Decoupled (SSA-based) Register Allocators : from Theory to Practice, Coping with Just In Time Compilation and Embedded Processors Constraints.

Quentin Colombet

Numéro National de Thèse: 2012ENSL0777

THÈSE

en vue d'obtenir le grade de

Docteur de l'École Normale Supérieure de Lyon – Université de Lyon

spécialité : Informatique

Laboratoire de l'Informatique du Parallèlisme

École doctorale Informatique et Mathématiques de Lyon

présentée et soutenue publiquement le 07/12/12

par Monsieur Quentin COLOMBET

Decoupled (SSA-based) Register Allocators : from Theory to Practice, Coping with Just-In-Time Compilation and Embedded Processors Constraints.

Directeur de thèse : Monsieur Alain DARTE
Co-directeur de thèse : Monsieur Fabrice RASTELLO

Après avis de : Monsieur Vivek SARKAR, Membre/Rapporteur
Monsieur Erven ROHOU, Membre/Rapporteur

Devant la Commission d'examen formée de :
Monsieur Erik ALTMAN, Membre
Monsieur Albert COHEN, Membre
Monsieur Alain DARTE, Membre
Monsieur Vivek SARKAR, Membre/Rapporteur
Monsieur Fabrice RASTELLO, Membre
Monsieur Erven ROHOU, Membre/Rapporteur

Abstract

In compilation, register allocation is the optimization that chooses which variables of the source program, in unlimited number, are mapped to the actual registers, in limited number. Parts of the live-ranges of the variables that cannot be mapped to registers are placed in memory. This eviction is called *spilling*.

Until recently, compilers mainly addressed register allocation via graph coloring using an idea developed by Chaitin et al. [33] in 1981. This approach addresses the spilling and the mapping of the variables to registers in one phase. In 2001, Appel and George [3] proposed to split the register allocation in two separate phases. This idea yields better and independent solutions for both problems, but requires a very aggressive form of live-range splitting, *split everywhere*, which renames all variables between all instructions of the program. However, in 2005, several groups [27, 84, 56, 16] observed that the static single assignment (SSA) form provides sufficient split points to decouple the register allocation as Appel and George suggested, unless register aliasing or precoloring constraints are involved.

Prior to this thesis, no alternative to this aggressive live-range splitting was available for decoupled register allocation with register aliasing. Other forms of architectural constraints, e.g., encoding and application binary interface (ABI) constraints, can be handled via a form of live-range splitting, more intensive than SSA but less aggressive than split everywhere [55].

This thesis covers all the aspects of decoupled register allocation under SSA with architectural constraints. In a first part, we focused on the spilling problem. Using an exact formulation of the spilling problem, we investigated the impact of SSA during this phase and compared several spilling approaches, exact or not, in our model. This comparison pointed out that SSA complicates the problem and that the state-of-the-art objective function, *the static spill cost*, used to optimize spill code placement may not be relevant for runtime performances. Following these observations, we evaluated several, existing or not, simplifications of the spilling problem that should help design a good spilling heuristic in terms of runtime performance, though not necessarily static spill cost.

The second part of the manuscript is dedicated to the assignment phase. We showed how to handle regular architectural constraints without intensive live-range splitting. Our approach is compatible with graph-coloring-based approaches, but also with scan-based approaches (traversal of the control-flow graph), as we demonstrated with our fast register allocator, *tree-scan* . Regarding register aliasing, we showed how to limit the effect of split everywhere and still having the decoupling property.

Finally, in a third part, we demonstrated how to improve the final assembly code via local recoloring techniques. These techniques help to deconstruct colored SSA, i.e., register-allocated SSA, and eliminates many copies instructions inserted for live-range splitting purposes.

We wanted all this work to be applicable to just-in-time (JIT) compilation for embedded targets, thus speed and memory footprint were a concern.

Keywords: Decoupled register allocation, register aliasing, precoloring, JIT, SSA, spilling.

Résumé

En compilation, l'allocation de registres est l'optimisation qui choisit quelles variables du programme source, en nombre illimité, sont stockées dans les registres physiques, en nombre limité. Les variables qui ne peuvent tenir en registre sont placées en mémoire. Cette éviction est appelée *spilling*.

Jusqu'à récemment, les compilateurs traitaient l'allocation de registres globalement via la coloration de graphes en utilisant une idée développée par Chaitin et al. [33] en 1981. En 2001, Appel et George [3] ont proposé de découper l'allocation de registres en deux phases distinctes. Cette idée permet de définir de meilleures solutions pour les deux problèmes, mais nécessite une forme très agressive de renommage des variables, le *split everywhere*, qui renomme toutes les variables entre toutes les instructions du programme. Cependant, en 2005, plusieurs groupes [27, 84, 56, 16] ont observé qu'en l'absence de contraintes d'aliasing de registres et de précoloriage, le passage en static single assignment (SSA) définit des points de renommage suffisants pour découpler l'allocation de registres tel que suggéré par Appel et George.

Avant cette thèse, pour les approches découplées, seule la technique agressive du *split everywhere* était disponible en présence d'aliasing de registres. Les autres formes de contraintes d'allocation, contraintes d'encodage d'instructions et celles dites d'application binary interface (ABI) peuvent être traitées par du renommage plus intensif qu'avec SSA mais moins qu'avec le *split everywhere* [55].

Cette thèse couvre tous les aspects de l'allocation de registres découplée sous SSA avec des contraintes architecturales. Dans une première partie, nous nous sommes concentrés sur le problème du *spill*. En utilisant une formulation exacte, nous avons mis en évidence le fait que SSA complique le problème et que la fonction objective de l'état de l'art, le coût statique de spill, utilisée pour optimiser le placement du code de spill n'est pas pertinente en ce qui concerne les performances d'exécution. Suivant ces observations, nous avons évalué plusieurs simplifications du problème, proposées antérieurement ou non, qui devraient permettre de concevoir un bonne heuristique en termes de performances d'exécution mais pas nécessairement de coût statique de spill.

La deuxième partie du manuscrit est dédiée à la phase d'assignation aux registres. Nous avons montré comment éviter un renommage intensif pour gérer les contraintes architecturales habituelles. Notre méthode est compatible avec les approches basées sur la coloration de graphe, mais aussi sur les scans (parcours du graphe de flot de contrôle), comme nous l'avons démontré avec notre allocateur de registre rapide, le *tree-scan* . En présence d'aliasing de registres, nous avons montré comment limiter les effets de *split everywhere* tout en ayant les bonnes propriétés des approches découplées.

Finalement, dans une troisième partie, nous avons montré comment améliorer le code assembleur final avec des techniques de recoloriage. Celles-ci aident à la déconstruction de SSA et à l'élimination des copies insérées par le renommage.

Enfin, nous voulions que nos travaux soient applicables à la compilation dite just-in-time (JIT) pour processeurs embarqués, ainsi la vitesse et l'empreinte mémoire ont été une préoccupation de tous les instants.

Mots clés : Allocation de registres découplée, JIT, SSA, aliasing de registres, contraintes de registres (precoloring), vidage en mémoire (spilling).

Contents

Acronym **5**

I Introduction **6**

1 Introduction **7**
1.1 Register Allocation . . . 7
1.2 Motivations . . . 9
1.3 Outline and Contributions . . . 10

2 Prerequisites and Hypotheses **12**
2.1 Program Representation . . . 12
2.1.1 Code Operations . . . 12
2.1.2 Control Flow Graph (CFG) . . . 14
2.2 Static Single Assignment (SSA) . . . 15
2.2.1 ϕ-Functions . . . 15
2.2.2 Strictness and Dominance Property . . . 16
2.2.3 Conventional SSA . . . 16
2.2.4 Deconstructing SSA . . . 16
2.2.5 Liveness and SSA . . . 17
2.3 Register Allocation . . . 18
2.3.1 Hypotheses . . . 18
2.3.2 Global Register Allocation . . . 21
2.3.3 Decoupled Register Allocation . . . 25

II Spill **31**

3 Studying Optimal Spilling in the Light of SSA **33**
3.1 Formulating "Optimal" Spilling . . . 34
3.1.1 Existing "Exact" Formulations . . . 34
3.1.2 Limitations of Existing Approaches . . . 35
3.2 A More "Optimal" Formulation . . . 38
3.2.1 Basic Formulation . . . 39
3.2.2 Emulating Other Formulations . . . 41
3.2.3 Handling SSA and ϕ-Functions . . . 42
3.2.4 Extended Formulation . . . 45
3.3 Experiments . . . 50

3.3.1 Solving Time . . . 51
3.3.2 Static Spill Cost . . . 52
3.3.3 Dynamic Counts . . . 56
3.3.4 Execution Time Measurements . . . 57
3.4 Conclusion . . . 62

4 Towards a Better Spilling Heuristic 63
4.1 Existing Spilling Criteria . . . 63
4.1.1 Static Spill Cost . . . 63
4.1.2 Furthest First . . . 65
4.2 Simplifying Assumptions . . . 67
4.2.1 The Instruction `store` . . . 67
4.2.2 The Instruction `load` . . . 70
4.3 Existing Heuristics . . . 72
4.3.1 Graph Coloring . . . 72
4.3.2 Scan-Based Approaches . . . 74
4.3.3 Decoupled Approaches . . . 75
4.4 Improving Runtime . . . 76
4.4.1 Latency . . . 76
4.4.2 Helping the Scheduler . . . 78
4.5 Conclusion . . . 79

III Coloring with Affinities and Antipathies 81

5 Coloring with Encoding Constraints 83
5.1 Graph Coloring with Repairing . . . 85
5.1.1 Model and restrictions . . . 85
5.1.2 Strategies . . . 86
5.1.3 Repairing Code . . . 90
5.2 Tree-Scan . . . 91
5.2.1 The Basic Algorithm . . . 91
5.2.2 Repairing . . . 94
5.3 Biased Coloring . . . 99
5.4 Related Work . . . 102
5.5 Experiments . . . 105
5.5.1 Graph Coloring and Repairing . . . 105
5.5.2 Tree-Scan . . . 107
5.6 Conclusion . . . 114

6 Decoupled Graph-Coloring Register Allocation with Hierarchical Aliasing 115
6.1 Background . . . 116
6.2 Spilling Test in Face of Aliasing . . . 120
6.2.1 Checking Colorability via Smith's Simplification Test . . . 120
6.2.2 Correct Spilling Test Handling Aliasing and Precoloring . . . 121
6.2.3 Improving Smith's Test with Live-Range Merging . . . 122
6.3 Semi-Elementary Form . . . 124
6.3.1 Criterion to Avoid Live-Range Splitting . . . 124
6.3.2 Local Merging of Live-Ranges . . . 126

6.4 Experiments . . . 128
6.5 Conclusion . . . 133

IV Post Phases 134

7 Parallel Copy Motion 136
7.1 Parallel Copy Motion . . . 137
7.1.1 Parallel Copies . . . 137
7.1.2 Moving a Parallel Copy Out of an Edge . . . 139
7.1.3 Parallel Copy Motion Inside Basic Blocks . . . 140
7.2 Permutation Motion and Region Recoloring . . . 141
7.2.1 Reversible Parallel Copies & Permutations . . . 141
7.2.2 Region Recoloring . . . 142
7.3 Applications . . . 143
7.3.1 Removing Parallel Copies from Critical Edges . . . 143
7.3.2 Shrinking Parallel Copies in a Basic Block . . . 147
7.4 Experiments . . . 150
7.4.1 The Impact of Copy Motion Out of Edges . . . 151
7.4.2 The Impact of Copy Motion in Basic Blocks . . . 153
7.4.3 All Together . . . 155
7.5 Conclusion . . . 156

8 Elimination of Parallel Copies Using Copy Motion on Data Dependence Graphs 158
8.1 Data Dependence Graphs . . . 159
8.1.1 Parallel Copies . . . 160
8.1.2 Parallel Copy Motion . . . 161
8.2 Copy Elimination on Data Dependence Graphs . . . 161
8.2.1 Downward Motion of Definitions . . . 163
8.2.2 Upward Motion of Uses . . . 173
8.2.3 Code Motion Past Cyclic Parallel Copies . . . 182
8.2.4 Algorithm Complexity . . . 182
8.2.5 Additional Remarks . . . 183
8.3 Experiments . . . 184
8.3.1 Copy Elimination after Full Coalescing . . . 185
8.3.2 Copy Elimination after *Decoupled* Register Allocation . . . 187
8.3.3 Coalescing versus DDG-Based Copy Elimination . . . 189
8.3.4 Runtime Behavior . . . 190
8.4 Related Work . . . 191
8.5 Conclusion . . . 192

V Conclusion 194

9 Conclusion 195
9.1 Contributions . . . 195
9.1.1 Spilling . . . 195
9.1.2 Coloring . . . 196
9.1.3 Post Phases . . . 197

9.2 Perspectives 198
9.2.1 Spilling 198
9.2.2 Coloring 198
9.2.3 Post Phases 199

List of Publications **200**

Bibliography **201**

A Appendix **209**
A.1 Coloring with Encoding Constraints 209

Acronym

ABI application binary interface

BF brute force coalescer

CFG control flow graph

CISC complex instruction set computing

CSSA conventional static single assignment

DDG data dependence graph

DFS depth-first search

IG interference graph

ILP integer linear programming

IR intermediate representation

IRC iterated register coalescer

ISA instruction set architecture

JIT just-in-time

KERNELS benchmarks from STMicroelectronics

LAO linear assembly optimizer

OPEN64 open source version of the SGI Pro64 compiler [49]

RISC reduced instruction set computing

RPO reverse post-order

SSA static single assignment

SSI static single information

VLIW very-long instruction word

Part I

Introduction

Chapter 1

Introduction

In computer science, compilation is the process that translates a source program into a destination program, that is equivalent in terms of behavior. Both programs may share their programming languages. Such process is called "source-to-source compilation". But it is not the common usage of compilers, the programs that perform the compilation. Indeed, compilers are generally used to translate machine-independent, usually human-written, programs into machine-depend programs. In their last phases, compilers have to deal with the actual constraints of the target machine, which, by definition, were not present in the original programs. This thesis focuses on this low-level aspect of compilation called "back-end compilation" and in particular on *register allocation*, which deals, among these architectural constraints, with the limited amount of fast storage space, i.e., the registers.

1.1 Register Allocation

Register allocation consists in mapping the unbounded set of variables used in a low-level program representation to the limited number of registers available in the target architecture. When not all variables can be mapped to registers, some are stored in memory to reduce register demand. This eviction to memory is called *spilling*. Memory transfers are costly in execution time, power dissipation, and code size, thus a good register allocator should reduce spilling in order to preserve the gains of previous optimizations. Indeed, other optimizations have their own profitability model that may not match register allocation concerns. Moreover, according to Hennessy and Patterson [59], register allocation adds the largest single performance improvement to compiled programs. For instance, Figure 1.1 gives the assembly code produced for x86 for a function computing factorial. In this example, the assembly code generated with register allocation enabled is twice as fast as the assembly without. Indeed, without register allocation, the program accesses variables `n` and `res` via the stack, i.e., they are allocated in memory, whereas with register allocation, it directly uses registers. Thus, register allocation has been extensively studied in the past.

As a reminder, it is always beneficial to improve the performance of the program in the embedded world even when the reactivity of the system is not a concern. Indeed, if a program requires less computations, the frequency of

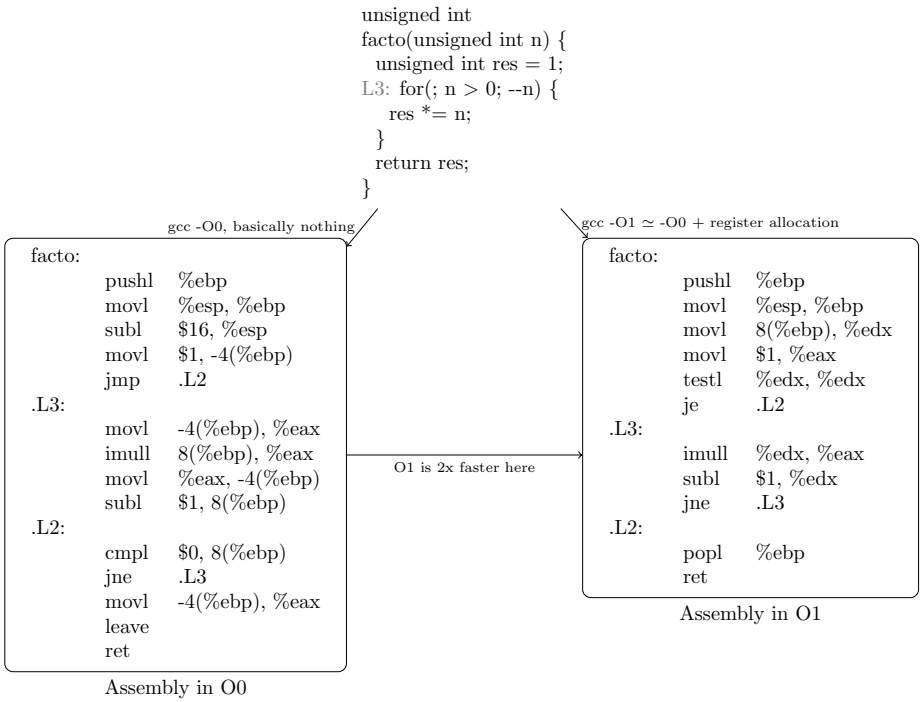


Figure 1.1: When enabling register allocation on gcc 4.4.3 for the x86 target, the generated assembly code is 2x faster for this example. In the x86 assembly code, the definition is the second operand of the instruction. For instructions with two arguments, like imull, the second operand is both read and written. The L3 label denotes the body of the original loop. Without register allocation, assembly in O0, `n` is accessed via memory location 8(%ebp) and `res` is accessed via memory location -4(%ebp). With register allocation, assembly in O1, accesses to the stack are eliminated as `n` is in register %edx and `res` is in register %eax.

the processor can be decreased without slowing down the application. On the other hand, if the frequency is not changed, the processor ends the computations earlier and can enter in idle mode sooner. In both cases, for the same amount of work, this spares the battery of the system.

Until recently, compilers performed register allocation using variants of graph coloring, as developed by Chaitin et al. [33]. This method gives fairly-good results in practice. However, nowadays, compilers are used in many different contexts. In particular, they have to cope with memory and/or time constraints, as implied by just-in-time (JIT) compilation, that are not compatible with graph-coloring-based approaches. Indeed, these approaches are known to be memory consuming and quite slow.

In the past few years, some researchers proposed to decompose register allocation in two phases [3, 56]. The first phase decides where to place spilling instructions (`load` and `store`) so that a second phase that assigns registers to variables will not generate additional spill code. For that to be possible, register-to-register copies (`move` instructions) may need to be inserted. The underlying assumption that makes such a decoupling efficient is that `move` instructions are more likely to be cheaper than memory transfers. Decoupled register allocation

is often associated with static single assignment (SSA) form [37] as, in strict SSA, the way live-ranges are split, explicitly, makes the second phase always feasible. This is when all variables can be mapped to any register. The case of precoloring and register aliasing is more complex as we will see.

Many recent register allocation algorithms follow such a *decoupled approach*, see for example [3, 57, 58, 84, 85, 93, 95, 104]. This model has important advantages. First, the separation between these two phases yields simpler and more modular implementations: different spilling heuristics can easily be combined with different register assignments. As an example, about 20% of the lines of code of the machine-independent code generator of LLVM [68] are exclusively related to register allocation. Thus, from an engineering point of view, it is interesting to design register allocators that are modular. Second, the local register pressure, a property that is easy to infer in decoupled designs, simplifies other compiler optimizations, such as redundancy elimination, and code analysis.

1.2 Motivations

Decoupled register allocation is an elegant approach to a complex problem. Its inherent qualities make it appealing for modern compilers. The feasibility of this approach has been well studied in the past few years, in particular by Hack [55], Pereira [83], and Bouchez [15]. In this thesis, we wanted to go beyond the feasibility aspects by proposing efficient solutions that may be applied to JIT compilation. Moreover, we wanted to address some pending questions. In particular, we focused on the following points.

A first aspect concerns the spilling phase. As already stated, SSA form is usually used in these allocators to ensure that, once the register pressure is low enough, graph coloring can be used for register assignment without additional spilling. However, how to perform the spilling phase itself, i.e., how to place `load` and `store` instructions, was not completely understood. In particular, the question we had was: *does SSA help for spilling?* We wanted to evaluate the impact, positive or negative, of SSA on the spilling model and the quality of the generated solution to derive good spilling heuristics.

A second aspect concerns the register assignment phase. As compilers are more and more embedded in the user environment, we wanted to supply fast and lightweight algorithms for register allocation. One of the questions we had was: *is it possible to use the elegant formalism of decoupled approach to derive fast algorithms?* This was clear in a simplified model, but less clear in the context of actual machines. Indeed, as Hack [55] showed, specific constraints of the instruction set architecture (ISA) can make the decoupling between the two phases more complicated. In some extreme cases, even extensive live-range splitting is not enough to handle complex ISA constraints. Moreover, it may even not be possible or desirable to apply this kind of splitting, depending on the compiler/architecture. In other words, a more precise question was: *is it possible to cope with these constraints and still use the elegant formalism of decoupled approach to derive fast algorithms?*

Another assumption concerns `move` instructions and spill instructions. As already stated, decoupled register allocation assumes that `move` instructions can be inserted to spare spill instructions. The direct question that we wanted to address was: *is it really true that `move` instructions are less expensive than spill*

instructions? Because of this assumption, it is likely that the number of `move` instructions generated increases compared to a non-decoupled approach. This led us to ask: *is it possible to reduce this number of `move` a posteriori?*

1.3 Outline and Contributions

This thesis is organized in six contributions dispatched in three different parts. These parts follow the regular compilation flow of a decoupled register allocator: Part II deals with the spilling phase, Part III with the coloring phase, and Part IV with post phases. Before, Chapter 2 of Part I completes this introduction by defining all introduced notions, such as graph coloring, live-range, register pressure, and so on. To clarify the assumptions that we make, it also presents in details the assumptions made by existing decoupled register allocators and it discusses the problems induced by architectural constraints. The rest of the manuscript is organized as follows.

The first chapter of Part II, Chapter 3, evaluates the impact of SSA on the modeling of the spilling problem. Using a newly-defined integer linear programming (ILP) formulation, we show that SSA form complicates the problem and that a naive handling of its specificities may end up in very bad cases. We then introduce two different handling of this form and demonstrate that they are sufficient to catch up the gap with non-SSA spillers. Moreover, we show that, thanks to these models, spillers based on SSA can achieve even better performances for an equivalent complexity of the implied analysis.

Chapter 4 comes back on the spilling problem but from an heuristic point of view. We review existing spilling criteria and heuristics and point out their advantages and weaknesses. Moreover, we evaluate empirically different simplifying assumptions that may help to derive simpler and faster heuristics, in particular in the JIT context. Finally, we propose a new cost model to help improving the runtime of the generated code. This chapter is the less elaborated one as it was done at the end of the thesis.

We then enter Part III, which deals with coloring. In a first chapter, Chapter 5, we give a formal model to deal with ISA and application binary interface (ABI) constraints in both graph-coloring-based and scan-based approaches without extensive live-range splitting. We introduce the concept of *antipathies* in graph-coloring-based approaches, a way to guide variables to be assigned to different registers, and describe different strategies to deal with them. These strategies require different implementation efforts, from very light to light, in existing approaches depending on the expected quality of the generated code. We define a new scan approach that takes advantages of the properties of SSA form, the *tree-scan* . We describe several methods to bias the coloring during any scan approach, including tree-scan, to limit the insertion of `move` instructions. We evaluate all our strategies in the state-of-the-art graph-coloring allocator, the iterated register coalescer (IRC) [51], and compare them to our tree-scan approach with different configurations of the bias methods. The evaluation focuses on the runtime, the compile time, the memory footprint, and the code quality with respect to `move` instructions. This evaluation includes also the latest scan-based approach, the preference-guided allocator [22]. Tree-scan proves to be a very aggressive register allocator, whose compile time and lightweight memory footprint make it appealing for JIT compilation.

