cluster* as a group of variables which do not interfere with each other. The cost of a cluster is defined as being the sum of the spill costs of the variables within it. A cluster corresponds to a stable in an interference graph.

The goal when performing register allocation is to optimize the use of registers. Imagine that only one register is available on the target architecture. An allocation of minimum spill cost for a program on this architecture will assign a cluster of maximal cost to the register. Imagine now that there are k available registers on the target architecture. An allocation which assigns the k clusters of highest costs to the k registers will not necessarily be of minimum spill cost, but hopefully it will not be far from the allocation of the minimum cost. This is the intuition upon which the clustering allocator is based and the trick is in the approximation of clusters of higher costs since we are not computing them optimally. We first approximate the cluster of maximal cost and then the cluster of maximal cost which does not contain any interval of the first cluster, and so on, until we put all the variables into a clusters. The cluster of maximal cost, computed from a list of variables, has more chances to contain high cost variables than other variables. Thus, to compute the cluster of maximal cost we iteratively add, as a priority, high cost variables which do not interfere with the variables already present in the cluster.

Algorithm 9 CLUSTER_VARIABLES

Input: *vars_list*: a list of variables ordered by decreasing spill cost
Var: *clusters_list*: a list of clusters

```
1:  while vars_list ≠ ∅ do
2:     cluster ← ⊥
3:     removal ← ⊥
4:     foreach: v ∈ vars_list do
5:        if ¬INTERFERE(v, cluster) then
6:           add v to removal
7:           add v to cluster
8:        end if
9:     end for
10:    remove removal from vars_list
11:    add cluster to clusters_list
12: end while
13: return clusters_list
```

The clustering is performed by Algorithm 9 which transforms *vars_list*, a list of a program's variables ordered by decreasing cost, into *clusters_list*, a list of clusters. It constructs a new cluster at each iteration of the while-loop. To compute the new cluster, each variable v of *vars_list* is added to the cluster if it does not interfere with any of the variables already within the cluster. Every time a variable v is added to *cluster*,

the cluster being created, it is added to *removal*. When all the variables have been tested, *cluster* is added to *clusters_list* and all the variables in *removal* are removed from *vars_list*. Then, the next round of while-loop starts. Finally, Algorithm 9 ends when every variable is in a cluster.

9.1.2 Allocation and Assignment

After the clusters have been computed, Algorithm 10 selects the set of allocated clusters. The k clusters that maximize the sum of their costs are allocated, k being the number of available registers.

Algorithm 10 ALLOCATE_VARIABLES

Input: *clusters_list*: a list of clusters
Input: k: the number of available registers
1: sort *clusters_list* by decreasing cost
2: **if** $size > k$ **then**
3: remove the last $(size - k)$ clusters from *clusters_list*
4: **end if**
5: **return** *clusters_list*

Algorithm 11 performs assignment by mapping each cluster to a physical register.

Algorithm 11 ASSIGN_VARIABLES

Input: *clusters_list*: a list of k clusters
Input: *registers*: the array of k available registers
Var: *map*: an array that maps each cluster to a register
1: $i \leftarrow 0$
2: **foreach:** $c \in clusters\_list$ **do**
3: $map[c] \leftarrow registers[i]$
4: $i \leftarrow i + 1$
5: **end for**
6: **return** *map*

Algorithm 12 performs all the complete steps of the clustering allocator algorithm.

9.2 The Clustering Local Memory Allocator

We present here the generalization of the clustering allocator algorithm for local memory allocation. The main changes between this generalization and the register allocation version are in the computation of clusters which is changed to take into account the different sizes of array blocks.

Algorithm 12 CLUSTERING_REGISTER_ALLOCATOR

Input: *vars_list*: a list of variables ordered by decreasing spill cost
Input: *registers*: the array of k available registers
Var: *map*: an array that map each cluster to a register
1: $clusters\_list \leftarrow$ CLUSTER_VARIABLES($vars\_list$)
2: $allocated\_clusters \leftarrow$ ALLOCATE_VARIABLES($clusters\_list, k$)
3: $map \leftarrow$ ASSIGN_VARIABLES($allocated\_clusters, registers$)
4: **return** *map*

9.2.1 Clustering of Array Blocks

We assume, like in the specialized version for register allocation, that an estimated spill cost has been computed for each array block.

We define a *batch* as a set of array blocks such that each array block of the set interferes at least with another array block of the set. We say that an array block A interferes with a batch B, if A interferes at least with one array block of B. We also say that a batch B_1 interferes with another batch B_2, if an array of B_1 interferes with B_2 or vice versa. A batch has a size which is defined as the sum of the size of the array blocks within the batch. The cost of a batch is defined as the sum of the spill costs of all the array blocks within the batch.

Unlike for register allocation where the variables of same size compete for registers, for local memory allocation, we have many array blocks of different sizes that will share the local memory. Thus, the goal is now to maximize the use of the different portions of the local memory. Let us assume we have a portion P of the local memory of size S. We want to find a set of array blocks (cluster) of maximal cost that can be allocated to P. Like for register allocation, we can define a cluster as a set of array blocks that do not interfere each other. But since, array blocks can have different sizes and many of them can be in P at the same time, our solution is to successively assign batches, which do not interfere and are of size smaller than S, to P. Thus, we re-define a cluster as a set of batches that do not interfere each other. We also define the size of a cluster as the size of the portion of the local memory where it will be assigned.

Now comes the question of how the size of the portions are chosen (the sizes of clusters) and how the clusters are constructed? Algorithm 13 depicts our method. It transforms *arrays_list*, a list of array blocks ordered by decreasing size, into *clusters_list*, a list of clusters. Algorithm 13 picks up A, the first array block (the array block of maximum size) of *arrays_list*, removes it from *arrays_list*, and creates *cluster*, a cluster of the size of A. It creates *batch* and adds it to *cluster*. Then, it adds to *potentials*, all the array blocks that can be potentially added to *cluster*, which are the array blocks that do not

Algorithm 13 CLUSTER_ARRAYS

Input: *arrays_list*: the list of array blocks ordered by decreasing size
Var: *clusters_list*: the list of clusters to return

```
1:  while arrays_list ≠ ∅ do
2:    // create the first batch of the cluster
3:    cluster ← ⊥
4:    A ← REMOVEFIRST(arrays_list)
5:    add A to batch
6:    add batch to cluster.batches
7:    cluster.size ← A.size
8:    // check off all the array blocks that can be in the cluster
9:    potentials ← ⊥
10:   foreach: array ∈ arrays_list do
11:     if ¬INTERFERE(array, batch) then
12:       add array to potentials
13:     end if
14:   end for
15:   // Iteratively find the batches of array blocks that will form the cluster
16:   while potentials ≠ ∅ do
17:     A ← REMOVEFIRST(potentials)
18:     size ← A.size
19:     add A to batch
20:     foreach: array ∈ potentials do
21:       if INTERFERE(array, batch) ∧ (weigh + array.size) ≤ cluster.size then
22:         add array to batch
23:         size = size + array.size
24:       end if
25:     end for
26:     remove all array of batch from potentials
27:     remove all array of batch from arrays_list
28:     foreach: array ∈ potentials do
29:       if INTERFERE(array, batch) then
30:         remove array from potentials
31:       end if
32:     end for
33:     add batch to cluster
34:   end while
35:   add cluster to clusters_list
36: end while
37: return clusters_list
```

interfere with *batch*. Afterwards, Algorithm 13 constructs iteratively the other batches to add to *cluster* (line 16 to 34). It adds the first array block of potentials to the batch and while the size of the batch does not exceed $A.size$ (the size of A), it adds to the batch the array blocks that interfere with one of the array block already in the batch. Once it becomes impossible to add a new array block to the batch, all the array blocks of the batch are removed from *potentials* and *arrays_list*. All the array blocks that interfere with *batch* are also removed from *potentials*, because two array blocks of two different batch of the same cluster cannot interfere. When *potentials* becomes empty, *cluster* is added to the *clusters_list* and a new cluster is constructed. Finally, all the array blocks will be in a cluster and Algorithm 13 returns the *clusters_list*.

9.2.2 Allocation and Assignment

Algorithm 14 ALLOCATE_ARRAYS

Input: *clusters_list*: a list of clusters
Input: *LMSize*: the size of the local memory
Var: *allocated_clusters*: the list of allocated clusters

```
 1: sort clusters_list by decreasing cost
 2: size ← 0
 3: foreach: cluster ∈ clusters_list do
 4:    if (cluster.size + size) ≤ LMSize then
 5:       add cluster to allocated_clusters
 6:       size ← size + cluster.size
 7:    else
 8:       goal ← (LMSize − size)
 9:       new_cluster ← DecreaseToGoal(cluster, goal)
10:       if ¬IsEmpty(new_cluster) then
11:          add new_cluster to allocated_clusters
12:          size ← size + new_cluster.size
13:       end if
14:    end if
15: end for
16: return allocated_clusters
```

Algorithm 14 performs the allocation. It receives as input *clusters_list*, a list of clusters, and *LMSize*, the size of the local memory. It then returns *allocated_clusters*, the list of allocated clusters. It sorts the list of clusters and adds the clusters to *allocated_clusters* while their size does not exceed *LMSize*. When $(cluster.size + size) \geq LMSize$, the function DECREASETOGOAL$(cluster, goal)$ tries to compute a new cluster of size *goal* from the array blocks of *cluster*.

Algorithm 15 ASSIGN_ARRAYS

Input: *allocated_clusters*: the list of allocated clusters
Var: *LMSize*: the size of the local memory
Var: *map*: an array that maps each array to an offset in the local memory

```
1:  offset ← 0
2:  foreach: cluster ∈ allocated_clusters do
3:    foreach: batch ∈ cluster.batchs do
4:      current_offset ← offset
5:      foreach: array ∈ batch do
6:        map[array] ← current_offset
7:        current_offset ← current_offset + array.size
8:      end for
9:    end for
10:   offset ← offset + cluster.size
11: end for
12: return map
```

Algorithm 15 assigns to each array block an offset in the local memory.