In Chapter 6, we then focus on register aliasing constraints. We show how the spilling test can be modified to take into account the particularities of such constraints. We then propose a new form of live-range splitting that we called the semi-elementary form. This form allows to decouple the spilling phase from the assignment phase, i.e., without any additional spill code, without requiring the extensive live-range splitting used so far. We demonstrate the benefits of this splitting in the context of graph-coloring-based approaches, in particular in terms of compile time and memory footprint, thus improving its applicability to JIT compilation.

We then continue with Part IV, concerning post phases, i.e., optimization phases after register allocation. In Chapter 7, we extend the theoretical framework of Bouchez [15] proposed to avoid the extensive edge splitting induced, in particular, by decoupled approaches when going out SSA form. Our method, based on the formalism of parallel copy motion, turns out to be able to improve the quality of the generated code although it was not its initial goal. It defines a nice way to move `move` instructions, thanks to region recoloring. Then, Chapter 8 allows even more general region recoloring as it provides a framework to perform parallel copy motion directly on data dependence graphs (DDGs). Both methods can be stopped at any time and still producing correct code, making them appealing for JIT compilation as they can improve the code until a certain time budget is consumed.

Chapter 9 concludes this manuscript.

Note: For the experiments, STMicroelectronics provided the compiler, the associated tools, e.g., profiler, linker, and the target processor, an embedded very-long instruction word (VLIW) media processor, the *ST231*.

Chapter 2

Prerequisites and Hypotheses

This chapter presents and details the important notions that we use in this manuscript. We first start with the program representation, defining step by step the elements that form a program and how they work together. We then introduce the static single assignment form, quickly discussing its concepts and properties as they will be essential to understand the decoupled register allocation. The next section presents the liveness, an important notion in register allocation. Then, we present the different approaches to register allocation, both global and decoupled, and in particular their hypotheses.

2.1 Program Representation

This section gathers the definitions of the notions related to a *program*, which is the input to the analysis and algorithms that we develop in this manuscript. In general, depending on the compiler, the input may be a source file, a complete application, a trace, etc. Whatever the input form is, the compiler *front end* translates it into an *intermediate representation (IR)*. The choice of the IR depends on the goals of the compiler; a given IR facilitates some operations but may complicate others. In our case, we deal with a low-level description of a function or procedure represented by a *control flow graph (CFG)*, which abstracts *basic blocks* and *instructions*, as defined hereafter.

2.1.1 Code Operations

Basic Block A basic block is a sequence (in general maximal) of instructions with only one entry point and one exit point. Each block is assigned a *frequency* that represents how many times it is executed exactly or as an approximation. This information can be obtained by profiling or heuristics [5].

Instruction An instruction, also called *operation*, takes a list of *arguments* to perform a computation, according to its *label* (e.g., move, add, jump, function call), and stores the *results* in a list of *definitions*. The number and the type of the arguments/definitions depend on the computation. We will use the terms *temporaries* or *variables* to denote arguments and definitions that may be assigned to a register, i.e., that are *allocatable*. For instance, the label argument

of a `goto` instruction is not allocatable. From this point, unless it is specified, the definitions and arguments terms refer to the definitions and arguments that are allocatable. In this thesis, the arithmetic semantics of instructions is not relevant. However, will be of particular interest the following instructions:

- `move`: copy a variable to another one.
- `store`: copy a variable to a memory location.
- `load`: copy a memory location to a variable.
- `jump`: create a control flow to another basic block.
- `call`: jump to another function and return.

Allocation Constraints In this thesis, we focus on reduced instruction set computing (RISC) architectures. In such a configuration, all the instructions, but special ones, use at most two arguments and define at most one result. This representation is called 3-address code. Moreover, all definitions and arguments must be in register when they are defined or used. On the other hand, complex instruction set computing (CISC) architectures offer the capability to use or define a variable directly from/to a memory slot but they have constraints on the usage of the instruction set that depend on the target processor. On x86, for instance, the number of operands that can reside in memory for a given instruction is limited to one. Moreover, on such architectures, every instruction has only two operands: one read-only and one read-write. Such a representation is called 2-address code.

For some instructions, a special processing is needed to cope with constraints coming from the hardware. There are mainly two kinds of such instructions. The first kind is instructions that use their operands implicitly. The location of these operands is defined by the application binary interface (ABI). For instance, the ABI of the ST200 family specifies that the first argument of call instructions is in register 16, the second in register 17, and so on. The second kind is pseudo (or virtual) instructions, i.e., instructions that do not exist on the architecture. These instructions are translated by the compiler into a sequence of actual architecture instructions. For example, STxP70 has no division instructions. In general, these translations are performed before register allocation. However, some of them will be introduced during or just before register allocation and will need to be translated after. This is the case of the *parallel copy* (see below) and of the *ϕ-function* (see Section 2.2.1).

Parallel Copy Parallel copies are virtual instructions that perform multiple `move` instructions *at the same time*. The `move`s represent the propagation of values performed by the parallel copy. The parallel semantics is fundamental, since performing `move`s in a sequential way with no care may cause a value to be erased before being copied to its proper destination, variable or register.

More formally, a parallel copy, denoted $(d_1, \ldots, d_n) \leftarrow (a_1, \ldots, a_n)$, assuming that all d_i are different, performs in parallel the n copies $d_i \leftarrow a_i$, which performs a `move` of variable a_i into variable d_i. A parallel copy can be represented as a directed graph, whose vertices are the variables involved in the parallel copy and there is an edge from a_i to d_i for each i. A particularity of this graph is that the in-degree of all vertices is at most 1 (such a graph is called windmill [92]). A parallel copy contains a *duplication* if it exists $i \neq j$ such that $a_i = a_j$, i.e., if its graph representation has a node with an out-degree at least 2.

A parallel copy contains a *cycle* if so does its graph representation. A parallel copy is *regular* if its graph representation is a chain. A parallel copy is *cyclic* if its graph representation is a single cycle. A parallel copy is *reversible* if its graph representation is a disjoint union of chains and simple cycles, i.e., if it is the union of regular parallel copies and cyclic parallel copies (Spartan parallel copy [86]), in other words, if it has no duplication. It can be completed into a permutation.

Such instructions have to be eliminated prior to the end of the compilation process, since they do not exist on actual architectures. The elimination process consists in mapping the parallel copies into a sequence of `move` or `swap` instructions [13, 86]. If `swap` instructions are not available, this process needs an additional variable in case the parallel copy is a union of (disjoint) cycles.

Parallel copies are a key structure for decoupled register allocation as we will show in Section 2.3.

2.1.2 Control Flow Graph (CFG)

The control flow graph is the object that abstracts the structure of the program, i.e., the basic blocks and the way the control flows between them.

General Structure A control flow graph $G = (V, E)$ is a directed graph where nodes or vertices (V) represent basic blocks and where edges (E) represent the possible control flow between basic blocks. The source of an edge is called the *source block*, its destination the *destination block*. The edges that flow in (resp. out) a basic block are its incoming (resp. outcoming) edges. For a given basic block, the source blocks of its incoming edges are its *predecessors*, and the destination blocks of its outcoming edges are its *successors*.

A node represents an *entry block* if it has no predecessor and an *exit block* if it has no successor. These represent the possible starting points (resp. ending points) of the execution of the program (typically a function). From now on, we assume that there is only one entry block and only one exit block. If not, we create a virtual entry (resp. exit) block, predecessor of all entry blocks (resp. successor of all exit blocks).

Edges have a probability, which can also be profiled or heuristically estimated. This information can be combined with the frequency of the source block, to obtain the frequency of the edge. An edge is *critical* if its source block has several successors and its destination block has several predecessors. Algorithms for removing critical edges are standard [4], when it is possible.

Figure 6.1(a) (Page 117) shows the CFG representation of a program.

Loops and Back-Edges A cycle in the CFG corresponds to a "loop" (a cyclic behavior) in the program. Such loops are worth to mention because they usually represent the hot spots of the applications, i.e., the most executed parts. The way loops are structured, in particular how they are nested, has to do with the theory of *natural loops*, *reducible graphs*, and *loop nesting forests* [90]. We do not intend to develop this theory here, just to recall intuitive notions.

A *back-edge* is defined, from a depth-first search (DFS) traversal of the CFG, as an edge (u, v) such that u is first visited in the traversal issued from v. By construction, the graph obtained by removing all back-edges from the CFG is

acyclic. If for all back-edges (u, v), v *dominates* u, i.e., all paths from the entry node to u traverse v, then the CFG is said *reducible*. In this case, a "natural" notion of loops can be defined as follows. Each node v that is the destination of a back-edge is the *loop header*, i.e., the entry, of a loop. If (u_1, v), ..., (u_n, v) are the back-edges leading to v, then the *body* of the loop is composed by all nodes that belong to a path from v to one of the u_i. The definition of loops in an irreducible CFG is not unique and relates to loop nesting forests [90].

Reachability and Program Point If there is a path from u to v, we say that u *reaches* v or v is *reachable* from u. By definition, this terminology applies to nodes of the CFG, i.e., basic blocks. We extend it in a natural way to instructions following the sequential order of instructions within a basic block.

During register allocation, compilers may insert instructions, mainly `move` and spill (`load`/`store`) instructions. The insertion happens between existing instructions, possibly on edges of the CFG. A program point denotes such a place, as illustrated in Figure 3.10.

Reverse Post-Order Traversal As for arbitrary directed graphs, there exist several ways to walk through a CFG. The reverse post-order (RPO) traversal has nice properties that will be used in the next chapters. This traversal orders the basic blocks as follows. First, a classical DFS is applied with postorder labeling, i.e., all children are numbered before their father. Then, the basic blocks are sorted in decreasing order of their numbering.

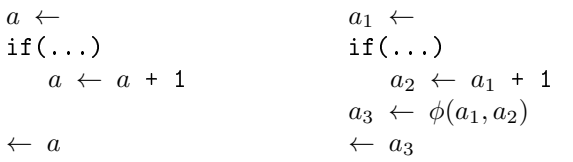
2.2 Static Single Assignment (SSA)

The static single assignment (SSA) form, or more simply SSA, satisfies the property that each variable is textually defined only once. This is a *static* property, not a *dynamic* property as a variable can be defined several times during the execution (for example in a loop). The SSA form was introduced in 1988 as an efficient support for some optimizations [1, 94]. Its foundations as well as the algorithms to build it were provided in 1991 [37]. How to translate out of SSA is detailed in [13].

2.2.1 ϕ-Functions

The single definition property is not achievable just by renaming the variables. Indeed, with renaming only, all the definitions that reach a given use (there is a path from the definition of the variable to the instruction that uses it) must share the same name, thus breaking the single definition property. For instance, this occurs for a loop counter defined both inside (increment) and outside (initialization) of the loop. To tackle this problem, SSA introduces special instructions called ϕ-functions.

ϕ-functions are placed at the first program point of a basic block. A ϕ-function produces one definition and uses as many arguments as the basic block has incoming edges. The semantics is that the definition is copied from the argument whose index equals the index of the incoming edge. When several ϕ-functions are placed at the start of a basic block, they are assumed to be performed in parallel. More formally, let B be a basic block with m incoming edges



(a) Original program (b) Program under SSA

Figure 2.1: The SSA representation. Operands on the left (resp. right) hand side of the ← symbol are definitions (resp. uses).

and a set of ϕ-functions, $\{d_1 \leftarrow \phi(a_{11}, \ldots, a_{1m}), \ldots, d_n \leftarrow \phi(a_{n1}, \ldots, a_{nm})\}$. These ϕ-functions are equivalent to placing a parallel copy on each incoming edge of B, where the i^{th} edge carries the parallel copy $(d_1, \ldots, d_n) \leftarrow (a_{1i}, \ldots, a_{ni})$.

Figure 2.1 presents a program with and without SSA. The definitions of a in the original program are renamed with a_1 and a_2. A ϕ-function is created to choose the right definition for the last use of a.

2.2.2 Strictness and Dominance Property

A program is *strict* if, for each use of a variable, all the control-flow paths from the entry of the program to this use traverse a definition. In other words, whatever the path used to reach an instruction, all its arguments have been defined. A non-strict program can be translated into strict SSA form by adding in the entry block, before constructing the SSA form, a dummy definition for all arguments that do not stick to the strict rule. Hence, from this point, we will only consider programs in strict SSA form.

An interesting property of strict SSA programs is that the definition of a variable *dominates* its uses, i.e., as already stated, every path from the entry to each use traverses the definition. We say that a node d *strictly dominates* a node u if d dominates u and $d \neq u$. A node v is the *immediate dominator* of u if v strictly dominates u and there is no node w such that v strictly dominates w and w strictly dominates u.

2.2.3 Conventional SSA

A program is in *conventional static single assignment (CSSA)* form if replacing, for all ϕ-functions, all operands (definition and arguments) with the same name does not change the semantics of the program. This property can be very useful to simplify some process, e.g., deconstructing SSA. However, all programs may not be in CSSA. In particular, this property is easily broken by copy propagation or code motion optimizations. When this property is needed, there are several algorithms to translate from SSA to CSSA [100, 11]. We point out however that this translation may impact the register allocation as it may insert `moves` and may create new variables.

2.2.4 Deconstructing SSA

Deconstructing the SSA form may not be as simple as it seems to be at first glance [24, 100]. This problem has been solved efficiently, both in terms of qual-

ity of the generated code and run time of the compiler [13]. Nevertheless, these approaches have been designed to work on codes that are not yet register allocated. On register-allocated codes, the single definition property is obviously broken as a register may be reused several times. Moreover, these approaches may create intermediate values, which will have to be allocated again, potentially causing new spill code.

For register-allocated codes, simple processes are usually applied, which rely on strong assumptions. For example, Hack [55] relies on the fact that `move` and spill instructions can be placed on the CFG edges (i.e., a basic block can be inserted), which makes the translation straightforward. On the other hand, Pereira and Palsberg approach [86] requires the program to be in CSSA form prior to coloring, that it does not have critical edges, and that the target architecture is able to perform `swap` instructions. Then, the deconstructing process, after coloring, places the parallel copies implied by ϕ-functions on the predecessor blocks. Their assumptions make this process easier. In particular, the CSSA form ensures that the parallel copies generated by the ϕ-functions have no duplication. See also Section 2.3.3.2 for a more detailed discussion related to the liveness of variables and to critical edges.

2.2.5 Liveness and SSA

A variable v is said to be *alive (or live) at a program point* p, if p belongs to a path from a definition of v to one of its uses. For a given variable, these program points form its *live-range*, as illustrated by the vertical bars in Figure 5.6 (Page 103). To make things simpler, we assume that all definitions are used. Considering a node, e.g., a basic block or an instruction, all the variables alive at the entry (resp. exit) point of the node are the *live-in* (resp. *live-out*) variables. Variables that are both live-in and live-out of a node and not defined within this node are called *live-through*. We often refer to these variables as the live-in, live-out, or live-through sets. Liveness information is usually determined using a backward data-flow analysis [36]. There are more efficient algorithms that use the property of SSA form to build live-in and live-out sets [12] as well as to develop fast queries to determine if a variable is live at a given point [14].

The liveness of operands of ϕ-functions have to be defined with care depending on where the implicit parallel copies will finally be placed. If we strictly stick to the semantics of ϕ-functions, which places copies on the incoming edges, each argument of a ϕ-function in block B is live-out of the related predecessor block of B but not live-in of B (unless further used) and each definition is live-in of B. If critical edges cannot be split (i.e., if a basic block cannot be inserted on the edge) or if jump instructions have allocatable operands and the copies need to be placed before such a jump, the liveness of the operands of ϕ-functions needs to be defined carefully. In our case, unless otherwise specified, we assume the standard liveness of ϕ-functions as stated above, i.e., with copies on the edges.

The notion of liveness defines the positions where a variable must be available in the *storage resources* (register or memory). Determining if two variables can share the same storage resource requires to know the exact behavior of the program (values of variables and execution paths), which is not possible. To approximate these constraints, the simplest way is to use liveness information. We say that two variables *interfere* (cannot share the same storage resource) if they are simultaneously live at some program point. Chaitin et al. [33] introduced a

particular case to relax this definition of interference: a and b interfere if and only if either a is live just after a definition of b and this definition is not a `move` from a to b, or the converse (inverting a and b). However, if b and c are simultaneously live and are both a copy of a, there is still an interference between b and c. In strict SSA, it is often assumed that all `move`s are removed by simple variable name propagation while the implicit `move`s induced by ϕ-functions are not analyzed. In this case, the two definitions are equivalent. Thus, in this thesis, unless otherwise specified, we assume that two variables interfere if and only if they are both simultaneously alive. This leads to the notion of *register pressure* at a program point p, which is the number of variables live at p. We will see in Section 2.3.3.2 that care has to be taken to make sure that the maximal register pressure corresponds to the *register need*, i.e., the number of register needed to allocate the variables.

2.3 Register Allocation

This section sets the hypotheses we make on register allocation. It also provides a quick view of related work, which is further discussed in each chapter according to the related point of view.

As already stated, register allocation is the problem of mapping the unbounded number of variables of a low-level representation of the program to the limited number of registers. When the registers are not sufficient, the memory, or more generally a *spilling destination*, has to be used. Thus, there are two main problems to address during register allocation:

Spilling Which variables should be evicted into memory and where the related `load` and `store` instructions should be placed.

Assignment Which register should be assigned to each variable.

There are two ways of dealing with register allocation. Global approaches perform register allocation with a single algorithm, i.e., they solve both the assignment and spilling problems in one integrated phase. On the other hand, *decoupled approaches* split this process into mainly two independent phases: spilling then assignment. This design is particularly interesting as it yields more modular and more specific optimizations to each phase implementation. The degree of independence of each phase depends on some assumptions, in particular regarding the insertion of `move`s and the architectural constraints, as discussed in Section 2.3.3. Despite the fact that these approaches are not optimal in the general case, they perform well in practice and in particular compared to global approaches as demonstrated by Koes and Goldstein [66].

2.3.1 Hypotheses

In this section, we discuss several aspects that may change the problem of register allocation. We fix the model we consider and our hypotheses for each aspect, for the whole manuscript.

Instruction Selection Register allocation considers mainly two storage locations: memory and registers. This is not uncommon that architectures feature several register files, i.e., independent sets of registers used for different purpose,

e.g., floating point registers, general purpose registers, etc. Some instructions, e.g., additions, may be available over several types of register files and it may be equivalent to use one or another. Hence, depending on the actual registers usage, it may be interesting to adapt the instruction to avoid spilling or make a better use of all register files. We will not consider this option in this work, i.e., we assume *a fixed instruction selection.*

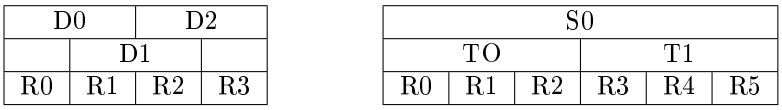
Spilling Destination In the spilling problem, the storage resource where the variables are evicted, i.e., the spilling destination, is assumed to be unique. In fact, a register allocator targeting an architecture featuring several register files could choose, thanks to `move` instructions between them, to spill some variables into another register files instead of spilling to the memory. Thus, it may be possible to use different spilling destinations to spill a variable as Lu et al. showed [72]. We choose not to do so, i.e., we assume that *the spilling destination is unique.*

Amount of Storage Location for Spilled Variables In general, the memory is used as the spilling destination. As it comes usually in far more amount than registers, it is assumed to be unlimited. In fact, a register file may not be directly moveable into the memory. This is the case for the branch registers on *ST231* architecture for instance, where the spill code is performed into general purpose registers. In such a case, the spilling destination comes in very limited amount. To cope with this problem, we assume that we can derive an order in which the allocations to the different register files can be processed without creating a new problem instance for an already-solved register file. For instance, for *ST231* architecture, we solve register allocation for the branch registers, then for the general purpose registers. Thus, spill locations created during the register allocation of branch registers are variables that are allocated during the processing of general purpose registers.

Thus, we make the following two assumptions: we assume that *we can allocate each type of register file independently following a predefined order* and that *the storage location where spilling is performed is unlimited.* In other words, in this thesis, we discuss problems with only one type of register file in mind. The proposed method can then be applied successively to each register file, adapting the spill cost to the unique spilling location.

Aliasing There exist architectures where the addressing of a register file is not unique, i.e., the same chunk of a register file can be accessed through different registers. These registers are said to *alias*. In such a configuration, we say that register allocation has to deal with *register aliasing*. In an *aliasing pattern*, i.e., the way registers alias within a register file, a *level* is defined by the sets of registers that have a specific bitwidth. Each level is numbered from the smallest to the largest bitwidth. In such a numbering, the first level is called the *atomic level* and its registers are the *atomic registers*.

The first assumption we make is that *registers at the same level do not alias*, i.e., we focus on aliasing patterns involving different bitwidths. We also restrict to a special form of aliasing. Figure 2.2 illustrates different patterns. When all registers at level l are composed by a contiguous number of registers of level $l-1$ and l covers completely $l-1$ (the level $l-1$ is a partition of the level l),



(a) Arbitrary aliasing pattern (b) Hierarchical aliasing pattern

Figure 2.2: Example of aliasing patterns. In this example, the arbitrary pattern (a) has an unaligned register, D1, at level 1. Moreover, this register aliases with D0 and D2. Hierarchical aliasing pattern (b) has no aliasing register within a level. Moreover, a level l is composed by all the elements of level $(l-1)$.

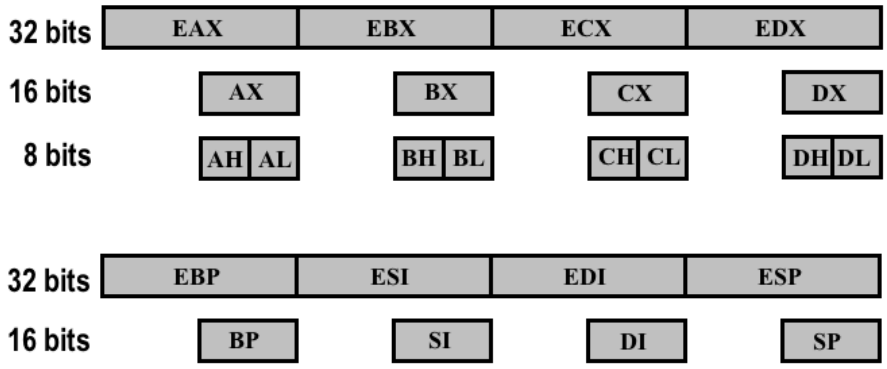


Figure 2.3: General purpose registers of x86 architecture. Due to encoding constraints, each level can address only 8 registers.

the aliasing pattern is said *hierarchical*. This pattern is used in several common architectures, like x86 and ARM. To strictly stick to the definition of an hierarchical aliasing pattern, the aliasing register files may need to be completed with non-allocatable registers. For instance, this is the case of the x86 architecture, see Figure 2.3, taken from Pereira and Palsberg [86]. This architecture uses 3 bits, i.e., it has 8 "names", to encode each access to a register for each operand. As it features eight 32 bits registers, it should have sixteen 16 bits registers. However, due to the encoding constraints, only 8 of these 16 registers are addressable. Thus, holes in the aliasing pattern appear. Nevertheless, we consider that this pattern is still hierarchical, since it can be filled with non-allocatable registers to match the definition. As a side remark, non-allocatable registers are not taken into account in register allocation, thus it is not necessary to explicitly add them. To summarize, in this thesis, *we consider only hierarchical aliasing.*

Instructions Scheduling It is well known that register allocation is impacted by the schedule of the instructions. A different schedule may require fewer registers. However, for most of our work, we do not take this opportunity into account. In other words, *we assume a fixed schedule of the instructions, unless otherwise specified.*

2.3.2 Global Register Allocation

2.3.2.1 Graph-Based Approach

In 1981, Chaitin et al. [33] introduced graph coloring as a global approach for register allocation. It was the first method that dealt with a complete function. Previous approaches were limited to one instruction (more precisely a tree of arithmetic operations) [97] or one basic block [46, 61]. Chaitin et al. approach relies on a clean and simple formalism, thus, making it appealing to use. The significance of this method can be seen in the number of publications related to register allocation via graph coloring, see [25, 8, 26, 35, 23] to quote but a few.