Algorithm 16 CLUSTERING_LM_ALLOCATOR

Input: *vars_list*: a list of array blocks ordered by decreasing size
Input: *LMSize*: the size of the local memory
Var: *map*: an array that map each cluster to a register

```
1: clusters_list ← CLUSTER_ARRAYS(clusters_list)
2: allocated_clusters ← ALLOCATE_ARRAYS(clusters_list, LMSize)
3: map ← ASSIGN_ARRAYS(allocated_clusters, LMSize)
4: return map
```

Algorithm 16 performs all the complete steps of the clustering allocator algorithm.

9.3 Experimental Evaluation

We report here the evaluation of our clustering allocator algorithm for register and local memory allocations.

9.3.1 Register Allocation

Methodology

These experiments have been performed with the same methodology used for experiments of Chapter 4. We compare our clustering allocator algorithm with the iterated-optimal allocation and the other algorithms presented in Section 4.2.

Results

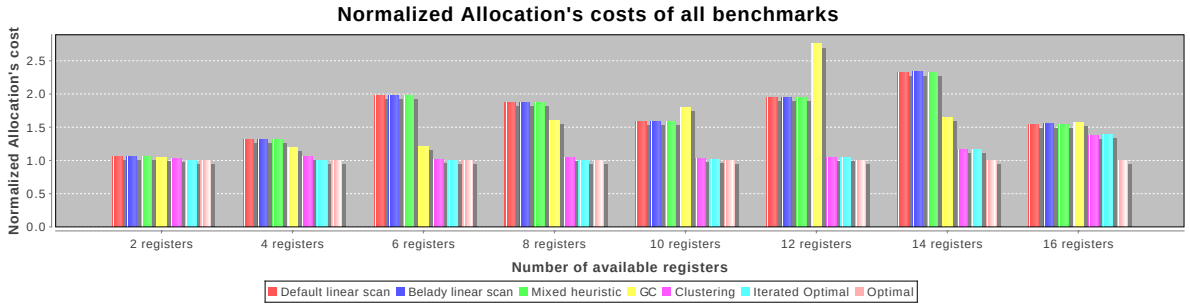

Figure 9.1: clustering allocator compared to other allocators for different register counts

Figure 9.1 shows the allocation costs normalized over the cost of the optimal allocation's cost, for configurations with different register counts going from 2 to 16 registers. For almost all the register counts, the clustering allocator heuristic is close to the optimal and to the iterated-optimal allocation algorithm. For the configurations with 14 and 16 registers, the optimal allocation outperforms both the clustering allocator and the iterated-optimal allocation algorithms. For the configuration with 16 registers the clustering allocator is slightly better than the iterated-optimal allocation algorithm.

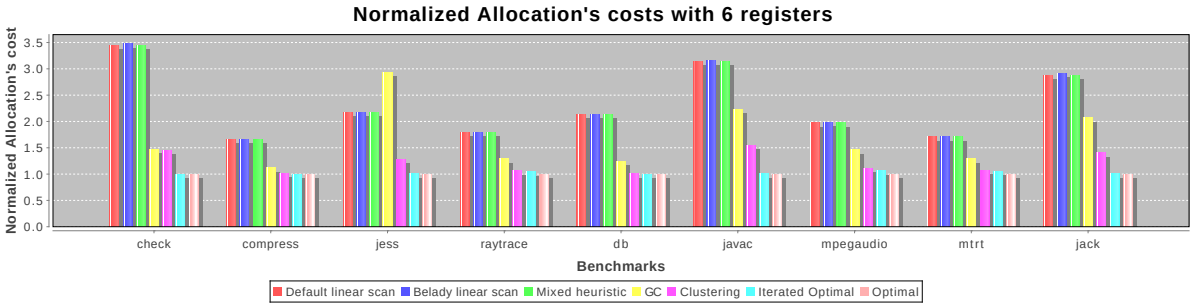

Figure 9.2: clustering allocator compared to other allocators when the register count is 6

Figure 9.2 reports for each individual benchmark the normalized allocation costs when we have a register count of six registers. For `check`, `jess`, `javac`, and `jack`, we note a degradation up to 60% of the optimal. But for the rest of the benchmarks, the clustering allocator algorithm is always close to the optimal and to the iterated-optimal allocation algorithm.

We can notice that the clustering allocator algorithm gives good results compared to the iterated-optimal allocation algorithm, although it has a quadratic time complexity. Indeed the function INTERFERE($v, cluster$) can be done in constant time if we represent

the live range of a cluster as the union of all the live ranges within the cluster and then perform a simple interference test between the live range of the cluster and the live range of the variable.

The clustering allocator and the iterated-optimal allocation algorithm are also very similar because they both proceed iteratively, but there is a subtle difference. The iterated-optimal allocation finds iteratively the set of variables to allocate, while the clustering allocator continues the process until the end before it decides which cluster it is more beneficial to allocate and can sometimes achieve better allocation than the iterated-optimal allocation as shown in Figure 9.1 for the configuration with 16 registers.