The General Problem Graph-coloring approaches build an undirected graph (V, E, A, w), the *interference graph (IG)*, where V is a finite set, $E \subseteq V \times V$, $A \subseteq V \times V$, and w is a function from A to $\mathbb{N}$. The set of nodes V represents the variables, the set of undirected edges E represents the *interferences* between variables, and the set of undirected weighted edges A represents the *affinities* between variables. If $(u, v) \in E$, u and v interfere, which means that they cannot share the same register. The neighbors of a node u are the nodes connected to this node by an interference, i.e., $\{v \mid (u, v) \in E\}$. The number of neighbors of u is its degree. An affinity $a = (u, v)$ means that u and v are connected by a `move` instruction in the program. Its weight $w(a)$ represents the gain of removing the related `move`, thus assigning the same register to both u and v. Figure 4.9 (Page 74) presents a program and its related interference graph.

Graph coloring consists in finding a function that maps each node to a color, so that two nodes connected by an edge (of E, i.e., an interference) do not share the same color. The related optimization problem consists in finding a mapping function that uses as few colors as possible, i.e., that computes the chromatic number of the graph. For register allocation, the number of possible colors is fixed by the number of registers, say k, so it is more related to the corresponding decision problem: is a graph G colorable with at most k colors, i.e., is it k-colorable? In other words, is the chromatic number of G at most k? This k-colorability problem is well-known to be NP-complete for arbitrary graphs and $k \geq 3$ [50, Problem GT4]. Moreover, Chaitin et al. [33] showed that, given an arbitrary graph G, it is always possible to build a program whose interference graph is G. In other words, for an arbitrary program, deciding whether k registers are sufficient to register allocate, through this graph-coloring formalism, a program without any spill is NP-complete. This motivated the use of a heuristic, based on a simplification scheme, to perform register allocation.

This simplification scheme is an old concept introduced by Kempe in 1879 [63]. It relies on the worst possible coloring of the neighbors of a node, worst in the sense that the neighbors are considered to use as many colors as possible. Using this idea, a node u can be safely removed from the graph if it has at most $(k-1)$ neighbors. In the worst case, each neighbor uses a different color, so at most$(k-1)$ colors are consumed by the neighbors of u. Thus, once its neighbors have been colored, it always remains at least one color to color u, whatever the coloring of its neighbors. Using this simplification process iteratively orders the nodes from less to most constrained ones. Then, a valid mapping function can be obtained by iteratively reintroducing the most constrained node and choosing a color compatible with its current neighborhood.

If the simplification process gives an order for all nodes, the coloring phase finds a valid solution by construction. In such a case, the graph is said to be *greedy k-colorable*. Of course, not all k-colorable graphs are greedy k-colorable.

What happens if a graph is not greedy k-colorable, i.e., if the simplification process is blocked because all remaining nodes have a degree at least k? That is where spilling comes into play. In 1982, Chaitin [32] proposed an heuristic to spill using the IG, which consists in deleting a node from the graph, based on a cost function. This principle has the effect of completely ignoring the live-range of the variable associated to this node, as if it was always stored in memory. For this reason, it is called *spill everywhere*. After such spills, the simplification can possibly proceed again, then the coloring of non-spilled nodes. However, on a RISC architecture for example, the arguments and definitions of instructions must reside in registers. A variable can thus never be spilled everywhere. To cope with this approximation, spill code is inserted for spilled variables at the end of the simplification process. To be as close as possible to the spill-everywhere approximation, the live-range parts that remain in registers are made as short as possible. Thus, `store` instructions, the instructions that copy the value from register to memory, are placed immediately after each definition of the variable whereas `load` instructions, the instructions that copy the value from memory to register, are placed immediately before each use of the variable. This creates new variables. Then, the IG is rebuilt and the simplification process redone. This is as long as no more spill code has to be inserted (in general twice, sometimes 3 times). To our knowledge, all graph-coloring-based register allocators use such a spill-everywhere heuristics, even if the placement of `load` and `store` instructions can also be re-optimized afterwards.

The Coalescing Problem Regarding the elimination of `move`s, graph-based allocators offer a natural way to model it. The `move`-related variables are connected by affinities, *coalescing* them means imposing that they are assigned the same color, which can be done by *merging* the two corresponding nodes. One possible optimization problem is then to merge as many affinity-related nodes as possible so that the sum of the weights of the coalesced affinity edges is maximal. This problem, known as *aggressive coalescing*, is NP-complete [15, Ch.5]. If too aggressive, this kind of coalescing may increase the chromatic number of the IG. Indeed, by fusing the live-ranges, merged variables may interfere with more variables. The first algorithm proposed by Chaitin et al. used this coalescing scheme prior to coloring.

On recent architectures, memory accesses and thus spill code are more likely to be more expensive that register-to-register `move`s. Moreover, as it was observed, aggressive coalescing may increase the number of spilled variables, but the opposite is also true: inserting variable-to-variable copies, i.e., splitting live-ranges, may help reducing the chromatic number of a graph [41]. To take advantage of these observations, another kind of coalescing, *conservative coalescing*, was introduced by Briggs et al. [26]. The optimization problem of conservative coalescing is similar to the aggressive coalescing, except that coalescing an edge should never increase the chromatic number of the graph. Deciding whether a coalesced graph is k-colorable with at most a given number of affinities not coalesced is NP-complete [15, Ch.5]. A simplification of that problem, called the incremental conservative coalescing, considers the affinities one by one and

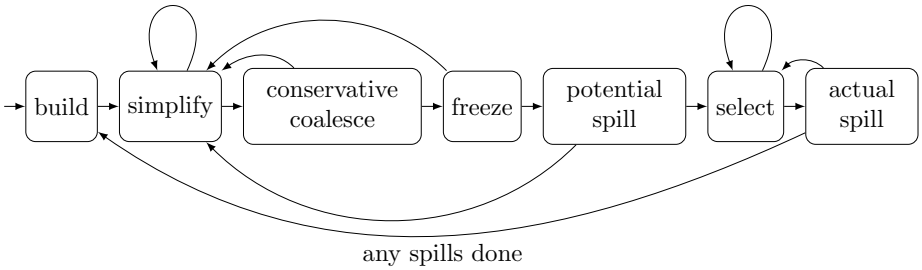


Figure 2.4: Iterated register coalescer from Figure 5 in George and Appel [51].

merge both nodes if and only if it preserves the k-coloring property. This variant is also NP-complete in the general case. Briggs et al. [26] proposed an heuristic for that problem. It is based on some properties of the nodes to be coalesced, which ensure that the coalesced node will be simplifiable at some point of the simplification process. This test is known as Briggs' rule. On the other hand, Briggs showed in his thesis [23] that performing, as a pre-phase, aggressive live-range splitting, i.e., inserting a variable-to-variable copy on each program point, gives mitigate performances. Indeed, as coalescing is heuristic-based, it may perform quite bad on large graphs. Therefore, in his allocator, live-range splitting was not used or only in a very limited fashion.

Following the incremental conservative coalescing idea, George and Appel [51] modified Chaitin's approach [32], using the improvements of Briggs et al. [26], to create the iterated register coalescer (IRC) allocator. This allocator is depicted in Figure 2.4. Its name comes from the fact that it iterates on the different phases, the simplifying (removing a node with at most $k-1$ neighbors), coalescing (merging the two extremities of an affinity), and spilling (removing a node with at least k neighbors) phases, as long as each can be performed (in this order). Nowadays, this allocator is considered to be the state-of-the-art register allocator for graph-based approaches. A new conservative test, known as George's rule, was also designed.

In 2004, Smith et al. [99] generalized the notion of degree (i.e., number of neighbors in terms of interferences) of a node to deal with register aliasing. Their generalization can be applied to any graph-based allocator using the simplification scheme. We will come back to this technique in Chapter 6. For completeness, we can also mention that, in 2002, Scholz and Eckstein [96] modeled the graph coloring problem using partitioned Boolean quadratic programming (PBQP). This model features a complete graph coloring approach, i.e., integrating spilling, coloring, and coalescing, for CISC architectures. One year later, Hirnschrott et al. [60] evaluated a classical graph-coloring approach versus an optimal approach also based on PBQP.

Criticisms Graph coloring is not exactly register allocation as Bouchez et al. [20] clearly pointed out. A first weakness is that the underlying CFG is completely obfuscated. In particular, the spill-everywhere strategy does not exploit any smart placement of `load` and `store` instructions, and cannot exploit the structure of the CFG, except indirectly through global node or edge weights.

A second weakness is that IRC-like allocators can generate useless spill code,

unless the allocator could check that the variables selected to be spilled will indeed help coloring. This is because the choice of spilled variables depends on a coloring criterion, which is related to an NP-complete problem for arbitrary graphs. This is also because the heuristic is greedy. To reduce this weakness, the concept of *potential spill* was introduced: if a variable, selected to be spilled, can nevertheless be colored in the coloring phase, it is not spilled. But still, the spilling problem, intermixed with the coloring problem, remains not well-understood and hard to capture in a graph-based register allocator.

The third limitation is that, by nature of the graph-based model, each variable is assigned a unique register whereas moving a variable from a register to another can help the coloring. Some attempts were made to circumvent this limitation [31, 78, 64], introducing various live-range splitting capabilities.

Regarding the runtime of the compiler itself, graph coloring approaches are known to induce a large memory footprint. Moreover, the iterative process implied by the spill code insertion leads to a waste of compile time. Indeed, between two iterations, the liveness information, the simplification order, and the graph structure have barely changed. Nevertheless, everything is redone.

2.3.2.2 Other Approaches

Graph coloring is not the only way to tackle register allocation. In 1990, Chow and Hennessy [34] proposed their priority-based allocator. In their model, live-ranges of the variables are in memory and they try to bring them back into registers, based on the priorities of the related live-ranges. The priorities are computed from program estimation, basically the frequency of the basic blocks, and machine parameters. When a live-range cannot entirely fit into register, it is split and new live-ranges are sorted in the priority list accordingly. When it fits, they choose the color that maximizes the number of neighbors having this color in their forbidden set.

Several "optimal" formulations were also proposed. In 1996, Goodwin and Wilken [52] proposed to solve the global register allocation problem using an integer linear programming (ILP) formulation. Their approach assigns the registers to variables and features `load`/`store` optimization, i.e., the optimization of the placement of `load` and `store` instructions, and coalesces existing copy instructions. However, it does not split the existing live-ranges. Two years later, Kong and Wilken [67] extended this formulation to deal with CISC architectures and their irregularities, in particular the 2-address code constraint. In 2002, Fu and Wilken [48] speeded up the resolution of Goodwin and Wilken's formulation. They took advantage of the structure of the program to remove redundant ILP constraints in the equations. More recently, in 2006, Koes and Goldstein [65] performed register allocation using a multi-commodity network flow model. Their expressive model, which relies on the program structure, optimizes the flows of variables from their definitions to their uses, minimizing the spill cost. Their approach can also be used in a decoupled fashion. We will study and develop such optimal models in Chapter 3.

All approaches presented in the previous paragraph rely on ILP and look for optimality in their respective model. However, in late 90s, scan-based approaches (i.e., allocators that work directly on the program, traversing instructions) appeared to cope with new compiler constraints implied by just-in-time (JIT) compilation. Poletto et al. [88], Traub et al. [103], and Poletto and

Sarkar [89] introduced *linear scan*. This allocator considers the linearization of the program as a unique large basic block and assigns variables to register or memory locations from top to bottom. It is very fast but may produce poor allocated code as the live-ranges of the variables are largely over-approximated, leading to spurious spill code. In 2005, Mössenböck and Wimmer [105] improved the spill code placement of linear scan, which was until that work based on spill everywhere, and features on-demand live-range splitting. Finally, in 2008, Pereira and Palsberg [85] proposed a new type of linear scan which, unlike previous approaches, deals with register aliasing thanks to aggressive live-range splitting (at all program points) and a "puzzle-based" solving.

2.3.3 Decoupled Register Allocation

To address the criticisms of graph coloring, in particular, regarding the spilling heuristic, Appel and George [3] introduced in 2001 *decoupled register allocation*. Their spilling phase uses an ILP formulation, which uses the peculiarities of CISC architecture and performs `load`/`store` optimization. It ensures that, at each program point, no more than k variables are alive. Then, to ensure that no more spilling will be necessary during the coloring phase, they insert, for each program point, a parallel copy of all live variables at that point. This form of aggressive live-range splitting is called *split everywhere*. The resulting graph is a collection of small interference graphs, each defined by the interferences in individual instructions, all connected by affinities capturing the parallel copies. For one instruction, the live-in set produces a clique, i.e., a complete graph where all nodes are connected to all nodes, so does the live-out set. Since, after spilling, both sets have at most k variables, each variable of the live-in set that is not live-out (or the converse) has degree at most $(k-1)$ and is thus simplifiable. Then, all other variables (those that are both live-in and live-out) can be simplified too. Therefore, the final graph is greedy k-colorable, unless register constraints are involved. We will come back to that aspect in Section 2.3.3.2. To avoid mitigated results for the coloring phase as reported by Briggs [23], when aggressive live-range splitting is used, Appel and George use an optimistic coalescing approach [80] instead of the classical incremental conservative coalescing. The optimistic coalescer fuses as many nodes as possible, but unlike aggressive coalescing, it can decoalesce nodes when it does not manage to simplify the graph. Appel and George reported that this approach gives good results as, in their case, the initial graph was known to be greedy-k-colorable. To our knowledge, this is the first decoupled register allocator.

2.3.3.1 Towards SSA-Based Decoupled Approach

The decoupled approach of Appel and George [3] has two main drawbacks. First, it relies on an ILP formulation for the spilling phase. This may be an issue for compiler users regarding run time or licensing[1], for instance. Also, Liberatore et al. [71] and Farach-Colton and Liberatore [43] showed that, even on a basic block, optimal spilling with load/store optimizations is a hard problem. No good alternative was available for the whole program at that time. Second, the split-everywhere strategy was considered too aggressive (too many insertions of parallel copies) and it is a difficult problem to find good and sufficient split

[1] As ILP solver may not be free of use.

points. Therefore, despite its inherent advantages (having two decoupled phases makes the problem "simpler" to address), this decoupled approach did not gain much interest until recently.

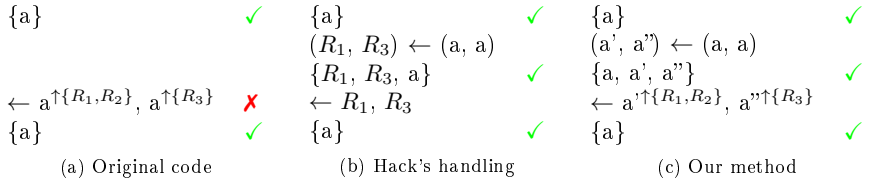
Around 2005 (at least for the publications), several groups observed that the interference graph (IG) of programs in SSA form is chordal [27, 84, 56, 16]. For such graphs, coloring with the minimal number of colors can be done in polynomial time using the simplification scheme. Also, the chromatic number equals the size of the largest clique of interferences in the graph. For programs in SSA, the largest clique is also defined as the largest liveness set (i.e., set of alive variables), over all program points, unless precoloring or aliasing is involved (see Section 2.3.3.2). Hence, SSA provides sufficient split points to enable a decoupled register allocation. The spilling phase ensures that, at each program point, the number of live variables is at most k. This property forms the *spilling test*, i.e., checking that at most k variables are simultaneously live. Then, the coloring phase uses the classical graph coloring algorithm.

In 2009, Braun and Hack [21] proposed a fast spilling heuristic for a decoupled approach, which offers a better spill code than with the spill-everywhere strategy. In 2009 too, Ebner et al. [39] presented an optimal spilling approach using a constrained minimum cut model (see again Chapter 3 for the discussion of optimal approaches). Then, in 2010, Braun et al. [22] detailed a fast coloring heuristic, also for a decoupled approach, that does not rely on graph coloring.

2.3.3.2 When Register Pressure and Register Need Do Not Match

In the previous section, we recalled the "spilling test" used in SSA-based decoupled allocators: if the register pressure (the number of simultaneously live variables) is not more than the number of available registers, the live-range splitting induced by SSA guarantees that all live-ranges can be colored, each with a single color, and with no spill. Actually, this is only partially true. Indeed, for programs having encoding or ABI constraints, i.e., for almost all programs, considering the register pressure is not enough to guarantee the colorability of the interference graph unless the compiler pre-formats the program in some way. In this section, we list the problems that can occur and how to transform the program to tackle them. Hack [55] uses the term *register pressure faithful* to express the case where the register pressure and the register need match, i.e., the register pressure is a good measure of how many registers are needed. More generally, what is important in a decoupled approach is to design a spilling test that is compatible with the coloring phase, i.e., that enables this decoupling.

Duplication Problem A first case where the register pressure may not be enough to capture the register demand of the program is when a variable must use several registers at the same time, due to architectural constraints. In such a case, we insert an explicit copy of the variable to perform a *duplication*, i.e., two variables names are now present in the program, which increases the register pressure and makes it match the register need. Figure 2.5a illustrates this problem. In this example, the encoding constraints of the instruction forces the argument, used twice, to be in two different registers. The same problem occurs when the argument is live-through and must reside in a subset of the register file that cannot traverse the instruction due to constraints on definitions.



(a) Original code (b) Hack's handling (c) Our method

Figure 2.5: Impact of duplications on the spilling test. The notation $a^{\uparrow\{R_1,R_2\}}$ indicates that the variable a should be in the subset of registers composed by $\{R_1, R_2\}$ for this use, i.e., it is *pinned* to $\{R_1, R_2\}$ for this use. Liveness sets are represented in braces for each program point, such as $\{a\}$. Implicit duplications (a) result in a non-faithful (too low) register pressure, indicated by ✗. The symbol ✓represents points where the register pressure is faithful, or more precisely, where it is a valid over-approximation of the register need. To increase the apparent register pressure, Hack [55] explicits the duplication by fixing the colors for all constrained arguments and by adding a parallel copy before the considered instruction (b). Our method (c) is similar but does not fix the color of the newly created arguments; the coloring phase will decide.

How many duplications of names need to be done to ensure that the register pressure gives the right number of registers needed? Hack addressed this question in details [55, Sec. 4.6]. If the code contains an arbitrarily-constrained instruction, it is NP-complete to decide whether there is a register allocation for this instruction where each variable is assigned to a unique register, i.e., without duplication [55, Ch.4]. In particular, determining the minimum number of copies to insert to make the register pressure faithful is NP-complete. Hack investigated a less general model where each operand is either unconstrained or constrained to a single register, and no two arguments (resp. results) are constrained to the same register (this last condition is obviously needed otherwise the coloring of the instruction is not possible unless both arguments, resp. results, always share the same value). With these hypotheses, Hack showed that the problem becomes tractable. Indeed, for such an instruction, a duplication is needed exactly for each live-through argument that is constrained to the same register as a result. However, when there is more than one constrained instruction, some live-range splitting may need to be done so that the allocations of the different instructions match. Hack showed that even with this restricted model, deciding if some split is required is NP-complete.

Based on this study, Hack proposed a heuristic approach to handle instructions with general constraints. For such instructions, the constraints are simplified by fixing the color of all constrained arguments via bipartite matching, then the same is done for the results, as explained in Figure 2.5b. Finally, a split (parallel copy) of all the live variables is inserted prior to each constrained instruction. We use a similar method, as presented in Figure 2.5c, except that we do not fix the color of the constrained operands when several choices are possible. This gives more freedom to the coloring phase. This preliminary step, with some duplications and extra splits, fixes the duplication problem. In Chapter 5, we also present a method that does not need to explicit the duplications.

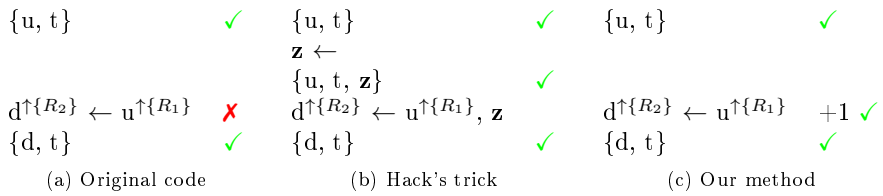


Figure 2.6: Checking live-in and live-out sets of an instruction may not be enough for an accurate spilling test. In (a), the register pressure of 2 fails to capture the over pressured point implied by ABI constraints (symbol ✗). Hack explicits this constraint with a dummy argument z, increasing artificially (on purpose) the pressure before the instruction. In our case, we check the register pressure on the instruction and add 1 due to the impossibility to recycle the color of the argument for the definition. The symbol ✓represents points where the updated register pressure is faithful.

The Encoding/ABI Problem The previous code preparation fixed the duplication problem. However, encoding and ABI constraints may still produce cases where the register demand is not accurately represented by the cardinal of the live-in and live-out sets. This is illustrated in Figure 2.6. Let us assume that, due to ABI constraints, an instruction must take an argument in R_1 and must define a result in R_2. Let us assume also that the architecture has only two registers and that three variables are involved: one argument, last used on that instruction, one definition, and one live-through variable. The set of variables live before the instruction is composed by the argument and the live-through variable. The definition and the live-through compose the set of variables live after the instruction. In both cases, the number of live variables equals 2, however it is clear that the live-through variable must be spilled. Following Hack's terminology [55, Ch.4], this example is not register-pressure faithful. To fix that problem, Hack proposed to add a dummy argument to that instruction and to define it just before the instruction. Using this trick, the dummy argument now appears in the set of live variables before the instruction and the register pressure used for the spilling test is relevant again.

We proceed slightly differently. Instead of "polluting" the program with such dummy variables, we prefer to check the register pressure on the instruction itself using the following method. We observe that this problem occurs only when a variable is last used in an instruction. Indeed, if it is not last used, it appears, with the definitions[2], in the live-out set of the instruction, thus it is correctly counted as consuming a register different than the definitions. Our method consists in traversing the set of last-used variables, restricted to variables constrained on that instructions, and trying to find a definition that can "recycle" a potentially-used color. Of course a definition can be used to recycle only one argument. The register pressure on an instruction is then given by the sum of the number of definitions, the number of last-used arguments, and the live-through, minus the number of variables that can be recycled.

In this method, an argument may be recycled by several definitions. Thus,

[2]We assume that dead definitions reach at least the live-out set of the related instruction.

in theory, if we pick the wrong recycling, we may over-estimate the register pressure on an instruction. In practice however, ABI constraints define a one-to-one mapping for all operands of the constrained instruction, thus the choice is fixed a priori. Also, encoding constraints are limited to "regular" instructions, i.e., instructions where at most two arguments and at most one definition are constrained at the same time. Therefore, an exhaustive search could be used with an acceptable cost, even if a heuristic pairing is also sufficient in practice.