9.3.2 Local Memory Allocation

To evaluate the clustering allocator algorithm in the context of local memory allocation, we have generated many weighted interval graphs. We are able to generate the graphs, to differentiate them and we also have control over the density of generated graphs.

Graph Generation

Graphs we have generated for testing our heuristic are based on Scheinerman's studies on random interval graphs [Sch88]. To create an interval graph with n intervals we generate $2n$ independent random variables $L_1, L_2, ..., L_n, R_1, R_2, ..., R_n$ and by coupling these random variables we form intervals $[L_1, R_1], ..., [L_n, R_n]$. We associated weights and costs to these intervals. Eventually these intervals form the random interval graph.

In order to represent the data structures in real life applications, we have added some constraints and limits on these graphs. We not only have control on the number of intervals but also on the minimum length of intervals, the maximum number of concurrently live intervals, the weights and the costs of intervals during the graph generation process.

During the graph generation, we also control the density of the generated graph. For a graph $G = (V, E)$ where n_V is number of vertices and n_E is number of edges in the graph, the density D is calculated with the following formula.

$$D = \frac{2|E|}{|V|(|V|-1)}$$

We are also capable of differentiating the interval graphs before generating weights and costs for them. We did this to justify that our heuristic does not only give good results for some specific type of graphs.

Methodology

For the purpose of our experiments, we have generated 500 graphs of density between 0% and 35%, we call these graphs (35% dense). We also generated 500 graphs of density between 36% and 70% called graphs (70% dense). We finally generated 500 graphs of density between 71% and 100%, called graphs (+70% dense).

We compared our clustering allocator with an ILP-based allocator and a best-fit allocator, which assigns an array block, represented here by our intervals, to the place in the local memory where it fits best. For each generated graph we use a local memory of size MaxSize. Due to the normalization, we use for these experiments the gain instead of the costs since the ILP-allocator can give for certain graphs a spill cost of zero. We also stop the ILP-allocator, for each graph, after half-an hour, if it does not return a solution.

Results

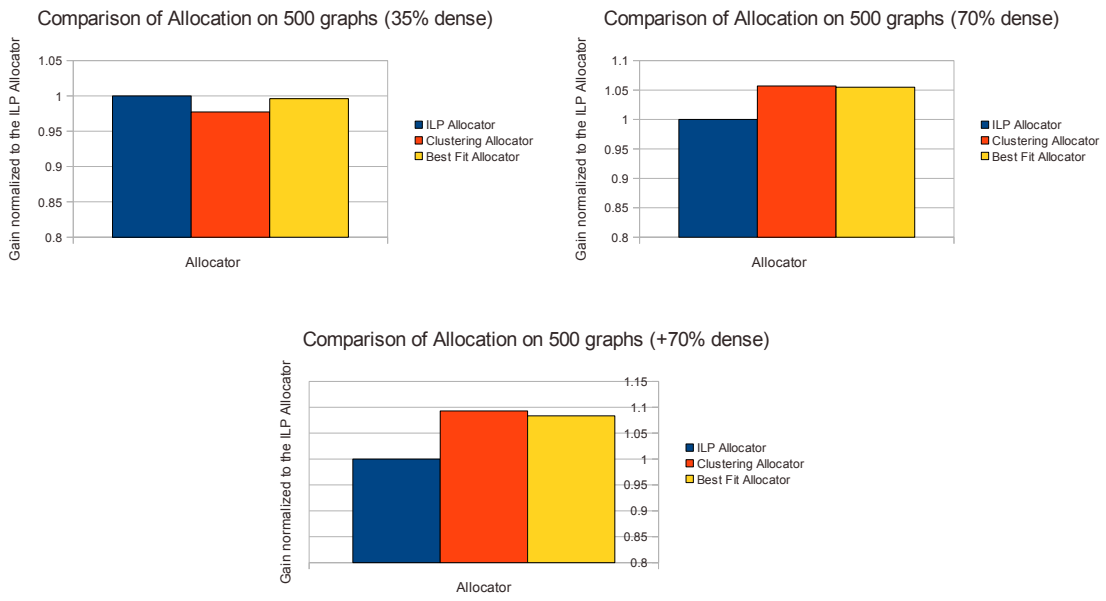

Figure 9.3: clustering allocator for local memory allocation

The three figures show the average of the allocation gain normalized with the ILP-based allocator.

Figure 9.3(a) shows the normalized allocation gain for graphs 35% dense. It does not show a strong improvement for our clustering allocator algorithm which is outperformed

by the best-fit allocator which is in turn outperformed by the ILP allocator. Figure 9.3(b) shows that for graphs 70% dense the clustering allocator and the best-fit are quite similar and produces better results than the ILP allcoator. For graphs that are denser (+70% dense), as depicted in Figure 9.3(c), the clustering allocator is slightly better than both the ILP allocator and the best-fit.

When evaluating the clustering allocator for local memory allocation, we noticed that the ILP-programs were not easy to solve and this shows how the problem is hard. The results we report may appear not very encouraging when comparing to the best-fit, but so far we do not know of a better heuristic to perform local memory allocation and we have asked the questions to many experienced researchers. We also think that the version of clustering allocator for local memory allocation has still room for improvements.