Edges and Critical Edges An additional problem is due to control-flow edges in conjunction with SSA. In Section 2.2.4, we recalled that the usual way of deconstructing SSA after coloring assumes that no edge is critical. The standard semantics of ϕ-functions, as recalled in Section 2.2.5, assumes that the related parallel copies are performed on the incoming edges, in particular that variables defined by ϕ-functions are not live-out of the predecessor blocks. However, when these parallel copies cannot be placed on the edges but must be placed earlier (at the end of a predecessor block or even before a jump instruction), the liveness information computed in SSA may not describe correctly what will be obtained after copy insertion, in particular the register pressure.

This approximation is not a problem unless the number of live variables is larger than the one considered during the spilling phase. Let us examine this more carefully. Let e be a regular (i.e., not critical) edge and assume that the destination block of e contains ϕ-functions. The number of variables in the live-out set of the source block of e is at most the number of variables in the live-in set of the destination block of e, by definition of the liveness (except for variables involved in the ϕ-functions, the live-in set is the union of the live-out sets of the predecessor blocks) and the semantics of ϕ-functions (there are at least as many definitions as arguments for each parallel copy). Since e is a regular edge, the parallel copy on e can be sequentialized in the source block of e as this block has no other successor. After this transformation, the definitions are in the live-out set of the source block of e and in the live-in set of its destination block.

The previous "proof" makes an invalid assumption. It assumes that the parallel copy will be expanded last in the source block. This may not be possible if a branch instruction is required and, in this case, the register pressure may be larger before the branch instruction as some of its arguments may not be live-out of the block. Therefore, even without critical edges, deconstructing SSA may require more registers than the liveness information obtained prior to the transformation indicated. Obviously, if critical edges are allowed, the problem is even more likely to occur. In other words, for this process to be correct, the assumption must be that all edges are splittable, i.e., one can interleave a basic block between the destination block and the source block of all edges. We will see in Chapter 7 how to avoid this aggressive edge splitting without changing the assumption during the spilling phase. Another possibility is to compute liveness information more carefully, taking into account the actual places where parallel copies will indeed be placed.

Register Aliasing When register aliasing, as defined in Section 2.3.1, comes into play, the split points defined by the SSA form may not be sufficient to color the live-ranges, with a simplification scheme, without additional live-range splitting. Indeed, in such a configuration, the coloring problem (i.e., each live-range

is mapped to a unique color) is NP-complete [9, 69]. The problem is not that the spiller gets wrong but more that the coloring phase cannot be guaranteed to find the optimal coloring. Using another form of coloring may solve this issue, e.g., Pereira and Palsberg's puzzle solver for some specific aliasing constraints [85]. In Chapter 6, we will give such a spilling test compatible with a coloring phase that uses graph coloring in presence of aliasing.

In conclusion, as already stated, what is important is that the spilling test and the coloring phase are compatible, i.e., that the spilling phase, driven by the spilling test, spills and splits live-ranges enough so that the coloring phase is guaranteed to succeed in assigning a unique register to each live-range. As we just presented in this section, except for the case of hierarchical aliasing that requires some more developments exposed in Chapter 6, standard register constraints can be taken into account by preparing the program with adequate live-range splits and duplications so that the register pressure is faithful, i.e., a right measure of the register need. Therefore, unless otherwise specified, we will not come back to this point in the manuscript. The reader must however remember that these details have to be taken into account somehow.

Part II

Spill

This part deals with the first phase of a decoupled register allocation, i.e., the spilling phase. At this stage of the compilation process, the idea is to lower the register pressure such that, at every program point, the assignment phase will be able to find a proper coloring for all the variables, without inserting more spill code. The spilling phase does not require that necessary split points for coloring are inserted prior to its processing. Thus, it is possible to spill on a program that is not in static single assignment (SSA), and to color the same program using SSA. Nevertheless, if split points exist, the spiller has to deal with them, in particular, it has to deal with ϕ-functions if the program uses the SSA form.

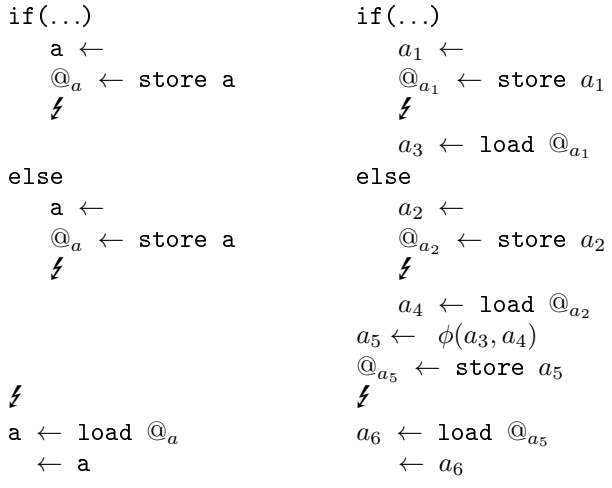
Our initial motivation was to analyze whether SSA is helpful or not to achieve good spilling. In the first chapter of this part, we investigate the impact of SSA on spilling with a flexible integer linear programming (ILP) formulation. We compare this formulation against existing optimal approaches, with respect to static spill cost, as well as heuristic-based approaches, and we evaluate the impact of SSA on the runtime of the generated code and the complexity, in terms of implementation, of the algorithms. Then, in a second chapter, using the experimental results of the first chapter, we discuss existing spilling criteria and existing heuristics. In particular, we validate experimentally several simplifying assumptions and point out the weaknesses of the approaches. We finally propose a new cost model to optimize for runtime and emphasize some of the characteristics of runtime performance that should be accounted so as to generate the fastest possible code.

In the rest of the manuscript, we use the lightning symbol (↯) to depict program points where the register pressure exceeds the number of available registers.

Chapter 3

Studying Optimal Spilling in the Light of SSA

The static single assignment (SSA) form may appear attractive for the design of spilling algorithms, because the underlying dominance tree often simplifies algorithms [21]. Also, a spill-everywhere strategy, i.e., considering than a variable is never in a register, can be realized by finding a maximal k-colorable subgraph in the interference graph, which is chordal in SSA. Although NP-complete [107, 18] (if k is not fixed), this problem may, in practice, appear simpler than for a general graph. However, considering the different SSA live-ranges, obtained from a given non-SSA variable, as unrelated means that `store`s may be needed for each spilled live-range, while only one might be enough for the original variable, as depicted by Figure 3.1. This may increase the spill cost considerably, unless the `moves` hidden in the SSA ϕ-functions are exploited.



(a) Spill in a regular program (b) Spill under SSA

Figure 3.1: In SSA, considering as unrelated the different live-ranges composing a variable may produce bad spill code.

To analyze these choices, and not just through heuristics, we needed an exact formulation of the spilling problem, as complete as possible, that exploits the structure of decoupled register allocation. As we show in Section 3.1, previous formulations either express the whole register allocation problem and are thus very expensive, or cannot express all solutions, due to some simplifying assumptions, in particular the fact that a variable cannot be stored simultaneously in a register and in memory. We thus developed a new integer linear programming (ILP) formulation to approach optimality even closer and to better understand the mechanisms involved during spilling. Section 3.2 first presents a simplified version of this formulation, which already subsumes most previous approaches, and then shows extensions that incorporate more advanced features. Section 3.3 gives a thorough analysis of the results obtained for the SPECINT 2000 and EEMBC 1.1 benchmarks and discusses the important features for optimal spilling on `load-store` architectures.

To summarize, this chapter features:

- A new simple, flexible, and expressive ILP formulation for spilling on `load-store` architectures, which allows us to accurately model variable liveness, rematerialization, SSA and `move` instructions, memory coalescing, placement restrictions of `load`/`store` operations, spill everywhere, etc.
- A detailed analysis of spilling choices that show, among others, a) the extreme importance of rematerialization, b) the difficulty of memory coalescing when `move` instructions (e.g., through ϕ-functions) are exploited, c) the strong interaction with post-pass scheduling.

3.1 Formulating "Optimal" Spilling

"Optimal" [1] spilling formulations are based on the notion of *program points* and *local* register pressure induced by a given solution to the spilling problem. They thus capture the number of live variables and their assignment to either memory or registers at a given point. Since spilling is a global problem, program points are connected according to the control flow graph (CFG) so that decisions at one point impose constraints at its neighbors in the CFG. This global model is usually expressed as an integer linear programming (ILP) instance, which is solved by a generic ILP solver, such as CPLEX or GLPK.

3.1.1 Existing "Exact" Formulations

The formulation of Goodwin and Wilken [52] models the complete register allocation problem, including the actual assignment of registers, using *live-range graphs* (LRG). A LRG models the live-range of a given variable with respect to a specific hardware register and thus needs to be instantiated once for *every* register. The initial LRGs are extended to capture spilling decisions along the variable's live range, i.e., `store` and `load` operations, register-to-register copies, and rematerialization. A major drawback of the formulation is the large size of the ILP instances. The problem stems from the duplication of the LRGs and also from redundancies arising from the LRG extensions. The approach

[1] We use quotation marks for the word "Optimal" because, as we will see, none of the exact formulations proposed so far, ours included, is able to represent all possible solutions.

thus appears rather expensive in practice. However, an optimized variant later addressed some of these issues [48].

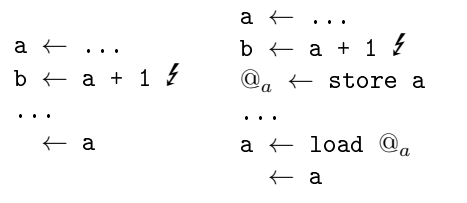
Appel and George [3] were the first to exploit the decoupling between spilling and register assignment by replacing the latter by a simpler constraint on *register pressure*. Developed for complex instruction set computing (CISC) machines, they demonstrated that this strategy considerably reduces the size of the ILP instances. However, they made a fundamental (and surprising) assumption: a variable *cannot* be stored simultaneously in memory and in a register. The problem can then be simplified by expressing, for each program point, the possible movements of a variable between the memory and the set of registers. However, this limitation leads to sub-optimal solutions, in particular to redundant `store` instructions. Indeed, each time a variable goes to memory, a `store` needs to be placed even if the variable has already been stored there in the past.

The approach of Koes and Goldstein [65] is based on multi-commodity network flow. All live-ranges are expressed using a single network-flow problem, where variables are represented by source and sink nodes, while other nodes represent allocation classes, such as constants, registers, and memory, at program points and instructions. The network capacities express constraints on the number of variables that can be assigned to each storage class simultaneously. Initially designed to solve the complete register allocation problem, including assignment, the approach can also be used to express the spilling problem alone, by merging nodes and summing their associated capacities so as to constrain the register pressure [66]. By adding some extra variables, called *anti-variables*, Koes and Goldstein avoid counting redundant `stores`. However, as for Appel et al., a variable may only be assigned to a single allocation class at any given program point. Due to this limitation, not all solutions can be expressed and the optimal can be missed too.

Ebner, Scholz, and Krall [39] address the spilling problem for SSA programs using a series of network-flow problems, one for each variable. Nodes correspond to instructions and edges to program points where `loads` can be placed. Every cut of such a network gives a solution to the spilling problem for that particular variable. To capture the register pressure, i.e., to consider all variables together, a *constrained min-cut* problem is defined by assembling nodes of the individual flow problems representing the same instruction into partitions with capacities. The placement of `stores` is not optimized: they are always inserted after the unique definition of a variable. Furthermore, the splitting of live-ranges due to SSA is kept unchanged, i.e., the implicit `moves` corresponding to ϕ-functions are not exploited.

3.1.2 Limitations of Existing Approaches

The approaches presented above were designed to solve register allocation and spilling under various constraints and assumptions stemming from complexity or modeling considerations. They thus often show slight limitations, sometimes unexpected, concerning "optimality", expressiveness, and even correctness, as we will illustrate. In the following, the symbol ↯ indicates a program point where the register pressure is too high and some variable needs to be spilled.



(a) before spilling (b) ineffective spilling

Figure 3.2: Spilling the variable `a` does not help.

3.1.2.1 Liveness

The extent of live-ranges is a surprisingly frequent source of problems, at least for load-store architectures. If a variable cannot be simultaneously in register and memory, as for the approaches of Appel et al. and of Koes et al., a variable stays live in a register *after* a use until the value can be spilled, unless the variable dies at that use in the original program. In Figure 3.2, the variable `a` remains live after its use and thus always interferes with `b`, regardless of the spilling decision. Here, `a` should be stored just after its definition and, at the same time, kept live in a register. In the worst case, these artificial interferences between uses and definitions, due to the strong hypothesis that a variable can never be simultaneously in memory and in register, may render the spilling problem unfeasible, e.g., when the number of variables defined and used is larger than the number of available registers.

A similar problem can arise with the initial formulation of Goodwin et al. [52], linked to the *block start* and *block end* transformations. Later results resolve this issue with additional ILP variables explicitly modeling *deallocation* [48].

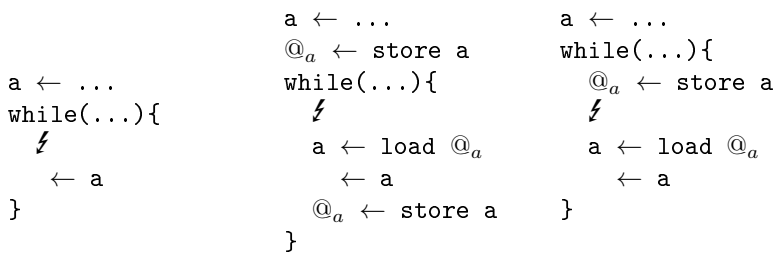
3.1.2.2 Living in Memory and Register Simultaneously

As we already mentioned, a major limitation of the approach of Appel et al. is the assumption that a variable may either be kept in memory or in a register, but never in both at the same time. Besides the unnecessary extension of live-ranges shown previously, this may lead to spurious `store` operations as shown by Figure 3.3b. The necessary `load` operation inside the loop "destroys" the previously-spilled value in memory and forces a useless `store` operation inside the loop. The "optimal" solution in their model is given by Figure 3.3c. The actual optimal solution cannot be expressed. It would consist of the code from Figure 3.3b without the `store` inside the loop.

A similar example can be built for the formulation of Koes and Goldstein, despite the fact that redundant `store`s cost zero in their model. Figure 3.4 shows three spilling scenarios on a simple loop. Redundant `store`s that are not emitted are shown in gray. Figure 3.4c is close to the optimal solution. However, a useless `load` operation is required before the loop due to the use of the variable `a` on the else-branch within the loop.

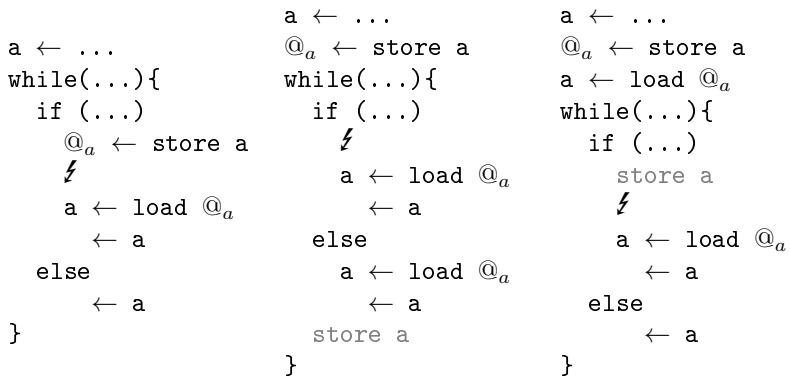
3.1.2.3 Rematerialization

It is well-known that rematerialization has a great potential to reduce spill costs by recomputing values instead of storing and re-loading them from memory.



(a) before spilling (b) Appel 1 (c) Appel 2

Figure 3.3: Spurious `store` operations following a `load`.



(a) Koes 1 (b) Koes 2 (c) Koes 3

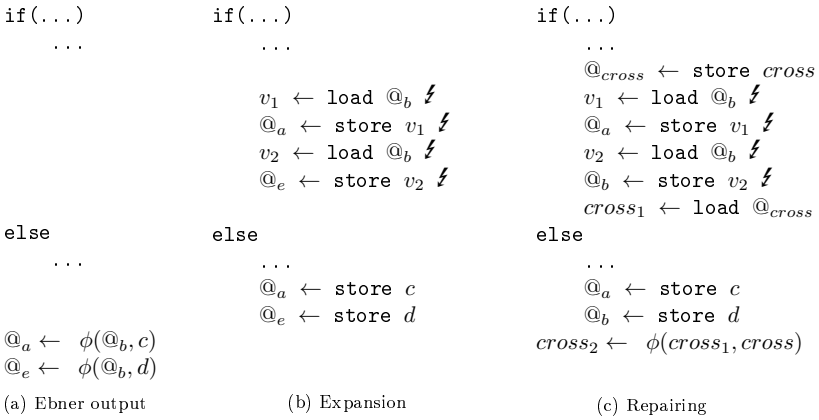
Figure 3.4: Even when redundant `store`s cost zero, sub-optimal spilling solutions may be generated.

However, in the context of "optimal" spilling, rematerialization and its impact on code quality and solving times is hardly studied.

The approach of Appel et al. does not address rematerialization. As for Koes et al., they model rematerialization of simple constants using a dedicated allocation class. Again, the fact that a variable cannot be accessed through multiple allocation classes at the same time prevents it to be used as a rematerialized operand and to stay available in memory. This may be needed for further usage after a CFG join point if the variable is not rematerializable on the other path. As for the model of Goodwin et al., it is restricted to variables holding a constant value throughout their entire live-range. Moreover, rematerializable variables *cannot* be spilled to memory. This limits the optimization since variables in non-SSA programs are often rematerializable only on parts of their live-ranges.

3.1.2.4 Memory Coalescing under SSA Form

SSA simplifies the register assignment phase but its benefits for spilling are less clear. An important aspect, not covered by any of the existing "optimal" formulations, is the modeling of ϕ-functions, in particular the effect of spilling the



(a) Ebner output (b) Expansion (c) Repairing

Figure 3.5: Hidden costs due to mixed type of operands in ϕ-functions in Ebner et al. Removal of ϕ-functions may require to insert `load`/`store` instructions. These instructions may need an additional register in case of memory to memory copy (b). This additional register may exceed the register pressure and may imply additional spill code for a variable crossing that region (c).

result and/or arguments of a ϕ-function. Ebner et al. treat them as completely-independent variables and thus do not exploit the implicit copy relations, in their cost model. Instead, they place `load`s and `store`s a posteriori, once spilling decisions of SSA variables are done. This implies a hidden cost and potentially very bad spilling decisions as illustrated in Figure 3.5. Spilling heuristics [21] usually avoid the problem by requiring the program to be in conventional static single assignment (CSSA), where the operands and the definition of a ϕ-function do not interfere. In this case, the related variables can be stored at the same memory location, without the need of additional memory operations. Otherwise, the copy relations and the possible coalescing (i.e., sharing) of memory locations among ϕ-operands have to be modeled to derive an accurate cost model [55], as we discuss in the next sections.

3.2 A More "Optimal" Formulation

This section presents a new ILP formulation of the spilling problem, more accurate than previous solutions (but specialized to load-store architectures) and flexible enough to evaluate different opportunities when designing spilling strategies. It can also *emulate* the spilling formulations given in Section 3.1 with a few additional constraints. We first present a simplified version for non-SSA programs, then describe extensions to handle `moves`, in particular those implicit in the SSA representation.

Given a program represented by a CFG, with weights indicating the execution frequency of each instruction or basic block, our formulation seeks the cost-optimal placement of `store`s and `load`s, with no other modification of the program (e.g., no re-scheduling). These spill operations can be placed on *program points* before and after every instruction. Additional program points might be available at CFG joins and splits, depending whether the CFG edges

can be split or not. An optimal solution might require to perform multiple spill operations at a given program point. Without loss of optimality, we choose to perform all `store`s first, then all `load`s, since this order reduces the local register pressure. The relative order of the individual `store`s (respectively `load`s) is not relevant and is thus not modeled.

3.2.1 Basic Formulation

For every variable live at a given program point, we record whether its value is available in a register, in memory, or in both. This depends on the instructions reading/writing the variable and on the spill operations. Additional constraints ensure that the number of variables held in registers does not exceed the number of available registers. A fundamental feature of our model is that a variable can die in register and/or in memory at any moment. For a variable v live at a program point p, we introduce the following 0-1 variables:

$\rho_{1,p,v} = 1$ iff v is available in a register at the beginning of p.
$\rho_{2,p,v} = 1$ iff v is available in a register at the end of p.
$\mu_{1,p,v} = 1$ iff v is available in memory at the beginning of p.
$\mu_{2,p,v} = 1$ iff v is available in memory at the end of p.
$s_{p,v} = 1$ iff v is stored to memory at (the beginning of) p.
$l_{p,v} = 1$ iff v is re-loaded from memory at (the end of) p.

The variables $s_{p,v}$ and $l_{p,v}$ can be deduced from the other 0-1 variables. Nevertheless, we keep them for readability and to simplify the specification of the ILP cost function.

3.2.1.1 Constraints

Definitions and Uses On a load-store architecture, a variable v must be in a register immediately after its definitions and immediately before its uses. In other words, for a program point p that immediately precedes an instruction that uses v, the variable v must be in a register at the end of p, i.e.:

$$\text{(Use)}\ \rho_{2,p,v} = 1$$

Similarly, for a program point p that immediately follows an instruction defining v, the variable v must be in a register at the beginning of p, but is not available in memory:

$$\text{(Def}_R\text{)}\ \rho_{1,p,v} = 1 \qquad \text{(Def}_M\text{)}\ \mu_{1,p,v} = 0$$

Loads and Stores To do a `load` (resp. `store`) of v at a program point p, the variable v has to be available in memory (resp. register) at the beginning of p:

$$\text{(Load)}\ l_{p,v} \leq \mu_{1,p,v} \qquad \text{(Store)}\ s_{p,v} \leq \rho_{1,p,v}$$

To make things simpler (this does not change optimality), we add the following constraints, which mean that a `load` (resp. `store`) does assign a register (resp.

memory location), at the end of p:

$$\text{(Load*)}\ l_{p,v} \leq \rho_{2,p,v} \qquad \text{(Store*)}\ s_{p,v} \leq \mu_{2,p,v}$$

Propagation A variable v is available in a register at the end of a program point p if it was available in a register at the beginning of p or it has just been read from memory using a `load`:

$$(\text{Reg}_p)\ \rho_{1,p,v} + l_{p,v} \geq \rho_{2,p,v}$$

Similarly, a variable is available in memory if it was already in memory or if it has just been written using a `store`:

$$(\text{Mem}_p)\ \mu_{1,p,v} + s_{p,v} \geq \mu_{2,p,v}$$

It remains to ensure the consistency between two successive program points p and q for a variable v that is not defined by the possible instruction between p and q: v is in register (memory) at the beginning of q if it is in register (memory) at the end of all program points p that immediately precede q.

$$(\text{Reg}_{p,q})\ \rho_{2,p,v} \geq \rho_{1,q,v} \qquad (\text{Mem}_{p,q})\ \mu_{2,p,v} \geq \mu_{1,q,v}$$

Note that by using inequalities ($\geq$) instead of equalities ($=$), it is possible to release register and memory locations at any time (i.e., v dies), both within and between program points.

Register Pressure There should be at most k variables in a register at the beginning and at the end of each program point p, where k is the number of available registers:

$$(\text{Pres}_b)\ \sum_v \rho_{1,p,v} \leq k \qquad (\text{Pres}_e)\ \sum_v \rho_{2,p,v} \leq k$$

Example For the two successive program points p and q surrounding the instruction `b ← a + 1`, the following constraints are generated: $\rho_{1,q,\texttt{b}} = 1$, $\mu_{1,q,\texttt{b}} = 0$, $\rho_{2,p,\texttt{a}} = 1$, $\rho_{1,q,\texttt{a}} \leq \rho_{2,p,\texttt{a}}$, and $\mu_{1,q,\texttt{a}} \leq \mu_{2,p,\texttt{a}}$. These last two constraints are similar for variables whose live-ranges traverse the instruction. Register pressure constraints are also added. Figure 3.6 illustrates the relation among the associated ILP variables.