9.4 Discussions about the Algorithm

This section discusses some practical issues, that are specific to register allocation, when using the clustering allocator algorithm. We also discuss how the clustering allocator can be used to perform allocation while the assignment is performed with another algorithm.

9.4.1 Practicability in the context of Register Allocation

When defining the clustering allocator we did not explain how we dealt with the spilled variables. As we previously saw in Chapter 1, when a variable is spilled it does not disappear, it is replaced by a set of tiny variables which must be taken into account. We have two possible solutions for this issue:

1. We can handle them like the linear scan, which assigns them to a free register whenever they are read or written. If there is no available register it spills locally an allocated variable and uses its register. On architecture like x86, we can also take advantage of the complex addressing mode which permits to get operands directly from memory.

2. We can also restart the process like the graph coloring until we find an allocation which does not spill.

9.4.2 Using the Clustering Allocator for Allocation only

For both the local memory and register allocations, the clustering allocator algorithm can be used for only the allocation phase and another algorithm can be used to perform

assignment. In a context where there is no guarantee about the fact that the assignment will be performed without further spills, using another assignment algorithm can result in some extra-spilling after the allocation phase.

In the context of decoupled register allocation of SSA programs, where the assignment can be solved polynomially, the clustering allocator algorithm can be used for the allocation phase. After the allocation, we can guarantee that an assignment is possible and the assignment algorithm will find it. The reason is that the interference graph of a program under (strict) SSA-form is a chordal graph and it will be chordal after an allocation with the clustering allocator. After the allocation, some nodes will be suppressed from the graphs or transformed into a set of tiny live ranges but the resulting graph will still be chordal.

For the same reason, in the context of local memory allocation, when the weighted graphs are superperfect [LXK11] or proper, the clustering allocator algorithm can be used to perform allocation with the guarantee that the forthcoming assignment will be done without further spills.

9.5 Conclusion

In this chapter we have introduced a very interesting heuristic called clustering allocator which appears to work for both register and local memory allocations.

For register allocation, the results show that the clustering allocator outperforms the graph coloring and the linear scan and is often close to the optimal. We hope that its simplicity and its low quadratic-complexity will make it a very serious rival to both linear scan and graph coloring approaches.

For local memory allocation, the results obtained so far are not exceptional but we do believe that the clustering allocator can be improved.

Conclusion and Perspectives

This thesis addressed two memory optimizations, namely register allocation and local memory allocation, performed by the compiler, that aim to optimize the use of registers and local memories within a computer.

Register allocation has been shown to be NP-complete. Indeed, Chaitin et al. have shown that the *spill-free* problem is equivalent to the graph coloring problem [CAC+81]. Moreover, register allocation is not only bounded to the spill-free problem; if the answer to the spill-free problem is no, the goal of register allocation is also to reduce the impact of the spilled variables on the execution time of the program (*spill minimization*).

The local memory allocation problem is NP-complete [VWM04]. While there exist many heuristics to deal with the local memory allocation problem, little is known about the optimization problem, its complexity, and its interplay with other optimizations.

1 Contributions

This dissertation makes the following contributions:

1. **Split Register Allocation:** we designed a split compilation framework dedicated to register allocation. We experimentally validated the effectiveness of split register allocation and its portability with respect to register count variations, relying on annotations whose impact on the bytecode size is negligible.

2. **Iterated-Optimal Allocation:** we have introduced a fast register allocator that performs allocations that are close to optimal ones, by iteratively solving optimal sub-problems of the global allocation problem. The iterated-optimal allocation algorithm is pseudo-polynomial (polynomial, when parameters are fixed to small values) on SSA programs.

3. **Experimental Validation of a Decoupled Local Memory Allocation:** we validate the decoupling of the allocation and assignment stages in the context of

local memory allocation; after an optimal allocation phase relying on a generic and scalable integer linear program, we demonstrated a total absence of fragmentation-induced spills during the assignment-phase.

4. **Basis of Theoretical Foundations for Decoupled Local Memory Allocation:** we designed and analyzed a new variant of the ship-building problem called the submarine-building problem. We showed that this problem is NP-complete on interval graphs, while it is solvable in linear time for proper interval graphs. We also give a criterion to guarantee the feasibility of the submarine-building problems for proper interval graphs and then we extend it to weighted *not-so-proper* interval graphs.

5. **Clustering Allocator:** we propose a heuristic-based solution, the *clustering allocator*, which decouples the local memory allocation problem and aims to minimize the allocation cost. The clustering allocator algorithms packs the variables or array blocks into clusters and performs the allocation on these clusters. The clustering allocator while devised for local memory allocation is a very good solution for the register allocation problem.

2 Perspectives

We are aware that our work is not exhaustive and could be improved in many ways. Here we point out some extensions that can complement this work.

1. We would like to implement the iterated-optimal allocation and the clustering allocator in the context of a SSA-based register allocator offered in a framework like LLVM. This would help us to validate the interesting results presented in Chapter 4 and Chapter 9. Especially, we would like to apply the clustering allocator algorithm to the aliased register allocation where the size of the different variables (8, 16, 32, or 64 bytes) will vary in a limited manner.

2. We have shown that MaxSize can be used as a colorability criterion. We want to extend this colorability criterion to a more general class of graphs or to find a good solution to approximate weighted interval graphs into weighted NSP graphs, while achieving good results. In case of a good approximation algorithm, we plan a complete automation of it and of the clustering allocator algorithm in a research compiler, relying on integrated polyhedral compilation techniques for data-flow analysis, loop transformation and code generation.

3. In our approach to local memory allocation, we only considered *single-threaded* code running on a *single local memory*, we would like to extend this work to environments with many threads sharing the same local memory. We are particularly interested in deterministic thread-level parallelism and synchronous languages where the interactions between live ranges in concurrent threads is well behaved

4. GPU architectures are evolving towards shared memories that are increasingly bigger (AMD Fusion, ARM MALI, Intel Larrabee). Programming these architectures more efficiently requires more attention to memory locality. In such a context, we are interested in new architectures with a (possibly partitioned) global address space and programming models (as HMPP[12] and OpenCL), that furnish more support for software-controlled local memories, and thus help the developer or the compiler manage these software-controlled local memories in a more transparent way. We would like to consider these new models in our future work.

[12] `http://www.openhmpp.org`

List of Figures

List of Tables

Bibliography

[ABS02] Oren Avissar, Rajeev Barua, and Dave Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Trans. Embed. Comput. Syst.*, 1(1):6–26, 2002.

[Aea05] B. Alpern and et al. The Jikes RVM project: Building an open source research community. *IBM Systems Journal*, 44(2):399–418, 2005.

[AG01] Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. In *PLDI'01*, pages 243–253, Snowbird, Utah, USA, June 2001.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2 edition, August 2006.

[And03] Christian Andersson. Register allocation by optimal graph coloring. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 33–45, Berlin, Heidelberg, 2003. Springer-Verlag.

[ANH99] Ana Azevedo, Alex Nicolau, and Joe Hummel. Java annotation-aware just-in-time (ajit) compilation system. In *Proc. ACM 1999 Conf. on Java Grande*, pages 142–151, 1999.

[AP02] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, October 2002.

[ARM98] ARM. Document No. ARM DDI 0084D, ARM Ltd. ARM7TDMI-S data sheet, 1998.

[Ayc03] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35:97–113, June 2003.

[BCT94] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, 1994.

[BCZ90] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *In Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 125–134, 1990.

[BDdD+09] Benoit Boissinot, Alain Darte, Benoit Dupont de Dinechin, Christophe Guillon, and Fabrice Rastello. Revisiting out-of-SSA translation for correctness, code quality and efficiency. In *CGO'09*, pages 114–125, 2009.

[BDGR06a] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register allocation: What does the NP-completeness proof of Chaitin et al. really prove? In *WDDD'06*, Boston, MA, 2006.

[BDGR06b] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register allocation: What does the NP-completeness proof of Chaitin et al. really prove? or revisiting register allocation: Why and how. In *LCPC'06*, LNCS, New Orleans, Louisiana, 2006. Springer Verlag.

[BDR07a] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of register coalescing. In *CGO'07*, 2007.

[BDR07b] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of register coalescing. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 102–114, Washington, DC, USA, 2007. IEEE Computer Society.

[BDR07c] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of spill everywhere under ssa form. In *LCTES'07*, pages 103–112, 2007.

[BDR08] Florent Bouchez, Alain Darte, and Fabrice Rastello. Advanced conservative and optimistic register coalescing. In *CASES'08*, pages 147–156, 2008.

[BEH91] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrating register allocation and instruction scheduling for riscs. *SIGPLAN Not.*, 26:122–131, April 1991.

[Bel66] L. A. Belady. A study of replacement algorithms for virtual storage computers. *9th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1966.

[BH09] Matthias Braun and Sebastian Hack. Register spilling and live-range splitting for ssa-form programs. In Oege de Moor and Michael Schwartzbach, editors, *Compiler Construction*, volume 5501 of *Lecture Notes in Computer Science*, pages 174–189. Springer Berlin / Heidelberg, 2009.

[BHT92] Miklós Biró, Mihály Hujter, and Zsolt Tuza. Precoloring extension. i. interval graphs. *Discrete Math*, pages 100–1, 1992.

[Bla06] S. M. Blackburn. The dacapo benchmarks: java benchmarking development and analysis. In *OOPSLA'06*, pages 169–190, New York, NY, 2006. ACM.

[BLAA01] Rajeev Barua, Walter Lee, Saman Amarasinghe, and Anant Agarawal. Compiler support for scalable and efficient memory systems. *IEEE Trans. Comput.*, 50:1234–1247, November 2001.

[BPMR03] Michael Burns, Gregory Prier, Jelena Mirkovic, and Peter Reiher. Implementing address assurance in the Intel IXP, 2003.