3.2.1.2 Objective Function

Our goal is to minimize the expected cost of spill code at runtime (code size could also be modeled). We denote by F_p the expected execution frequency of program point p and by $C_{store_{p,v}}$ and $C_{load_{p,v}}$ the costs of a `store` and a `load` for variable v at p. The parameterization of the costs with p and v gives additional freedom for our advanced formulations presented later. We then aim at minimizing:

$$\sum_p \sum_v F_p \left(C_{store_{p,v}} s_{p,v} + C_{load_{p,v}} l_{p,v} \right)$$

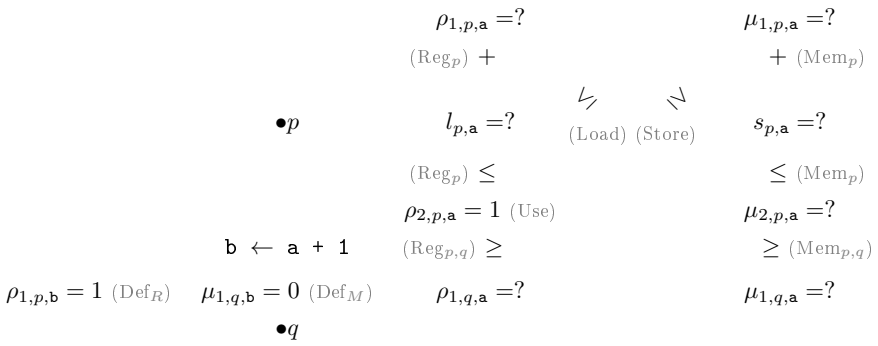


Figure 3.6: Generated ILP variables and the related constraints on an instruction and its surrounding points. Question marks denote values to be set by the ILP solver. The name of the rule is given next to the corresponding constraints.

3.2.1.3 Fully Rematerializable Variables

A variable v is *fully* rematerializable if all its definitions evaluate to the same value that is recomputable on *every* program point for free. This is the particular case of rematerialization captured in the model of Goodwin et al. In our basic ILP formulation, we can easily express it as follows. For a program point p_v after a definition of a fully-rematerializable variable v, instead of applying Def_M and Def_R, we simply force $\mu_{1,p_v,v} = 1$ (then loading means recomputing) but leave $\rho_{1,p_v,v}$ unspecified (a solution with $\rho_{1,p_v,v} = 0$ means that the definition is removed at this point). We then redefine $C_{load_{p,v}}$, the cost of loading (here, of recomputing), by C_{remat_v}. Finally, to take definition removals into account, we subtract $F_{p_v} C_{remat_v}(1 - \rho_{1,p_v,v})$ from the objective function. A more general model of rematerialization is given in Section 3.2.4.

3.2.2 Emulating Other Formulations

With a few additional constraints, we can emulate other ILP approaches, in particular those of Appel and George, and of Koes and Goldstein, as well as heuristic strategies such as the *spill-everywhere* approach.

To emulate spill-everywhere simplifications, for each variable v, we restrict the program points where a `store` (resp. `load`) can be inserted to the points immediately after the definitions (resp. before the uses) of v. This translates into setting the ILP variables representing the insertion of `load` or `store` to 0 on all the points that do not match the previous constraints. Moreover, in a spill-everywhere strategy, when a variable is spilled, *all* its definitions are stored and *all* its uses are loaded. To achieve this behavior, all the permitted `store`s and `load`s of a variable have to be linked. To not change the way ILP variables and constraints are generated in our implementation, we added constraints that state that all these ILP variables are equal. Another way could be to use a single ILP variable for all these actions.

To emulate Appel and George, we just need to forbid a variable to be in register and memory at the same time. This can be done by adding the constraint $\mu_{2,p,v} + \rho_{2,p,v} = 1$ for every program point p and variable v live at p. As for $\mu_{1,p,v} + \rho_{1,p,v} = 1$, it is implied by the propagation constraints $\mathrm{Reg}_{p,q}$

and $\text{Mem}_{p,q}$. An alternative formulation is to force a `store` (resp. `load`) to release the corresponding register (resp. memory location):

$$(\text{Appel}_l)\ l_{p,v} + \mu_{2,p,v} \leq 1 \qquad (\text{Appel}_s)\ s_{p,v} + \rho_{2,p,v} \leq 1$$

It is interesting to note that, if we do not add the Appel_l constraints but only keep the Appel_s constraints, i.e., a `load` does not force the variable to die in memory, we retrieve the model of Koes and Goldstein, in which the cost of a `store` is zero when the variable has already been stored. Actually, to get a faithful emulation, we should slightly weaken the model to express the limitation exposed in Section 3.1.2.1. This can be done by adding $\rho_{1,p,v} = 1$ for every program point p after a use of variable v that is not the last use.

Note that these two emulations, of Appel et al. and of Koes et al., are both derived by over-constraining our basic formulation. Thus, our formulation is more general and expresses more solutions. Compared to the formulation of Appel et al., our ILP unknowns express where variables are stored (register and/or memory) while Appel et al. express the movements of these variables between register and memory (mutually exclusive).

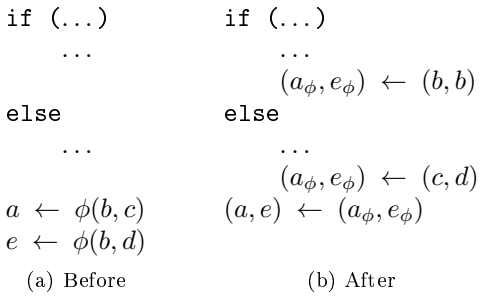
3.2.3 Handling SSA and ϕ-Functions

We now explain how to extend the previous basic formulation to deal with SSA programs. Several approaches are possible depending on whether live-ranges of SSA variables are considered to be unrelated, we call this the *basic SSA* approach (see Section 3.2.3.1), or whether copy relations implicit in ϕ-functions are exploited. In this latter case, the fact that arguments of a ϕ-function can interfere complicates the formulation: we then propose two solutions, an *optimistic approach* that may require repair code, and thus optimizes an under-estimation of the spill costs, and a *pessimistic approach* that conservatively exploits memory-to-memory copies. The way we handle ϕ-functions can also be used to exploit regular `move` operations, thanks to the notion of *local equivalence class* that will be explained later on. However, this extension has limited impact for the benchmarks we considered, which have few `moves`. We also present support for more sophisticated rematerialization.

3.2.3.1 Basic SSA

The easiest way of handling SSA programs is to consider live-ranges of SSA variables as unrelated and to interpret ϕ-functions as copies between variables. The basic formulation of Section 3.2.1 can then be applied on the code that would be obtained by direct out-of-SSA translation [100]. In this process, the different ϕ-functions are represented by parallel `move` operations that are implicitly placed at the program point representing a ϕ-function and its predecessors as illustrated by Figure 3.7. These parallel copies are then sequentialized, which may require an additional variable.

This approach, although correct, has several weaknesses. For load-store architectures, it requires every argument of a ϕ-function to pass through a register at the corresponding copy. This may increase spilling. Also, each ϕ-function potentially induces a `store` if the corresponding variable is spilled. Finally, the fact that a particular sequentialization is chosen a priori may preclude opportu-



(a) Before (b) After

Figure 3.7: Replacement of ϕ-functions.

nities. Thus, when considering SSA variables, it is preferable to combine spilling with a form of copy coalescing, in particular the coalescing of memory locations.

3.2.3.2 Optimistic Coalescing

A more natural handling of ϕ-functions is to consider a ϕ-function as a propagation between program points, i.e., to transfer values of a ϕ-function through registers and memory: the result of a ϕ-function is available in register (resp. memory) if all other arguments are in register (resp. memory). More formally, for every program point p_i, $1 \leq i \leq n$, preceding a program point q that represents a ϕ-function $a_0 \leftarrow \phi(a_1, \ldots, a_n)$, we add the following two constraints:

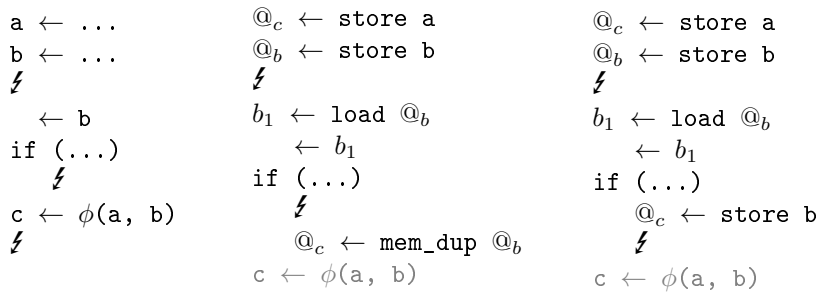
$$(\text{Phi}_R)\ \rho_{1,q,a_0} \leq \rho_{2,p_i,a_i} \qquad (\text{Phi}_M)\ \mu_{1,q,a_0} \leq \mu_{2,p_i,a_i}$$

In this approach, implicit memory-to-memory copies, expressed by the constraints Phi_M, are allowed at no cost. This model is used in the heuristic of Braun and Hack [21], assuming that the program is in CSSA, which guarantees that no actual memory copies are required (how Ebner et al. [39] capture ϕ-functions is not explained). Indeed, in CSSA, variables connected by ϕ-functions do not interfere and can be spilled to the same memory location.

The same approach can be used *optimistically* for programs that are not in CSSA, by observing that memory live-ranges are shorter than the original live-ranges and thus, after spilling, are less likely to interfere than the original live-ranges. After ILP solving, ϕ-functions whose results are not in a register at their definition point are converted to ϕ-functions with memory operands. The live-ranges of all memory locations are then computed and coalesced using aggressive coalescing [33]. Finally, repair code is inserted that performs a transfer from the memory location of a ϕ-function argument to the appropriate destination, when the argument has not been coalesced with the result of the ϕ-function. These additional costs are not reflected in the ILP objective function, which may lead to sub-optimal solutions. Also, in the worst case, the repair code may locally increase register pressure, which might lead to additional local spilling.

3.2.3.3 Pessimistic Coalescing

The *pessimistic* approach proceeds in the opposite manner. Parallel `move` operations are implicitly placed at the program point representing a ϕ-function and its predecessors, as illustrated in Figure 3.7a. Next, liveness is computed, an



(a) Original (b) Optimistic/pessimistic (c) Optimal

Figure 3.8: Optimal memory duplication placement.

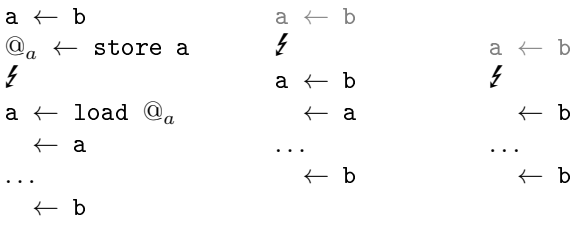
interference graph of all live-ranges is built, and aggressive coalescing is used to define sets of coalescable variables. These sets are then used during the construction of the ILP problem to express memory-to-memory duplications and take their costs into account in the ILP objective function. This is expressed using two new constraints Mem_{cpy} (copy at no cost) and Mem_{dup} (duplication) detailed hereafter.

This approach is *pessimistic*, because, whenever a variable interferes statically with another variable in the original program, it is assumed that the spill locations of these variables will also interfere in the final program. After ILP solving, however, we might encounter that these memory locations actually do not interfere, because the variables are not kept in memory throughout their complete live-ranges. Using a post-processing, we may thus eliminate useless memory duplications, again by coalescing memory locations, and lower the spill cost. In contrast to the *optimistic* variant, this post-processing is optional and not required for correctness.

3.2.3.4 Optimal Coalescing

None of the approaches presented in the previous sections captures the optimal solution, as shown by the example of Figure 3.8. Optimally solving the memory coalescing problem along with the spilling problem is intractable at the moment due to the subtle semantics of ϕ-functions and the complexity of capturing the actual live-ranges in memory, which are not known before spilling is done. The problem of expressing optimal solutions for non-CSSA programs is thus left open. However, we can draw a hierarchy between the different approaches compared to an optimal solution, as follows.

The basic SSA approach over-constrains the program by forcing the operands of ϕ-functions to be in register. Clearly, this might be sub-optimal in certain cases. The pessimistic approach might also yield sub-optimal solutions, due to its conservative choice of coalescable memory locations and the resulting overestimation in the ILP objective function. The optimal solution can still be achieved in some cases during post processing, i.e., when spurious memory duplications are eliminated by coalescing. Due to its added expressiveness, the pessimistic approach is guaranteed to give better solutions than the basic SSA approach. The optimistic approach, in contrast, may find solutions whose objective function are even *better* than optimal. This may happen when memory



(a) useless spill (b) code motion (c) renaming

Figure 3.9: Exploiting `moves` as special operations.

locations are falsely assumed to be coalescable. Repair code is consequently inserted to correct this underestimation, resulting in the final, potentially sub-optimal, spill code. Since these underestimations of the cost are implicit in that model, it can end up with a lot of such insertions. The solution may then even be worse than that of the basic SSA model, even if this is unlikely.

3.2.4 Extended Formulation

We now present an extension of the basic ILP formulation described in Section 3.2.1, which can be customized to express the different approaches proposed earlier, by predefining some variables or by omitting certain constraints. These details can be skipped at first reading.

3.2.4.1 Handling Regular Copy Operations

As described in Section 3.2.3, an important feature is to be able to exploit `moves`, which are implicit in the SSA ϕ-functions. In particular, we want to take into account the fact that memory coalescing may not be possible. Actually, the same situation occurs for regular `moves`, which may appear in both SSA and non-SSA programs.

Moves Figure 3.9 illustrates a situation where spill code can be avoided by exploiting `move` operations, with either code motion or renaming. To express such an optimization, we introduce the notion of *local equivalence classes* as the set of variables, denoted $EC_{p,v}$, that carry the same value as v at program point p (these sets can be statically pre-computed). This allows us to express several additional features. For example, whenever a variable v is used, we may choose to read another variable u from its equivalence class, if u is in register, or to `load` from the memory location of u. We may also allow to insert an explicit register-to-register copy between u and v. To describe the constraints more easily, we treat the original `move` as a virtual operation, using an artificial program point. Figure 3.10 illustrates the handling of equivalence classes.

Crossing Variables On load/store architectures, memory-to-memory copies require a register. Hence, we have to account for an additional register at a program point with memory copies, unless the memory locations of all copies can be coalesced. We also want to express explicitly the newly-introduced re-materialization and `move` operations, even if some have cost 0 in our objective

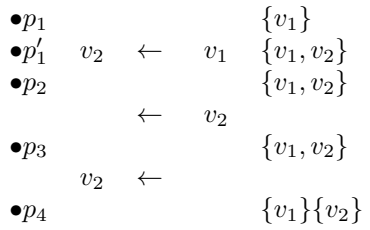


Figure 3.10: Moves & local equiv. classes (in brackets). Program points are on the left, instructions in the middle and local equivalence classes on the right. An equivalence class is valid only on the related program point. On p_1, only one variable is alive (v_1), thus this point has only one equivalence class: $\{v_1\}$. The point p'_1 defines v_2 as a copy of v_1, hence v_1 and v_2 share the same value at p'_1 and p'_1 equivalence class is $\{v_1, v_2\}$. Between p_3 and p_4, v_2 is redefined. On p_4, v_1 and v_2 do not share the same value anymore, thus p_4 has two equivalence classes: $\{v_1\}\{v_2\}$. Live-ranges *may* also be extended (here v_2 at p_3).

function. A fixed order of these operations may lead to a suboptimal solution. Nevertheless, to keep our ILP formulation practical, we chose the following static ordering: (1) `store` operations, (2) memory-to-memory copies, (3) `load` operations, (4) rematerialization operations, and finally (5) register-to-register `moves`. The assignment of a variable to a register may be released either at the beginning of the program point by a `store` or in the middle by a register-to-register `move`. We account for the variables *crossing* this region in a register to ensure that the register pressure never exceeds the number of registers.

ILP Variables The extended formulation introduces 6 new 0-1 variables, for each variable v live at a program point p:

$mem\_cpy_{p,v} = 1$ *memory copy* into memory slot of v.
$mem\_dup_{p,v} = 1$ *duplication* into memory slot of v.
$has\_mem\_dup_p = 1$ at least one *memory duplication* at p.
$move_{p,v} = 1$ register-to-register *move* to v at p.
$remat_{p,v} = 1$ rematerialize v at p.
$cross_{p,v} = 1$ v still in register after the `store`s at p.

Both memory *copies* and memory *duplications* represent memory-to-memory transfers. The difference between them is that memory copies are only applicable to memory locations that can be coalesced, in which case they are for free. Memory duplications on the other hand cause a `load` followed by a `store` and, in addition, require a register.

3.2.4.2 Constraints

In the extended formulation, most constraints change to express the opportunities offered by local equivalence classes. As these changes are straightforward, we summarize them quickly and focus the discussion on the additional spill operations. When `moves` are not exploited, equivalent constraints can be derived using singleton equivalence classes.

Crossing Due to the additional spill operations and the fact that memory duplications might require temporary live-ranges that are not visible at the beginning or the end of the program point, we track variables crossing through the program point in a register. The resulting propagation constraint for a variable v at program point p becomes:

$$\text{(Cross) } \rho_{1,p,v} \geq cross_{p,v}$$

Using Equivalence Classes Instead of requiring a given variable to be in register at a use site, it is sufficient that some variable $u \in EC_{p,v}$ is available in a register. Note that, of course, $v \in EC_{p,v}$. Thus, at a program point p preceding a use of variable v, we apply the following constraint:

$$\text{(Use) } \sum_{u \in EC_{p,v}} \rho_{2,p,u} \geq 1$$

Similar constraints allow a `load` (resp. `store`) to read from the memory location (resp. register) of another variable:

$$\text{(Load) } l_{p,v} \leq \sum_{u \in EC_{p,v}} \mu_{1,p,u} + s_{p,u}$$
$$\text{(Store) } s_{p,v} \leq \sum_{u \in EC_{p,v}} \rho_{1,p,u}$$

Moves We do not represent an explicit `move` between variables as an instruction but as a program point with additional constraints. Instead of forcing the operands of the `move` into a register using the regular Use or Def_R constraints, we indicate that the result is neither in memory nor in register. For a program point p representing an explicit `move` defining a variable v, we write:

$$\text{(Def}_{move}\text{) } \rho_{1,p,v} = 0,\ \mu_{1,p,v} = 0$$

This has the effect of killing any previous value of v (but v is added in the right equivalence class). However, since the original instruction is removed, we have to provide a way to instantiate the `move`, if needed. A `move` $v \leftarrow u$ can be performed anywhere along the original live-range of v, or beyond if desired, as long as u belongs to the equivalence class of v. Given the equivalence class $EC_{p,v}$, a `move` can be instantiated at p under the following conditions:

$$\text{(Move) } move_{p,v} \leq \sum_{u \in EC_{p,v}} cross_{p,u} + l_{p,u}$$

Note that, in contrast to other constraints, we use $cross_{p,u}$ instead of $\rho_{1,p,u}$ to express that `moves` appear after the `store` and memory duplication operations on a program point.

Memory Copies Memory-to-memory copies have different implications depending on whether the related memory slots are coalescable or not. A truly-optimal spilling approach would require to solve the memory coalescing problem along with the spill code placement. This is hardly an option since coalescing,

even aggressive, is NP-complete [17]. The constraints presented hereafter can be used with different coalescing strategies, including integrated approaches.

Let v be a variable live on a program point p and let $CC_{p,v} \subseteq EC_{p,v}$ be the set of variables whose memory slots can be coalesced with the memory slot of v. A *memory copy* can be performed at no cost under to following condition:

$$(\text{Mem}_{cpy})\ mem\_cpy_{p,v} \leq \sum_{u \in CC_{p,v}} \mu_{1,p,u} + s_{p,u}$$

A *memory duplication* can be done regardless of whether v can be coalesced with the source of the duplication as long as both are in the same equivalence class, thus:

$$(\text{Mem}_{dup})\ mem\_dup_{p,v} \leq \sum_{u \in EC_{p,v}} \mu_{1,p,u} + s_{p,u}$$

To limit register pressure, we need to know whether at least one memory duplication is to be performed on a point p. We can express whether a memory duplication is performed at p using a 0-1 variable $has\_mem\_dup_p$:

$$\forall v,\ has\_mem\_dup_p \geq mem\_dup_{p,v}$$

Register Pressure The following constraint ensures that the register pressure is not exceeded within a program point p, even when *memory duplications* are performed:

$$(\text{Pres}_c)\ \sum_v cross_{p,v} + has\_mem\_dup_p \leq K$$

Rematerialization The rematerialization is explicitly defined in the extended formulation. The purpose is to have a clean model of what is in memory and what is not. This information is important when memory-to-memory copies read from a rematerializable variable (see Figure 3.11). At a program point p where the rematerialization of v does not require any argument, we allow an additional "spill" operation as follows:

$$(\text{Remat})\ remat_{p,v} \leq 1$$

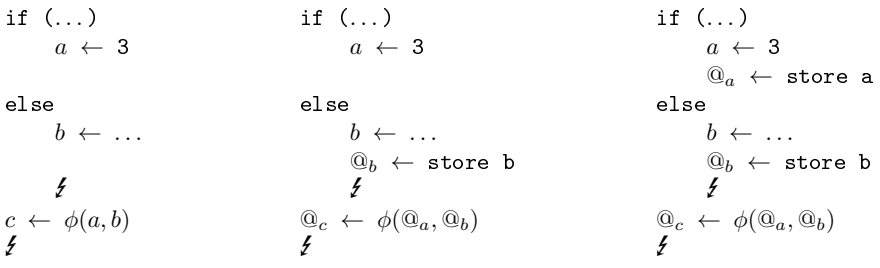
For a rematerialization reading from a set of arguments A:

$$(\text{Remat}_A)\ |A| remat_{p,v} \leq \sum_{u \in A} cross_{p,u}$$

More complex compositions of rematerialized expressions are straightforward to express for SSA programs. Since we limit our evaluation to simple rematerialization, we do not discuss these capabilities any further at this point.

Propagation A variable v can be in register at the end of p, if it is the result of a `move`, a `load`, a rematerialization, or if the variable crosses p in register:

$$move_{p,v} + l_{p,v} + remat_{p,v} + cross_{p,v} \geq \rho_{2,p,v}$$



(a) Origin

(b) Variable a is not in memory. Loading c may give an undefined value.

(c) Extended model

Figure 3.11: Memory copies implies a correctness problem with rematerialization as presented for the basic formulation.

Likewise, v can only be in memory at the end of p, if its memory location was defined by a `store`, *memory duplication or copy* on p, or was available before:

$$s_{p,v} + mem\_dup_{p,v} + mem\_cpy_{p,v} + \mu_{1,p,v} \geq \mu_{2,p,v}$$

Finally, the constraints to propagate between two program points are unchanged. Local equivalence classes are not used here, so as to capture the cost of register-to-register and memory-to-memory copies. Note also that the constraints of the extended formulation can easily emulate our basic formulation by pre-setting the 0-1 ILP variables representing the new spill operations and by restricting equivalence classes. The same is true for the proposed approaches to coalesce memory locations of copy-related variables, either coming from ϕ-functions or from regular copies. These approaches are summarized in Table 3.1.

Basic SSA	No exploitation of `moves` and ϕ-functions
	No explicit memory copies nor duplications
	Coalescing and repairing *after* ILP
Pessimistic	Explicit moves at ϕ-functions
	Aggressive memory coalescing *before* ILP
	Coalescable memory copies for free
	Memory duplications with cost
	No repairing needed
Optimistic	Free memory copies at ϕ-functions
	All variables are assumed to be coalescable
	Explicit memory copies are free
	Coalescing and repairing *after* ILP

Table 3.1: This table lists the different strategies we proposed to deal with moves and memory coalescing under SSA and details their handling.

3.3 Experiments

We made our experiments on the *ST231* embedded processor for media applications. This is a 4-way parallel VLIW architecture, supporting one memory operation per instruction bundle. It features a direct-mapped cache of 32KB for instructions (64b lines) and a 4-way set associative cache of 32KB for data (32b lines). Both caches are connected to a shared bus memory controller with an average latency of 120 cycles to access off-cache data. For in-cache data, the latency between the `load` and a use is 3 cycles. The pipeline is stalled automatically if this latency is violated, i.e., at least 3 instruction bundles have to follow a `load` to hide the cache latency. The data cache follows a write-through strategy. A store buffer for memory writes allows to group up to 4 `store` requests into a single bus transaction. In case of a `store`/`load` conflict in the store buffer, the `store` must be processed down to the memory before being reloaded.

We implemented our ILP spiller in the static C compiler of STMicroelectronics, which is based on Open64[2]. Register allocation, and thus spilling, is performed in a separate back-end optimizer that comes with the production compiler. The register allocation uses a decoupled approach, where spilling is by default performed using a heuristic and assignment using graph coloring. In the following experiments, we compare several spilling approaches, both exact and heuristic:

Appel-G Appel and George's ILP Formulation [3],
Coloring Heuristic using iterated register coalescing [51],
Basic Our basic formulation, see Section 3.2.1,
SpEv Basic formulation emulating spill everywhere,
Koes-G Emulation of Koes and Goldstein's ILP Formulation [65].
BasicSSA Naive handling of SSA, see Section 3.2.3.1,
SpEvSSA Emulation of spill everywhere under SSA,
Optimistic Extended formulation, see Section 3.2.3.2,
Pessimistic Extended formulation, see Section 3.2.3.3,
Hack Hack's SSA-based spilling heuristic [55].
Braun-H Braun and Hack's SSA-based spilling heuristic [21].

The first 5 configurations were evaluated using regular non-SSA programs, while the others were applied to SSA programs. In both cases, critical edges were split prior to spilling. We also show results for configurations with equivalence classes enabled (marked by a suffix *_ec*) and with rematerialization enabled (suffix *_remat*) – both disabled by default, unless it is a basic feature of the model like Koes-G and Braun-H. For the ILP solver, we used IBM CPLEX for academics, version 12.2. All configurations were tested on the benchmark suites EEMBC 1.1 and SPECINT 2000, excluding the C++ program eon. The compiler was invoked using the `-O3` optimization level with the number of allocatable registers limited to 8 (4 callee-saved and 4 caller-saved registers) so that spilling effects are more apparent (see comments below). The cost model is based on basic block frequencies, which were either derived from profiling feedback or from estimates according to Ball and Larus' heuristic [5].

The experiments investigate the solving time of our formulation, its impact on static spill costs over all benchmark programs, and the effects on the runtime

[2] `http://www.open64.net/`

behavior of the EEMBC benchmarks. Runtime measurements for SPEC are not shown, because the benchmarks are too large for our architecture's instruction cache. The programs spent up to 65% of the time waiting for the instruction cache, making all runtime measurements irrelevant for spilling. We set a time limit of 1000 seconds for all ILP configurations. To avoid the impact of random results when the optimal solution is not reached, all presented numbers refer to optimally-solved instances only. To reduce the solving time, a heuristic supplies an initial solution for all ILP configurations. Hack's heuristic is used for SSA programs, graph coloring otherwise. Therefore, when the solver reaches the time limit, the provided solution is at least as good as the related heuristic.

Note on the number of registers The *ST231* is classically shipped with 64 registers. However, this number is too big to reveal any interesting difference on the runtime of EEMBC benchmarks. In other words, the amount of spill code is too limited relatively to the rest of the program. For SPEC, this amount of registers could have been relevant, but the runtime is not, as explained above. In the end, we chose 8 allocatable registers to recall ARM and x86 architectures.

3.3.1 Solving Time

Our primary goal was to express the spilling problem in a simple and flexible way. Speed was not a major concern. However, to reduce the solving times, we restricted the program points where loads and stores can be performed, without losing optimality. For example, for all formulations where a variable v can be simultaneously in register and in memory, it is sufficient to consider solutions where loads (resp. stores) for v are just before (resp. after) each use (resp. definition) of v and at the end (resp. beginning) of basic blocks [52].

After 20s, whatever the formulation, 90% of the functions of EEMBC are solved optimally. For SPEC, whose functions are larger, this takes 65s. SpEv is the fastest configuration to solve. After 5s, 95% (resp. 90%) of the functions are solved optimally for EEMBC (resp. SPEC). As a comparison, Appel-G solves 79% (resp. 85%) of the functions optimally in 5s. After 1000s, all 656 functions of EEMBC were solved optimally, except for the Optimistic and Pessimistic configurations which reached the time limit for 2 of them. For SPEC, 99% of the 5060 functions are solved optimally when reaching the time limit whatever the configuration. Note that we excluded from these numbers the functions that do not require any spill code when solved by the heuristic used to initialize ILP. In this case, we do not invoke the ILP solver. In other words, the presented numbers do not include 221 functions of the EEMBC and 719 functions of the SPEC benchmarks that are solved optimally in a trivial manner (with no spill).

Figure 3.12 gives an overview of the solving times for all EEMBC and SPEC benchmarks. The curves show the percentage of functions solved optimally in a given amount of time. These timings may change depending on the workload of the machine and heuristic choices of the ILP solver. We observe that the solving times of most configurations behave similarly, except for the SpEv configuration which is consistently solved the fastest for both benchmark suites. As expected, the solving times increase for larger instances having more points, i.e., because of larger functions or additional variables introduced by SSA form.

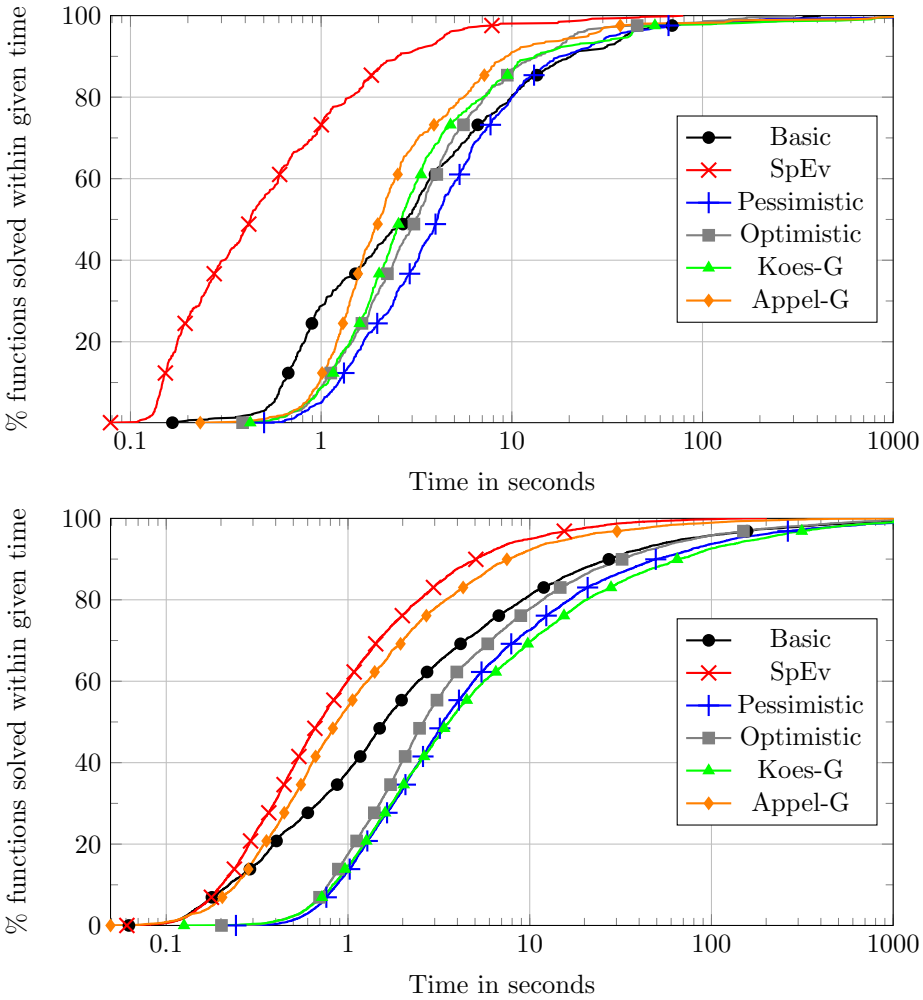


Figure 3.12: Percentage of functions that can be solved optimally in the given amount of time for EEMBC (top) and SPEC (bottom). The markers on the curves help to identify the related configuration, they do not represent the actual measurements, which would have been too dense.

3.3.2 Static Spill Cost

In this section, we compare the different approaches with respect to the *static* spill costs, following the cost model provided by Open64, where a `store` costs 1.25, a `load` 3.25, and a rematerialization 1 (all numbers multiplied by the expected execution frequency). These costs are computed from the actual spill code after clean-ups and insertions of repair code, e.g., due to memory coalescing or duplication. They may thus differ slightly from the ILP objective value. Figure 3.13 shows the geometric mean of the costs over all benchmarks, normalized to the Appel-G configuration and obtained by summing the costs of all functions that are optimally solved by both the Appel-G configuration and the spilling strategy at hand. For the heuristic approaches, we consider all functions that are optimally solved by Appel-G. In addition, the variation is depicted using the minimum and maximum.

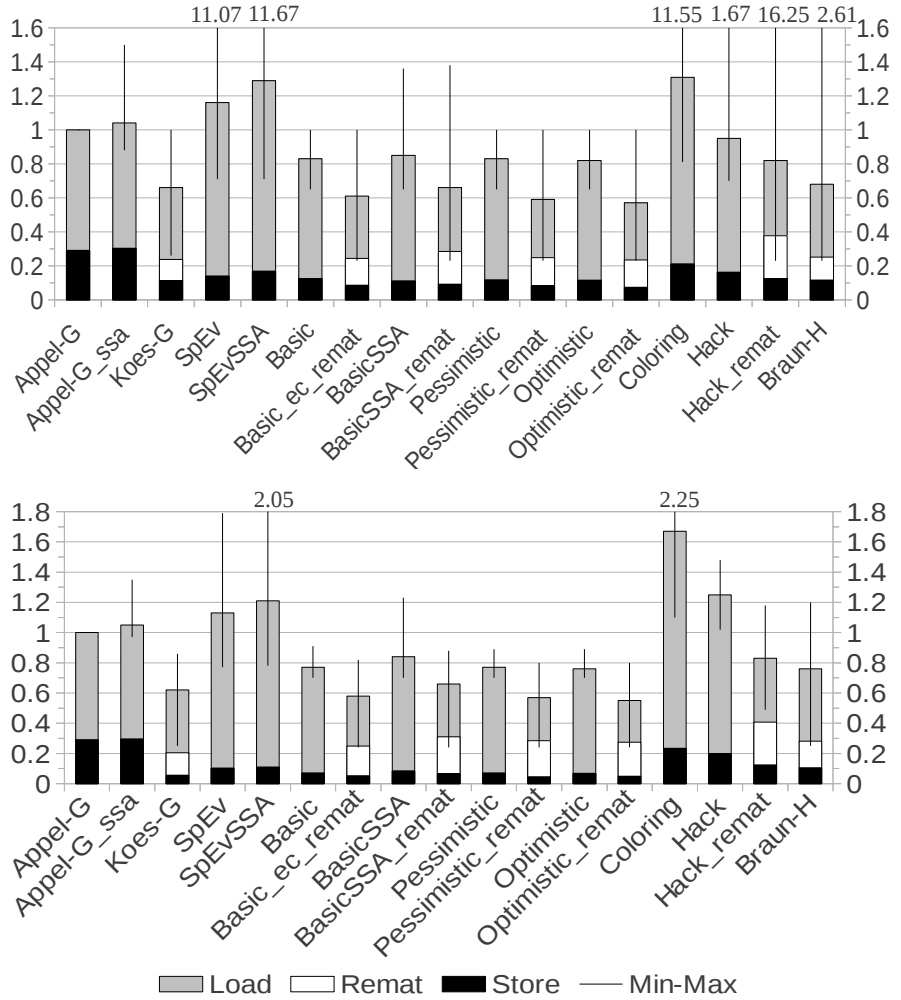


Figure 3.13: Geometric mean of static spill costs for EEMBC (top) and for SPEC (bottom), using frequency estimates. (Lower is better)

Despite its restrictions, Appel-G performs much better than optimal spill everywhere (SpEv), which is the worst, except for the graph coloring heuristic (Coloring), which is also spill everywhere. This is particularly clear on Figure 3.14, which represents the static spill costs per function, where the curve of spill everywhere is almost always above Appel-G (y=1). All other configurations based on our ILP formulation outperform Appel-G by about 20% or more. This is mostly due to the elimination of spurious `stores` (recall Figure 3.3). The impact of SSA is interesting to examine. For spill everywhere, the fact that SSA offers live-range splitting does not fully counterbalance the requirements implied by uses of the naive modeling of ϕ-functions. In this setting, SSA increases the static spill costs by 13% for EEMBC (8% for SPEC). BasicSSA performs better than Appel-G, but increases spill costs in comparison to Basic (non-SSA), with quite a few bad cases, due to the naive handling of ϕ-functions. This can also be observed with Appel-G under SSA, which is 4% (resp. 5% for SPEC) worse than Appel-G. Note that SSA (without rematerialization) also leads to

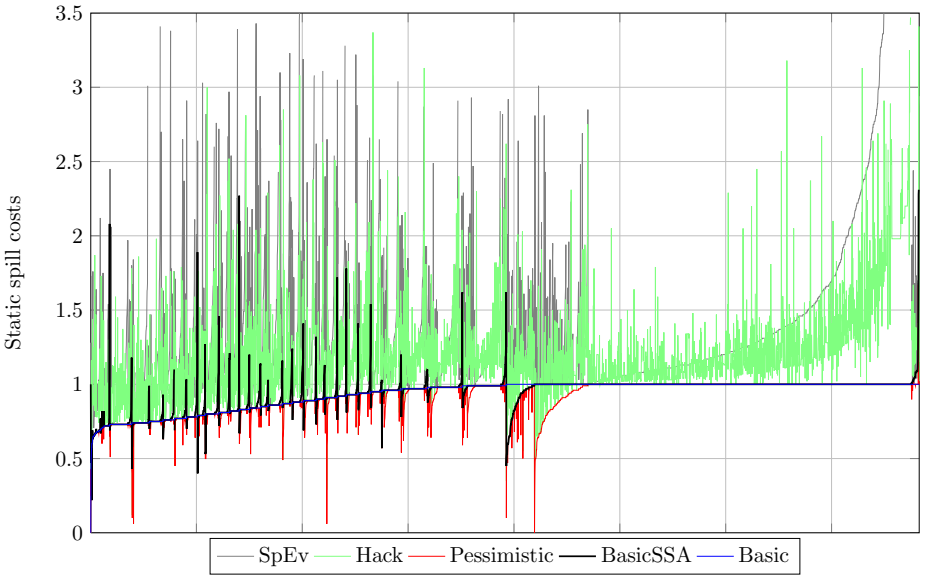
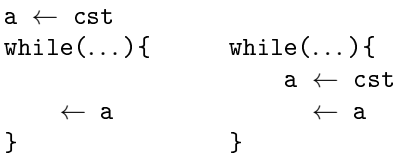


Figure 3.14: Static spill costs per function for EEMBC and SPEC, with frequency estimates. Results are normalized to Appel-G (y=1). For readability, functions where the static spill costs are the same for all ILP configurations (Appel-G, SpEv, Pessimistic, BasicSSA, Basic) were filtered. This filter applied to almost 37% of the functions. Results are sorted in increasing order of static spill costs according to Basic, then Pessimistic, then SpEv. (Lower is better)

some improvements in a few cases. This is surprising, since the naive handling of SSA constrains the solutions and also because all solutions reachable under SSA should also be reachable without SSA (when rematerialization is disabled). This is due to side effects. First, the spill code of a function may differ between the SSA and the non-SSA versions, leading to different register assignments. Then, since Open64 propagates information on register usage from sub-functions to call sites, this may lead to changes in register pressure around call sites and subsequent differences in spilling. Second, SSA construction performs a simple copy propagation that preserves CSSA constraints. This optimization can change the register pressure, hence the global spill code. These observations are particularly obvious in Figure 3.14, where some configurations achieve a cost of zero, whereas the reference, Appel-G, requires some spilling.

In contrast to the naive handling of SSA (in BasicSSA), exploiting ϕ-functions as proposed in Section 3.2.3 delivers good results without degradations (for both Optimistic and Pessimistic). Note that our current settings mostly result in CSSA programs. More aggressive transformations violating CSSA may thus have an adverse impact to the optimistic strategy.

Rematerialization gives remarkable improvements of more than 20% in comparison to the same configuration without rematerialization, and generally more than 40% compared to Appel-G. Good rematerialization is thus essential to reduce spill costs and ought to be considered accordingly. Moreover, configurations with rematerialization particularly profit from SSA. This is particularly true for simple rematerialization strategies. Indeed, in SSA, each rematerializable live-range matches a single variable, which results in more opportunities.



(a) Origin (b) After spill

Figure 3.15: Ad-hoc rematerialization support in Hack heuristic. Rematerialization is assumed to be free. Rematerializable values are not kept in register to allow other variables to use this location. Also, rematerialization occurs before every related use even if spilling was not need.

In principle, our extended formulation could handle more general cases than the simple rematerialization of constants. However, its full potential is not exploited here to preserve comparability with the other approaches.

Interestingly, Koes-G performs quite well. It is close to our formulation in the same configuration, i.e., simple rematerialization and not under SSA, Basic_ec_remat. Indeed, both geometric mean are within 4% for EEMBC (5% for SPEC). In particular, this formulation is able to remove most of the spurious `stores` compared to Appel-G. However, it makes less use of rematerialization than Basic_ec_remat. In that configuration, 32% of the static spill costs stem from rematerialization for EEMBC (34% for SPEC) whereas for the Koes-G formulation, it amounts only to 19% for EEMBC (24% for SPEC). This experimentally confirms the exposed limitation of Section 3.1.2.3.

In terms of static spill costs, the heuristics give better results for EEMBC than for SPEC. For rematerialization, we extended Hack's heuristic so that rematerializable values are always recomputed and never kept in registers, i.e., rematerialization is always assumed to be free. This heuristic, named Hack_remat, gives good results, especially for SPEC. However, we observe some bad cases for EEMBC (16.25x w/o profiling feedback) because these rematerializations are counted 1 in the Open64 cost model, while the heuristic behaves as if it costs zero. Figure 3.15 gives an example of such a bad case. Braun and Hack provided another heuristic to handle rematerialization. Their spilling algorithm is based on a farthest-first strategy using next-use distances. When two variables have the same next-use distance, rematerializable variables are spilled first. This Braun-H heuristic is the best among all the heuristics, also for the worst cases. This is not surprising. Like Hack's original approach, it does not rely on a spill everywhere strategy and, moreover, it provides an explicit control over rematerialization costs – avoiding some of the bad cases.

Figure 3.14 helps to draw a hierarchy between the different approaches depicted (without rematerialization). Spill everywhere is globally the worst, but Hack's heuristic shares the trends of that configuration. BasicSSA competes with Basic but suffers many bad cases (black spikes above Basic). Finally, Pessimistic is the best. Indeed, it avoids the bad cases of BasicSSA with its special handling of SSA ϕ-functions, plus it takes advantages of the simple copy propagation, discussed previously, that occurs when building SSA. This experimental ranking is slightly different than expected, since Basic should at least match the SSA-based configurations. For that, we could have implemented a simple copy propagation working on non-SSA code before spilling. We did not for two

	frequency estimates						profiling feedback					
	# ld/st			# instr.			# ld/st			# instr.		
Config.	G.m	Min	Max	G.m	Min	Max	G.m	Min	Max	G.m	Min	Max
Appel-G	1	1	1	1	1	1	1	1	1	1	1	1
−_ssa	1.01	0.89	1.34	1	0.92	1.16	1.02	0.95	1.12	1.01	0.94	1.11
Koes-G	0.84	0.34	1	0.95	0.84	1.04	0.85	0.37	1	0.94	0.85	1.01
SpEv	1.04	0.82	1.84	1	0.84	1.42	1	0.83	1.22	0.98	0.83	1.04
SpEvSSA	1.03	0.82	1.8	1	0.85	1.4	1	0.74	1.16	0.98	0.83	1.09
Basic	0.94	0.78	1.02	0.97	0.87	1.01	0.94	0.74	1	0.97	0.84	1.02
−_ec_remat	0.81	0.34	1	0.93	0.75	1	**0.82**	**0.37**	1	0.93	0.78	1.03
BasicSSA	0.95	0.78	1.12	0.98	0.86	1.1	0.95	0.74	1.1	0.98	0.84	1.17
−_remat	0.83	0.45	1.12	0.94	0.81	1.14	0.83	0.37	1.07	0.94	0.81	1.11
Pessimistic	0.93	0.78	1.02	0.98	0.87	1.1	0.94	0.73	1.02	0.96	0.83	1.02
−_remat	**0.8**	**0.34**	1	0.93	0.8	1.02	0.82	0.37	1	0.93	0.78	1.04
Optimistic	0.94	0.78	1.05	0.97	0.86	1.04	0.94	0.73	1.05	0.96	0.84	1.05
−_remat	0.81	0.34	1	0.93	0.77	1.02	0.82	0.37	1	0.92	0.8	1
Coloring	1.14	0.9	1.98	1.06	0.92	1.49	1.09	0.76	1.44	1.02	0.86	1.14
Hack	0.99	0.78	1.16	0.98	0.82	1.1	1.01	0.86	1.28	0.97	0.86	1.06
−_remat	0.83	0.34	1.1	**0.92**	**0.75**	1.03	0.85	0.37	1.1	**0.92**	**0.77**	1.06
Braun-H	0.89	0.57	1.17	0.96	0.82	1.13	0.91	0.56	1.11	0.97	0.8	1.13

Table 3.2: Number of dynamically-executed instructions with frequency estimates and profiling feedback (Geometric mean/Min/Max). Note that Koes-G and Braun-H feature simple rematerialization. (Lower is better)

reasons. First, this would not have changed the problem, mentioned before, due to different colors in the callee functions. Second, SSA does simplify some optimizations and we wanted to be able to stay in that configuration.

Static spill costs with profiling feedback show the same trends as frequency estimates. Thus, we did not include them here.

To conclude, restricting to solutions where a value is either in memory or in register (but not in both) gives good results but only if spurious `stores` are eliminated like in Koes-G configuration. In particular, this approach gives better results than spill everywhere strategies, which is not surprising. This tends to prove that a spilling heuristic can use this simplification and still achieve fairly good results (however, problems such as those mentioned in Section 3.1.2.1 have to be avoided). As for the strategies that we defined for dealing with SSA, they perform very well too, especially if ϕ-functions are exploited. Furthermore, SSA enables more rematerialization, which turns out to be very important.