[BSL+02] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*, CODES '02, pages 73–78, New York, NY, USA, 2002. ACM.

[CAC+81] G. J. Chaitin, Mark A. Auslander, Ashok K. Chandra John Cocke, Martin E. Hopkins, and Peter W.Markstein. Register allocation via coloring. *Computer languages*, 6:47–57, 1981.

[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 Trans. Program. Lang. Syst.*, 13:451–490, October 1991.

[Cha82] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN'82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–105, New York, NY, 1982. ACM.

[Che92] Lin Chen. Optimal parallel time bounds for the maximum clique problem on intervals. *Inf. Process. Lett.*, 42(4):197–201, 1992.

[CMB06] John Cavazos, J. Eliot B. Moss, and Michael F.P. Oâ Boyle. Hybrid optimizations: Which optimization algorithm to use? *CC'06*, 2006.

[CR95] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in olden, 1995.

[DLLK04] Victor De La Luz and Mahmut Kandemir. Array regrouping and its use in compiling data-intensive embedded applications. *IEEE Trans. Comput.*, 53(1):1–19, 2004.

[ea88] M. Berry et al. The perfect club benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputer Applications*, 3:5–40, 1988.

[Fab79] Janet Fabri. Automatic storage optimization. In *ACM Symp. on Compiler Construction*, pages 83–91, 1979.

[FCL00] Martin Farach-Colton and Vincenzo Liberatore. On local register allocation. *J. of Algorithms*, 37(1):37–65, 2000.

[GA96] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, 1996.

[GEB08] Andy Georges, Lieven Eeckhout, and Dries Buytaert. Java performance evaluation through rigorous replay compilation. *SIGPLAN Not.*, 43(10):367–384, 2008.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley)).* Addison-Wesley Professional, 2005.

[Gol04] Martin Charles Golumbic. *Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics, Vol 57).* North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 2004.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[HGG06] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in SSA-form. In *CC'06*, pages 247–262, 2006.

[HR00] E.G. Hallnor and S.K. Reinhardt. A fully associative software-managed cache design. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 107 –116, june 2000.

[IBMD07] Ilya Issenin, Erik Brockmeyer, Miguel Miranda, and Nikil Dutt. DRDU: A data reuse analysis technique for efficient scratch-pad memory management. *ACM Trans. Des. Autom. Electron. Syst.*, 12(2):15, 2007.

[Ins97] Texas Instruments. TMS370Cx7x 8-bit microcontroller, Texas Instruments, 1997.

[Iye99] Arun Iyengar. Design and performance of a general-purpose software cache. In *Journal of Parallel and Distributed Computing*, 1999.

[JK00] J. Jones and S. N. Kamin. Annotating java class files with virtual registers for performance. *Concurrency – Practice and Experience*, 12(6):389–406, 2000.

[Jon02] Joel Jones. *Annotating mobile code for performance*. PhD thesis, Champaign, IL, USA, 2002. AAI3070343.

[Kan01] Mahmut T. Kandemir. Array unification: A locality optimization technique. In *CC'01*, pages 259–273, 2001.

[KAP97] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *PLDI'97*, pages 346–357, Las Vegas, Nevada, June 1997.

[KC01] Chandra Krintz and Brad Calder. Using annotations to reduce dynamic optimization time. In *PLDI'01*, pages 156–167, New York, NY, 2001. ACM Press.

[KDH+05] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5), 2005.

[KRI+01] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In *DAC'01*, pages 690–695, 2001.

[KS98] K. Knobe and V. Sarkar. Array SSA form and its use in parallelization. In *ACM Conf. on Principles of Programming Languages (PoPL'08)*, pages 107–120, San Diego, CA, January 1998.

[LCC+07] P. Lesnicki, A. Cohen, M. Cornero, G. Fursin, A. Ornstein, and E. Rohou. Split compilation: an application to just-in-time vectorization. In *GREPS'07*, Brasov, Romania, September 2007.

[Lee98] Corinna G. Lee. UTDSP benchmarks, 1998.

[LFCK99] Vincenzo Liberatore, Martin Farach-Colton, and Ulrich Kremer. Evaluation of algorithms for local register allocation. In *Proceedings of the 8th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*, pages 137–152, London, UK, 1999. Springer-Verlag.

[LFX09] Lian Li, Hui Feng, and Jingling Xue. Compiler-directed scratchpad memory management via graph coloring. *ACM Trans. Archit. Code Optim.*, 6:9:1–9:17, October 2009.

[LGX05] Lian Li, Lin Gao, and Jingling Xue. Memory coloring: A compiler approach for scratchpad memory management. In *PACT'05*, pages 329–338, 2005.

[LNX07] Lian Li, Quan Hoang Nguyen, and Jingling Xue. Scratchpad allocation for data aggregates in superperfect graphs. *SIGPLAN Not.*, 42(7):207–216, 2007.

[LPP08] Jonathan K. Lee, Jens Palsberg, and Fernando Magno Quintão Pereira. Aliased register allocation for straight-line programs is NP-compl ete. *Theoretical Computer Science*, 407:258–273, 2008. Preliminary version in Proceedings of ICALP'07, 34th International Colloquium on Automata, Languages and Programming, pages 680–691, Wroclaw, Poland, July 2007.