3.3.3 Dynamic Counts

Table 3.2 compares the impact of the spilling strategy on the number of instructions that are dynamically executed, as reported by the *ST231* profiling tools when running the final assembly code. This corresponds to a sequential machine model, where every instruction completes in 1 cycle. The table provides the geometric means, the best cases, and the worst cases, over all benchmarks normalized to Appel-G.

The improvements seen for the static spill costs are still reflected by the dynamic execution counts and basically show the same trends with respect to the different configurations. However, the extent of the improvements is of

course reduced. This is due to the fact that the reported numbers include other code not related to spill code. In particular, Table 3.2 reports all `loads`/`stores` and not just those inserted during spilling. Overall, our approaches are very effective at eliminating dynamically-executed instructions. The best one (Pessimistic_remat) reduces the number of `loads`/`stores` by 20% (and up to 66%!). Even the total number of instructions is reduced by 7% (and up to 20%!). However, Hack with our rematerialization support achieves the best result for the total number of instructions. This may look surprising since the static spill costs were always worse than the Pessimistic approach. This is because the rematerialization support we implemented for that heuristic was selected with respect to the target architecture (*ST231*) and type of codes. Most rematerializable values come from constants and symbols representing base addresses of arrays. On the *ST231* family, most of these constants can be encoded directly with the operation itself or can be supplied as addressing mode. Consequently, most of the instructions emitted during the spilling phase as rematerialization do not manifest themselves in the final assembly code. Nevertheless, we did not want to model instruction re-writing but only the placement of instructions. This shows why static spill costs may be quite far from reality.

As a side-effect, reducing the number of memory accesses reduces the traffic to and from memory. Indeed, we also measured a reduction of instruction and data cache misses, although the objective functions of the ILP formulations assume a perfect cache (i.e., no cache misses).

3.3.4 Execution Time Measurements

We now focus on measurable runtime effects of the different strategies, specifically for the *ST231* architecture, which, unlike the sequential model that corresponds to *counting* instructions (Section 3.3.3), involves instruction and memory *latencies*, as well as instruction *bundling*.

The leftmost bars of Figure 3.16 report the geometric means (normalized to Appel-G) of the runtimes obtained by executing the programs compiled using the various spilling strategies using frequency estimates. The huge gains of static spill costs generally do not carry over to equivalent gains in execution time. For example, the 20% improvements in static spill costs of the configurations without rematerialization result in a moderate runtime mean gain of about 2%. Overall, we measured mean runtime improvements from 2% to 8%. Considering the best cases for individual benchmarks, we see impressive improvements that go up to about 30%. Note also that the Hack_remat heuristic is even performing slightly better than our "optimal" configurations.

Analyzing the individual benchmarks, we found that the difference between the dynamic execution counts and the actual execution times are mostly due to architectural features that are not accounted for in the spilling model but may occur on most targets. As mentioned in Section 3.3.3, this difference is not due to cache misses. However, memory latencies under a hit turned out to be highly relevant. So far, for both the static spill costs and the dynamic execution counts, we considered that `load` and `store` instructions induce a non-zero cost, irrespective of their placement within basic blocks. In practice, the runtime overhead of these instructions depends on the ability of the post-pass scheduler to hide their latencies. If the scheduler succeeds, even bad spilling decisions in terms of the *number* of `loads` might actually perform well. This explains why

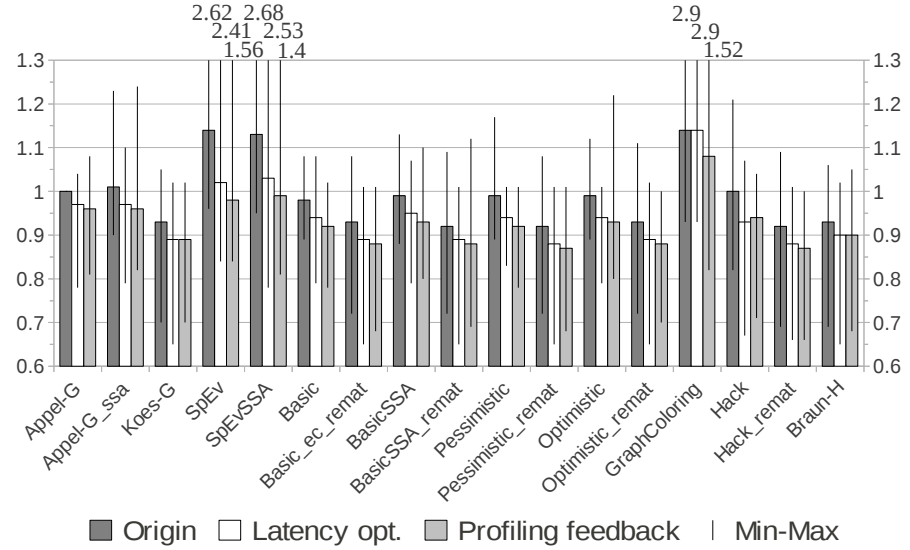


Figure 3.16: Runtime results for EEMBC, with post processing and profiling feedback. Note again that the Koes-G and Braun-H configurations feature simple rematerialization. (Lower is better)

the different configurations are relatively close to each other regarding execution time. The opposite effect is also possible. In particular, we observed that, due to the way `load`s and `store`s are placed (see Section 3.3.1), the post-pass scheduler of Open64, which runs after register assignment, fails to hide many latencies and to pack spill code nicely into bundles. This is particularly visible for the spill-everywhere strategies, since `load`s are systematically placed right before uses.

We thing that analyzing such runtime measurements is important, although they are almost never provided in the literature. In addition to revealing the weaknesses of the heuristics or formulations, they also reveal the weaknesses of the cost models that drive them (e.g., the fact that some `load`s can be cheaper than predicted) as well as the difficulties due to the architecture (e.g., instruction scheduling with bundles is more difficult to model than sequential execution).

Latency Post-Processing For purely-sequential targets, the cache-hit latency could be roughly modeled using the parametric costs we presented in our formulation (see Section 3.2.1.2). Consider Figure 3.17. Loading v on point q would cause the maximum latency, e.g., 3. Now, consider the point before the instruction `inst`, which is the instruction just before the use of v. Loading v at p will increase the `load`-to-use distance (*load-use distance*) by 1, reducing the related latency by 1, since *inst* will be executed before the use of v. This load-use distance can be computed for each program point to reflect the cache-hit latency. However, two inaccuracies lie in this model. First, the cost does not reflect the effect of other spill instructions inserted between a `load` and a use. Second, it assumes a fixed scheduling, i.e., it does not consider that instructions and, in particular, spill instructions can be reordered during post-pass scheduling. Regarding the first point, the model gives an over-approximation, which can be improved during the final insertion of spill code. Therefore, we believe

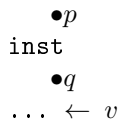


Figure 3.17: Effect of load-use distance on overall program performance. Although program points p and q are in the same basic block, loading v at p is more efficient than loading at q, where the load-to-use latency is fully paid.

it is a good model for this kind of targets if someone is ready to pay the extra cost of widening the solution search space (remember that, to speed up the ILP computations, not all points are considered for spill insertion, see Section 3.3.1).

Anyway, our target is not a purely sequential one and modeling the latency in the ILP may be expensive and is not straightforward. To have an accurate latency model, we would need to be able to formulate the bundling of instructions a priori. But this bundling may change according to the inserted spill instructions. As for the latency model for a purely-sequential target, we could estimate the number of bundles between a `load` and a use, so that we would not consider the spill code but still have an overestimation of the latency.

Finally, we chose a simpler method that is applicable for all targets. Instead of accounting for the cache-hit latency in the ILP itself (we will discuss this possibility in Chapter 4), we designed a heuristic, after spilling but before register assignment, which moves `loads` up within basic blocks, while respecting register pressure. The pseudo-code of this heuristic is given in Algorithm 1.

Algorithm 1 Post latency optimization

```
Require: Operands of all operations are on the same register file.
Require: No precolored variables.
 1: procedure optimizeLatency(Program p, integer limit)
 2:     for each block in p do
 3:         hideLoadLatency(block, limit)

 1: procedure hideLoadLatency(BasicBlock block, integer limit)
 2:     for each instruction in block from top to bottom do
 3:         if isSpillCode(instruction) and isLoad(instruction) then
 4:             currentLive ← getLiveVariablesAfter(block, instruction)
 5:             remainingLatency ← getRemainingLatency(instruction)
 6:             while remainingLatency > 0 and isDefined(instruction.prev) do
 7:                 prevInst ← instruction.prev
 8:                 // Cannot move instructions before an early instruction like ϕ or label.
 9:                 if isEarlyInstruction(prevInst) then
10:                     break
11:                 if checkDependency(prevInst, instruction) then
12:                     // Update liveness as if instruction is moved up
13:                     currentLive ← currentLive \ prevInst.defs
14:                     currentLive ← currentLive ∪ prevInst.args
15:                     if currentLive.size() ≤ limit then
16:                         block.moveInstructionUp(instruction)
17:                         remainingLatency ← remainingLatency - eatenLatency(prevInst)
18:                     else
19:                         break
```

In Procedure *hideLoadLatency*, the given basic block is traversed from the beginning to the end. The principle is that all instructions that appear before the current instruction have already been treated and will not move anymore,

while `loads` will move upwards (from bottom to top) among them. Thus, the approximation of the latency is computed on already-fixed instructions. On Line 3, the *isSpillCode* check avoids the need of memory alias analysis. However, with such analysis, one could also move regular load instructions. At Line 5, the function *getRemainingLatency* returns the latency that remains if the memory operation is placed at this program point, i.e., taking into account the latency of operations and a rough approximation of bundles (0.25 bundle for a `load` for example). Coming back to the example of Figure 3.17, assuming that the load-use latency of the target is 3 and a load of v has been placed on program point p, the remaining latency is 2. Line 9 prevents to move a `load` before an instruction that must be first in a basic block, e.g., a ϕ-function or a block label. Line 11, the *checkDependency* function performs a data dependency check to ensure that the instruction can be moved above (i.e., before) the previous instruction, i.e., it checks that prevInst does not define registers and memory locations read by the load and vice versa. The `load` is then moved up if the register pressure allows it, i.e., remains below the acceptable limit. Line 17, the eatenLatency function can be adapted to reflect the actual architectural parameters. In our case, it returns 1/4 per instruction, i.e., it assumes that the bundles are dense.

For our target, the load-use latency is not the single source of regression. The bundle density is also relevant. To improve this point too, we perform a similar optimization on `stores`, pushing them down (and not up as `loads`). The interest of such `stores` placement is to give more freedom to the final scheduler for placing `stores` into bundles. Without that, because of post-coloring constraints (write after read), the `stores` may be stuck at some place.

These heuristics were applied to all spilling strategies as depicted by the middle bars in Figure 3.16, resulting in additional runtime improvements of about 4% on almost all configurations, while preserving the same static spill costs! The spill-everywhere strategies, which place `load` operations right before uses, profit the most, showing mean speed-ups of more than 10%. Manual inspection of the resulting code in all configurations indicates that our heuristic is able to resolve almost all spill-code latency-related violations, without changing the spilling decisions. Some bad cases still remain that will discussed later.

The rightmost bars of Figure 3.16 give the results obtained using accurate profiling feedback combined with the latency heuristic. In other words, we check the performance of the model in a configuration where the static spill costs reflect much closer the actual runtime behavior. In this setting, the runtime of our formulations, i.e. Basic, BasicSSA, Pessimistic, and Optimistic, with and without rematerialization, is 8% to 13% better than Appel-G in the original setup (Appel-G leftmost bar compared to the rightmost bars for our formulations) and 4% to 9% better than Appel-G with profiling and latency optimization (rightmost bar for all). Our optimal formulations show clear improvements in this setting. But not all worst cases are eliminated, as shown in Figure 3.18, which gives the runtime measurements per benchmarks. When profiling feedback is enabled (bottom chart), the number of bad cases for the formulations that are supposed to be better than Appel-G regarding static spill costs (Basic and Pessimistic on these charts) is reduced compared to without (top chart) and the peaks are smaller. These bad cases come from the interaction between spill code and bundling, for one part, and, for the other part, from code placed on critical edges. As already stated, the density of the instruction bundling has a large impact on the runtime. There are even cases where a too-aggressive spilling

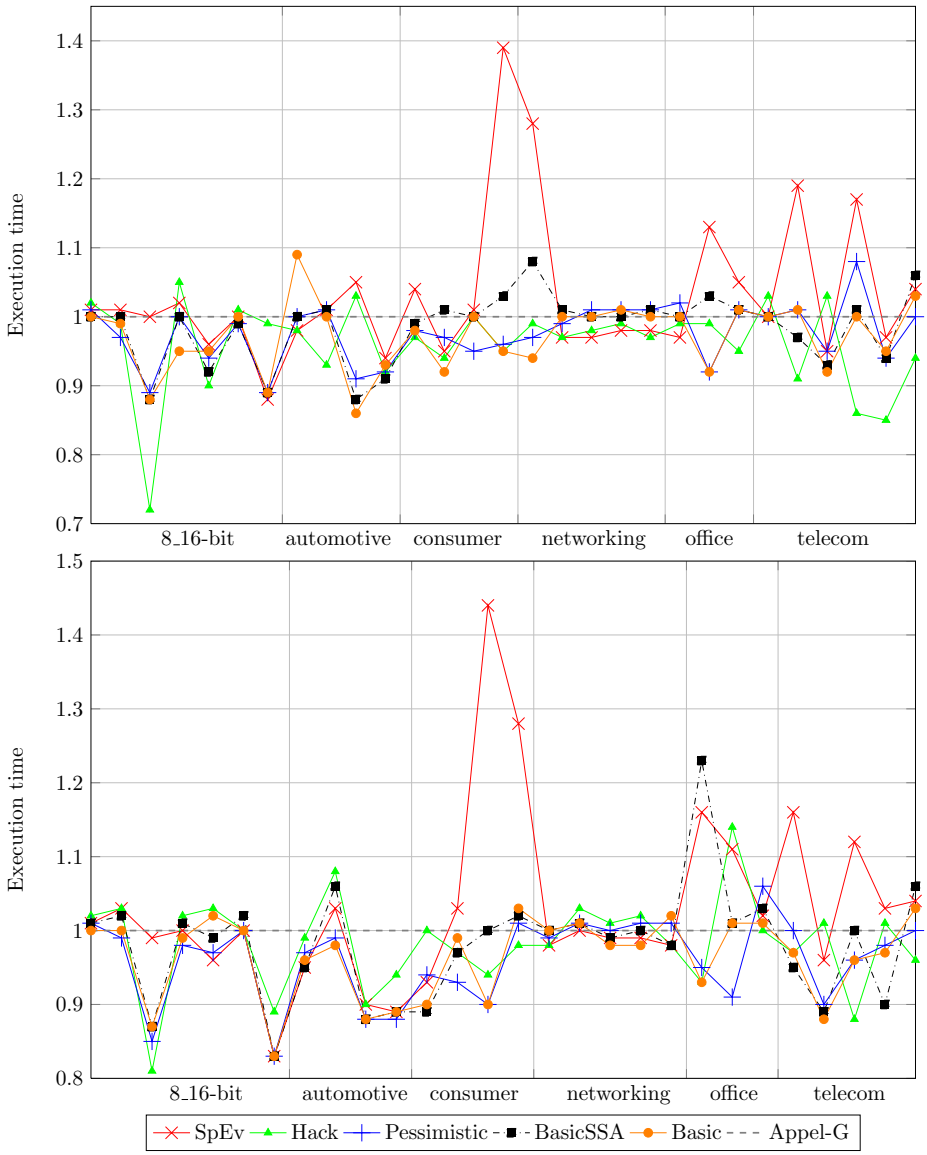


Figure 3.18: Runtime results per bench for EEMBC using frequency estimate (top) and profiling feedback (bottom). In both cases, latency optimization is performed after spilling and before coloring. (Lower is better)

leads to codes where register pressure is at the upper limit on large sequence of instructions, preventing the post-pass scheduler to move anything and hide latencies. As for the second cause of regression, due to critical edge splitting, it is applicable to all targets. When no code is placed on the basic block that is inserted to split such an edge, the block is automatically removed, and so is the related branch instruction. When spill code is inserted in such a block, the overhead of the branch instruction is unavoidable. It may happen that the benefit of placing spill code at that point is negligible compared to the overhead of the branch instruction. Static spill costs do not model this effect, even if it could be done in our ILP. However, even with this potential improvement, we still believe that static spill costs are not a good-enough metric to evaluate the quality of the generated spill code. We will come back to this in Chapter 4.

Note that, at least for the EEMBC benchmarks, Hack's heuristic performs very well, despite a few bad outliers. The same applies for Braun-H heuristic.

3.4 Conclusion

Decoupled register allocation gained much interest due to SSA form and its properties. While SSA provides clear advantages during register assignment, its benefits for spilling were yet unclear. We proposed a new accurate formulation of the spilling problem using integer linear programming, applicable to SSA and non-SSA programs. It is more expressive than previous approaches and, additionally, it offers a great flexibility to model alternative spilling strategies. For example, we can accurately emulate previous "optimal" spilling techniques, as well as strategies used in existing heuristics.

We demonstrated that, if spilling is to be done under SSA, it is preferable to exploit the implicit `moves` in ϕ-functions. Treating SSA live-ranges as unrelated achieves acceptable results on average, however, it has an undesirable worst-case behavior. Formulating the problem *exactly* for static spill costs is intractable at the moment, both because of memory coalescing in non-CSSA programs and of the parallel semantics of ϕ-functions, which can exhibit cycles. We presented two strategies for handling ϕ-functions, an optimistic and a pessimistic one, that provide equivalent or superior results compared to spilling on non-SSA programs, in particular because basic rematerialization is more powerful in SSA.

Our study shows that good improvements can be obtained in terms of static spill cost and dynamic counts. Also, the benefit of rematerialization is considerable. However, runtime improvements for our target VLIW architecture are smaller due to the cost model used for spilling, which does not consider the actual memory latencies and instruction bundling. This issue is often ignored in previous work on spilling and should be studied more closely. A possibility to explore is to define a more accurate cost model for spilling with latency and edge splitting consideration. An alternative is, as we proposed, to move `loads`, heuristically, after spilling but before register assignment.

Chapter 4

Towards a Better Spilling Heuristic

In this chapter, we review several aspects of spilling heuristics to help derive better ones, with respect to actual runtime performance. In the first section (Section 4.1), we discuss spilling criteria, what they model and what their intended use was. In the second section (Section 4.2), we investigate several simplifying assumptions and evaluate their impact on the runtime. We then discuss existing spilling heuristics and their weaknesses in Section 4.3. Finally, in Section 4.4, we give hints to improve the runtime performance of the generated code.

4.1 Existing Spilling Criteria

4.1.1 Static Spill Cost

4.1.1.1 Description

The static spill cost metric estimates the cost for placing `load` and `store` instructions at specific places. It assigns a constant cost to `load` and `store` instructions, typically, the number of cycles needed to execute the related instruction, i.e., the cycles required for computation plus the latency. This cost is weighted by the frequency of the related basic block. Thus, the cost of a spilling instruction is the same everywhere in a basic block. Goodwin and Wilken [52] used this observation to limit the program points to consider for spilling and thus to restrict the search space of their integer linear programming (ILP) formulation. Using this metric, the objective function of a spiller is to minimize the sum of the weighted cost of all spilling instructions.

4.1.1.2 Scope

Let us go back to the characteristics of the cost per instruction. This cost is constant whatever the actual latency will be. Therefore, the model is accurate if and only if the latency is actually paid. This is the case on targets that stall on each memory instruction. However, nowadays, this is usually not the case.

For modern architecture, the behavior of memory instructions depends on the state of the cache. For an off-cache access, the processor stalls until the memory is written in the cache. This is compatible with the static spill cost

model. However, these off-cache accesses are difficult to predict, since highly dynamic. Moreover, the cost accounted in the static spill cost is based on the cache latency, i.e., only a few cycles, whereas off-cache accesses are an order of magnitude longer, typically hundreds of cycles. Nevertheless, spill code is usually small compared to the cache size and is local to a function. That is why this model assumes a perfect cache for spill code accesses, i.e., cache hit latency.

For an in-cache access, the processor continues its execution until it reaches a use of the destination register of the requested memory address. At that point, it has to wait for the remaining latency, if any. However, if it has executed enough instructions, the latency is completely hidden. Therefore, for this kind of architecture, the model is inaccurate unless the spill instruction are placed such that the latency is fully paid. This is the case, for instance, for a `load` instruction placed just before the related use.

To sum up the hypothesis of this model are:

- Perfect cache
- Fully-paid latency

4.1.1.3 Applications

Spill Everywhere Heuristics based on spill everywhere place `load`s just before uses and `store`s just after definitions. Thus, the static spill cost metric in that model makes perfect sense. Indeed, this placement matches the worst case scenario regarding latency.

In graph coloring based allocators [32, 51, 31, 81], this metric is coupled with a notion of profitability. For each variable, its static spill cost is divided by its degree in the graph. This modified cost takes into account how many variables will "benefit" from the spilling of this variable. Thus, for these allocators, this modified cost balances the blindness toward the program structure.

Chow and Hennessy [34], in their priority-based allocator, also used a spill-everywhere model. Like graph-coloring based allocators, they balanced the static spill cost with the length of the live-range.

The objective function with spill everywhere gives a pessimistic cost of the final assembly code. Hence, post-passes optimizations can be used to hide latency or to remove spurious `load` and `store` instructions.

Exact Approaches Goodwin and Wilken [52], Appel and George [3], and more recently Ebner et al. [39] used this metric for their ILP approach. In all cases, the considered machine or placement did not match the hypothesis of the metric. They were able to demonstrate runtime speedup as they were comparing to heuristic-based approaches and, in particular, graph coloring, which is known to produce bad spill code. But, as we demonstrated in Chapter 3, this metric should not be the only goal to achieve good runtime performance. Appel and George faced the same problem but did not push their analysis as far as we did:

> *"Some of the benchmarks have a significant improvement in static spills [...] but no speedup; perhaps this is because we weight the spill costs by static estimation, and perhaps dynamic profiling would significantly improve the performance of the optimal spiller."*

We feel that such conclusion is not enough. Our conclusion is that profiling feedback is not the answer. The problem is that this static cost model is just not good enough to capture latency and post-pass scheduling interactions.

4.1.2 Furthest First

4.1.2.1 Description

The furthest-first method drives the choices of spillers by next-use distance. The underlying idea is that the furthest is a next-use, the longer the related variable will decrease the register pressure, if it is spilled. Thus, the cost of spilling code is not the primary objective. The important and difficult part of furthest-first-based heuristics is to choose the right next-use distance metric.

4.1.2.2 Scope

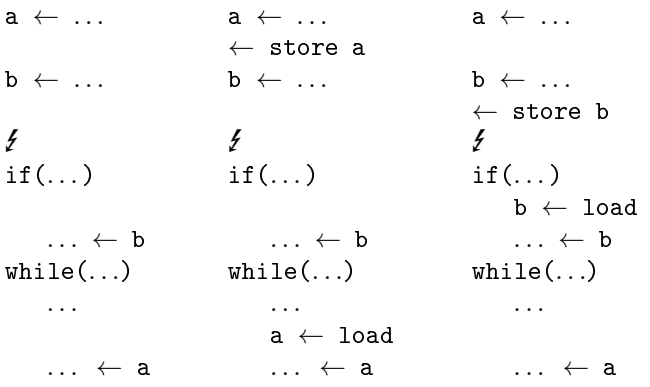
Originally developed for paging with write backs [7], the furthest-first method was then adapted to local register allocation by Farach-Colton and Liberatore [43]. Local register allocation deals with basic blocks, thus straight-line code. In that context, the next-use distance makes perfect sense. Moreover, if the latency of `load`s is not considered, all spilling instructions have the same cost and this cost does not depend on where they are placed since on straight-line code all program points have the same frequency. Thus, minimizing the number of spilled instructions also minimizes the amount of spill cost. Note that `load`/`store` placement for straight-line code is already NP-complete [43]. However, Guo et al. [54] showed that even if it does not minimize the number of `load`s and `store`s, this heuristic perform quite good in that context. In addition, an interesting side effect of the furthest-first method is that it tends to spill a variable for which the number of bundles or cycles before the next use is going to be large and, as a consequence, the post-pass scheduler is more likely to have more freedom to schedule the corresponding `load` and hide its latency.