[LXK11] Lian Li, Jingling Xue, and Jens Knoop. Scratchpad memory allocation for data aggregates via interval coloring in superperfect graphs. *ACM Trans. Embed. Comput. Syst.*, 10:28:1–28:42, January 2011.

[MFA01] Csaba Andras Moritz, Matthew Frank, and Saman P. Amarasinghe. Flexcache: A framework for flexible compiler generated data caching. In *Revised Papers from the Second International Workshop on Intelligent Memory Systems*, IMS '00, pages 135–146, London, UK, 2001. Springer-Verlag.

[Mot98] Motorola. M-CORE – MMC2001 reference manual, Motorola Corporation, 1998.

[MP02] Hanspeter Mössenböck and Michael Pfeiffer. Linear scan register allocation in the context of ssa form and register constraints. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 229–246, London, UK, 2002. Springer-Verlag.

[Nec97] G. Necula. Proof-carrying code. In *PoPL'97*, January 1997.

[NP95] Cindy Norris and Lori L. Pollock. An experimental study of several cooperative register allocation and instruction scheduling strategies. In *Proceedings of the 28th annual international symposium on Microarchitecture*, MICRO 28, pages 169–179, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[NVI08] NVIDIA. NVIDIA unified architecture GeForce 8800 GT, 2008.

[PF92] Todd A. Proebsting and Charles N. Fischer. Probabilistic register allocation. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 300–310, New York, NY, USA, 1992. ACM.

[PM98] J. Park and S.-M. Moon. Optimistic register coalescing. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, pages 196–, Washington, DC, USA, 1998. IEEE Computer Society.

[PP05] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In *In Proceedings of APLASâ05, Asian Symposium on Programming Languages and Systems*, pages 315–329, 2005.

[PP08] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation by puzzle solving. In *Proceedings of PLDI'08, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 216–226, Tucson, Arizona, June 2008.

[PQVR+01] Patrice Pominville, Feng Qian, Raja Vallée-Rai, Laurie J. Hendren, and Clark Verbrugge. A framework for optimizing java using attributes. In *CC'01*, LNCS, pages 334–354, London, UK, 2001. Springer-Verlag.

[PS99] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, 1999.

[RHAR06] Silvius Rus, Guobin He, Christophe Alias, and Lawrence Rauchwerger. Region array SSA. In *PACT'06*, pages 43–52, 2006.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 12–27, New York, NY, USA, 1988. ACM.

[SB07] Vivek Sarkar and Rajkishore Barik. Extended linear scan: An alternate foundation for global register allocation. In Shriram Krishnamurthi and Martin Odersky, editors, *CC'07*, volume 4420 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 2007.

[SBMG00] Mauricio Serrano, Rajesh Bordawekar, Sam Midkiff, and Manish Gupta. Quicksilver: A quasi-static compiler for java. In *OOPSLA'00*, 2000.

[Sch88] E. Scheinerman. Random interval graphs. *Combinatorica*, 8:357–371, 1988. 10.1007/BF02189092.

[SL62] S.S. and Lavrov. Store economy in closed operator schemes. *USSR Computational Mathematics and Mathematical Physics*, 1(3):810 – 828, 1962.

[SPP07] Anita Saha, Madhumangal Pal, and Tapan K. Pal. Selection of programme slots of television channels for giving advertisement: A graph theoretic approach. *Inf. Sci.*, 177(12):2480–2492, 2007.

[SvP01] Jan Sjödin and Carl von Platen. Storage allocation for embedded processors. In *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '01, pages 15–23, New York, NY, USA, 2001. ACM.

[SWLM02] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '02, pages 409–, Washington, DC, USA, 2002. IEEE Computer Society.

[TE04a] S.A.A. Touati and C. Eisenbeis. Early periodic register allocation on ilp processors. *Parallel Processing Letters*, 14(2), June 2004.

[TE04b] Sid-Ahmed-Ali TOUATI and Christine Eisenbeis. Early Periodic Register Allocation on ILP Processors. *Parallel Processing Letters, World Scientific*, 14:n2, June 2004.

[THS98] Omri Traub, Glenn Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *PLDI'98*, pages 142–151, New York, NY, 1998. ACM Press.

[UB03] Sumesh Udayakumaran and Rajeev Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *CASES'03*, pages 276–286, 2003.

[UDB06] Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.*, 5(2):472–511, 2006.

[VWM04] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '04, pages 104–109, New York, NY, USA, 2004. ACM.

[Wil96] Tim Wilkinson. Kaffe: A free jit virtual machine to run java code, 1996.

[WM05] Christian Wimmer and Hanspeter Mössenböck. Optimized interval splitting in a linear scan register allocator. In *VEE'05*, pages 132–141, New York, NY, 2005. ACM.

[Wol95] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[YG87] M. Yannakakis and F. Gavril. The maximum k-colorable subgraph problem for chordal graphs. *Information Processing Letters*, 24(2):133–137, 1987.