4.1.2.3 Applications

Straight Line Code and Linearized Code As already stated, the direct usage of this method for register allocation appears in Farach-Colton and Liberatore's work [43]. Later, this criterion has been used in linear-scan approaches [89, 105, 85]. In this context, the register allocation is performed on the whole program but the program is viewed as one big basic block, according to a possible linearization. The live-ranges are expressed on this large basic block, thus over-approximating the actual liveness. Despite the fact that the spilling cost is not homogeneous on this big basic block (some of the actual blocks are more frequent than others), linear scans use this criterion with respect to its original idea. They spill the live-range that will help to reduce the register pressure for the longest "interval", regardless of the actual spilling cost. Spilled variables are spilled everywhere to simplify the process.

Post-processing phases may be used afterwards to improve the solution [105]. Unlike graph-coloring-based approaches, a spilled variable is guaranteed to help reducing the register pressure. Indeed, it helps at least on the current processed point. However, linear-scan approaches are known to produce fairly bad spilling code. But their goal is mainly to allocate the code as fast as possible with a small memory footprint.

General, Not Linearized, Programs Hack et al. [58] proposed to extend the next-use distance to general programs. They defined the next-use distance as



(a) Original code (b) Min furthest first (c) Static spill cost

Figure 4.1: Counter-example of the efficiency of the furthest-first criterion (b) extended by Hack et al. [58] versus static spill cost (c).

the minimal number of instructions over all possible paths leading to a next-use. However, they did not demonstrate any runtime improvements in their paper. We can easily build examples where a spiller using their criterion performs worse that a spill-everywhere strategy using static spill cost, as depicted in Figure 4.1. This is not surprising, as there is no frequency consideration in their model. Nevertheless, using their method, one can perform a `load`/`store` placement and not just a spill-everywhere optimization.

Braun and Hack [21] furthest extended the previous model. They proposed to add a length on edges during the computation of the next-use distance. They set a long length for edges that exit loops. The idea is to consider that uses in loops are closer than uses outside a loop. Doing so, it is more likely that variables in a loop will be kept in registers. However, this extension is not able to avoid the bad pathological case of Figure 4.1.

From our point of view, both extensions lack a key point. The notion of profitability based on the fact that the interval that is not used the longest should be spilled is no more true for code that is not straight line code. Indeed, live-ranges spawn several branches and of course, the minimal distance to the next-use does not mean that it does not have the biggest not used interval. Figure 4.2 shows an example where spilling choices are bad because the initial spilled variable, b, does not have, globally, the longest live-range. To match the original spirit of the furthest-first method, we think that the total length of the live-ranges may be taken into account instead, not just a distance along one particular path. Indeed, this metric is more representative of the extent of the live-ranges where it helps reducing the register pressure. Moreover, a tweak can be made to take into account the frequency, or at least the nesting of loops, with the length of edges as proposed previously, as well as points where register pressure is high and where it is not. Taking into account the fact that the latency of a `load` can be hidden or not may be interesting too.

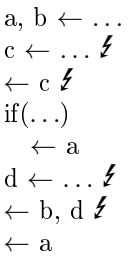


Figure 4.2: Spilling choice based on a furthest-first criterion with minimal next-use distance will choose to spill b. This choice does not help for the second over-pressured point around the live-range of d, thus a has to be spilled too.

4.2 Simplifying Assumptions

In Chapter 3, we investigated several methods to simplify the way we can handle the ϕ-functions of static single assignment (SSA). We do not come back here on that aspect. Instead, in this section, we check the impact on runtime of several assumptions, made in the literature or that we introduced, and that may degrade the static spill cost. To do that, we used the experimental setup and our ILP formulation defined in Chapter 3 in two different configurations: basic (non-SSA) and pessimistic (SSA), followed, optionally, by our post latency optimization and/or using profiling feedback. See Chapter 3 for details on the ILP formulation. Also, as explained in Chapter 3, we will report runtime numbers for EEMBC benchmarks only.

4.2.1 The Instruction `store`

On most architecture, `store` instructions are an order of magnitude cheaper than `load` instructions. Thus, it is a common practice for `load`/`store` placement heuristics to simplify the handling of `store`s.

Considering That `stores` Are Free The biggest simplification consists in considering that `store` instructions are free. In that setting, heuristics focus only on the placement of `load`s according to their spilling cost model. Note that, even if they are free, useless `store` instructions are of course not inserted in the different methods we evaluated.

The red curve with square markers in Figures 4.3 to 4.8 shows the impact of this assumption on the runtime. [1] For the original Basic configuration, given in Figure 4.3, i.e., with frequency estimate and without post latency optimization, this assumption produces only few worse cases (5 over 38) that are above 5%. This assumption shows comparable impact in the Pessimistic configuration, given in Figure 4.4. In both cases, on average, assuming that `store`s are free does not change anything on the runtime, even if analyzing more precisely each individual point may reveal interesting special cases.

Coupled with our post latency optimization, this assumption shows a similar impact on runtime for both configurations, see Figures 4.5 and 4.6. Indeed, the

[1] In all these figures, benchmarks are sorted so that one of the curves ("`store` at definition", then "Base") increases. This is an arbitrary choice to make the figures more readable.

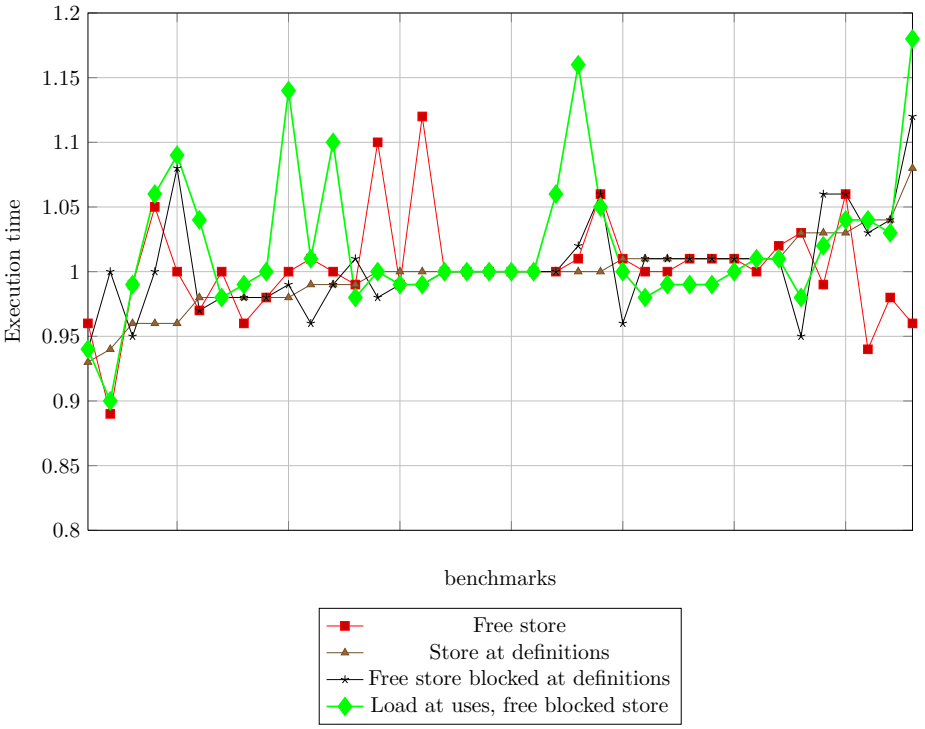


Figure 4.3: Impact of different simplifying assumptions on the runtime for Basic configuration, over all benchmarks (horizontal axis). Numbers are normalized to Basic configuration (original version) with frequency estimate. (Lower is better)

benchmarks where this assumption produces a runtime worse than the same setting (blue curve with circle mark) are limited. Finally, when the ILP works with accurate frequencies (i.e., with profiling feedback), the impact remains limited, see Figures 4.7 and 4.8. Again, on average, this assumption does not change anything. This is not so surprising because assuming that a `store` is free does not depend on the frequency estimation. Note also that these figures illustrate again the weakness of the static spill cost model, i.e., the model used in our ILP, since, although this simplification is a restriction, the runtimes improve in a few cases. Similarly, the Base curve, i.e., with accurate frequency, sometimes produces a few cases worse than with the same configuration without profiling feedback, i.e., with inaccurate frequency.

To conclude, assuming that a `store` costs nothing seems to be a reasonable assumption. However, to our knowledge, this assumption is never used alone, it is coupled with other assumptions as we will see now. It is indeed still important to not let the formulation or heuristic place the `store` anywhere in the code, even if it is considered as free.

Placing `stores` at Definitions A common assumption in spilling heuristics is to place a `store` at each definition point of the related variable. In particular, all heuristics that use a spill-everywhere approach, from linear scan to puzzle solving through graph coloring [32, 51, 85, 89], use that simplification. This is

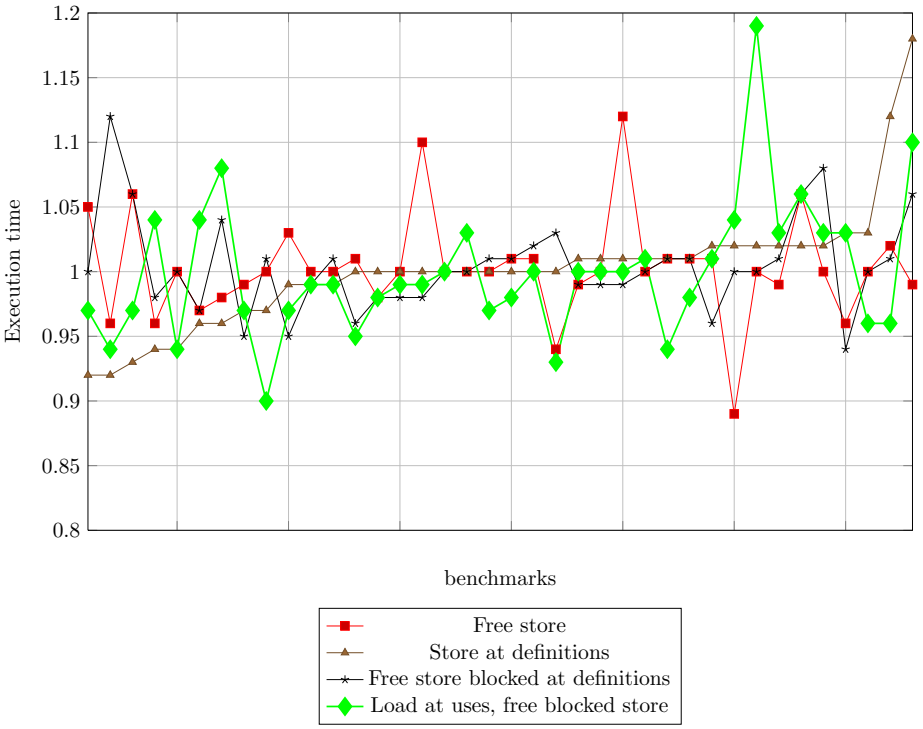


Figure 4.4: Impact of different simplifying assumptions on the runtime for Pessimistic configuration, over all benchmarks (horizontal axis). Numbers are normalized to Pessimistic configuration with frequency estimate. (Lower is better)

also true for more recent SSA-based spilling approaches [39, 55, 21]. Moreover, this simplification may be combined with the previous assumption, as in the progressive spill-code placement of Ebner et al. [39].

The brown curve with triangle markers in Figures 4.3 to 4.8 shows the impact of this assumption on the runtime. The black curve with star markers demonstrates the impact when coupled with the previous assumption (free `store`).

Alone, this assumption has a very limited impact on the runtime and in particular when performed under non-SSA programs, as depicted in Figure 4.3. Under SSA, there are some very bad cases, see Figure 4.4, with 2 benchmarks slowed down by more than 10%. Here, the fact that the frequency is estimated plays a role as we will see. When assuming that `store`s are free, the impact remains limited, as on average it does not change anything, but, still, a few worse cases are observed. The situation is a bit worse in SSA, mainly because SSA codes have more definitions (due to ϕ-functions), some of them harder to schedule in bundles, e.g., if several `store`s are inserted at the same place.

When our post latency optimization is enabled, as shown in Figures 4.5 and 4.6, the cases that are worse, i.e., the points above the Base curve (blue curve with circle markers), are more limited. Indeed, this optimization also helps to schedule the `store`s more freely. We directly see here the beneficial impact of this latency optimization. Therefore, coupled or not, these different assumptions make perfect sense for an heuristic in that configuration, i.e., with inaccurate frequency and post latency optimization.

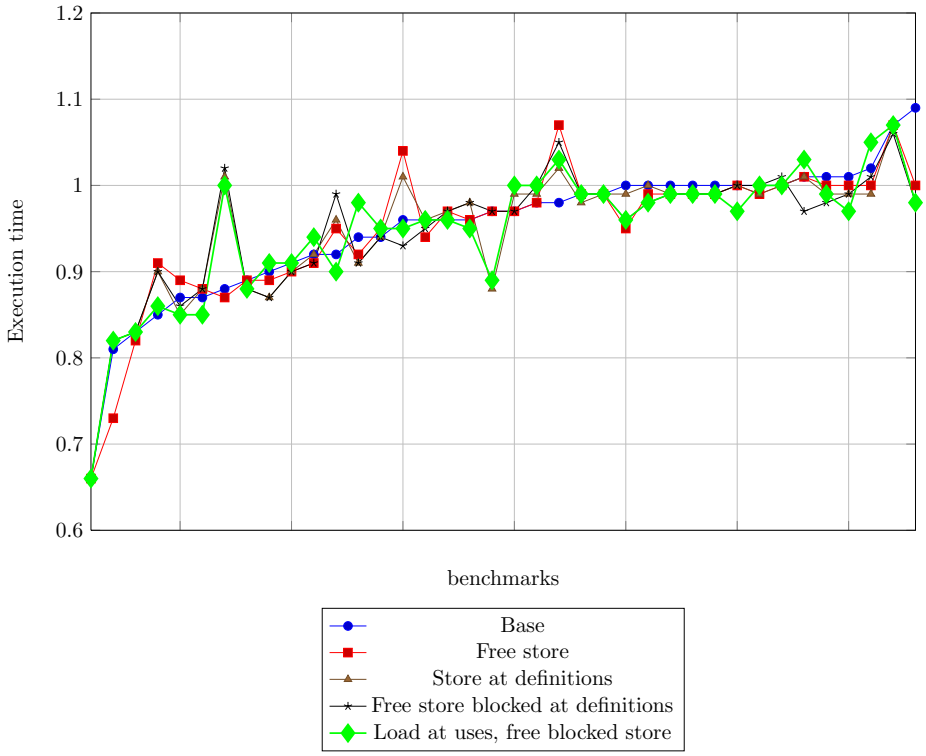


Figure 4.5: Impact of different simplifying assumptions on the runtime for Basic configuration followed by post latency optimization. The "Base" curve presents Basic in that configuration without any simplifying assumptions. Numbers are normalized to Basic configuration with frequency estimate. (Lower is better)

As expected, with profiling feedback, the results are even better for the simplification that considers `stores` only at definition, since its cost depends on the block frequency. Also, as already stated, due to the inaccuracy of the static spill cost model, the combination of both simplifications may sometimes perform even better than the general formulation based only on spill cost: a larger static spill cost, due to these limitations, may still produce a faster code.

4.2.2 The Instruction `load`

As `load` instructions are usually considered as expensive, the way they are handled involves in general fewer simplifications. Based on the static spill cost metric, Goodwin and Wilken [52] demonstrated that the optimality of this metric is preserved if the insertion points of `load` instructions are limited to the end of basic blocks or just before the related uses. This is easy to understand because the static cost is the same for all program points of a given basic block (the latency is not taken into account in such a cost). Spill-everywhere-based heuristics use even more limited insertion points: `loads` are inserted just before *all* related uses, even if the variable has been already loaded earlier. It is possible however to eliminate these redundant `loads` afterwards [10]. The SSA-based spilling heuristic of Braun and Hack [21] does not explicitly limit the insertion points of `load` instructions. However, by construction, it always

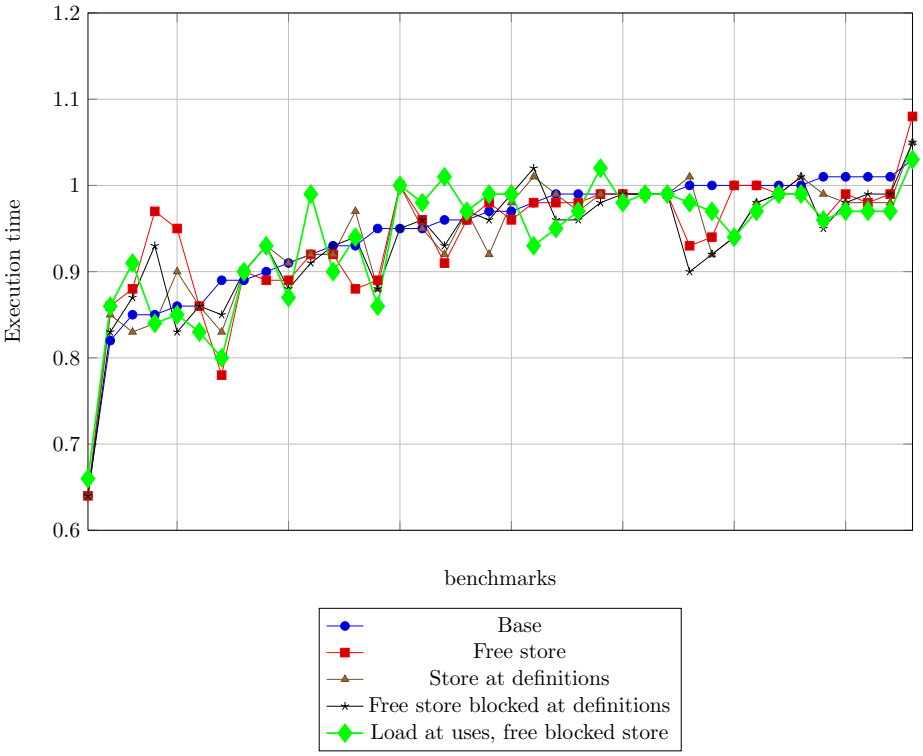


Figure 4.6: Impact of simplifying assumptions on the runtime for Pessimistic configuration followed by post latency optimization. The "Base" curve presents Pessimistic in that configuration without these simplifying assumptions. Numbers are normalized to Pessimistic with frequency estimate. (Lower is better)

inserts `loads` on edges, i.e., at the end of the previous basic blocks (there are no critical edges), or just before uses. But a previously-loaded variable can be reused, to avoid a redundant `load`.

Placing `loads` at Uses We decided to test a model simpler than the model of Goodwin and Wilken, and of Braun and Hack, but more general than a spill-everywhere strategy: in our ILP formulation, we limited the insertion points of `loads` just before uses but unlike the spill-everywhere approach, we did not force that every use of a spilled variable must be preceded by a `load`. Basically, this is equivalent to an optimal, with respect to the static spill-cost model, spill-everywhere approach coupled with a redundant `load` elimination optimization. Moreover, we used the assumption that `stores` are free and blocked at definitions as we showed they were valid simplifications for runtime performances.

The impact of this approach on runtime, in both a SSA and non-SSA context, is given by the green curve with diamond markers in Figures 4.3 to 4.8. Since this approach uses the frequency to determine the expected cost of a `load`, the accuracy of this information has an impact on the runtime. Moreover, as shown in Chapter 3, forcing `loads` to be placed just before uses produces the worst possible latency. Thus, it is not surprising that the performance of this simplification in the original setting is quite bad, as shown in Figures 4.3 and 4.4.

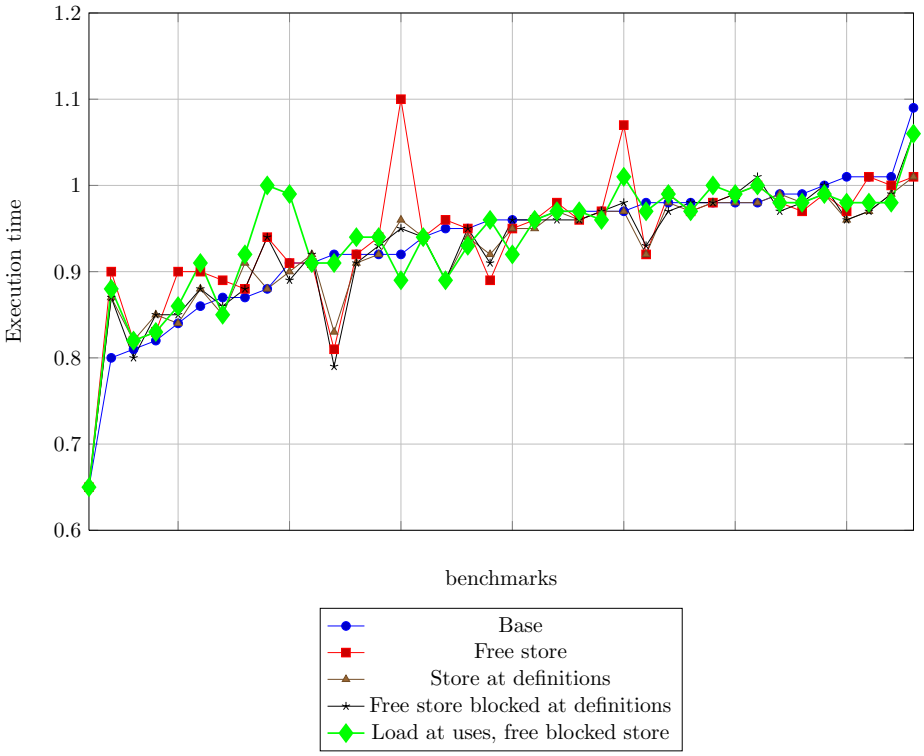


Figure 4.7: Impact of simplifying assumptions on the runtime for Basic with profiling feedback and post latency optimization. The "Base" curve presents Basic in that configuration without these simplifying assumptions. Numbers are normalized to Basic with frequency estimate. (Lower is better)

This observation completely changes when our latency optimization is enabled, as shown in Figures 4.5 and 4.6. In this setting, this simplification is, on average, as fast as its baseline (Base curve in blue with circle markers) but with a few worse cases. The same pattern can be observed when profiling feedback is enabled, see Figures 4.7 and 4.8.

In conclusion, we believe that this model is quite accessible to heuristics. Moreover, we showed that its impact on runtime, as soon as it is coupled with a latency optimization, is limited on average compared to an optimal model based on static spill cost. Therefore, we suggest to investigate this simplified model for a spilling heuristic. This has still to be explored.

4.3 Existing Heuristics

4.3.1 Graph Coloring

In graph-coloring-based approaches, spilling occurs when coloring fails. In this model, spilling is an ad-hoc mechanism plugged into a heuristic used for a problem (coloring), which is NP-complete for general graphs [33]. In particular, the effect of spilling is not well captured by the graph model. Indeed, unless the target machine can use memory operands, which is usually highly constrained when possible, spilled variables have to reside in registers at their definitions

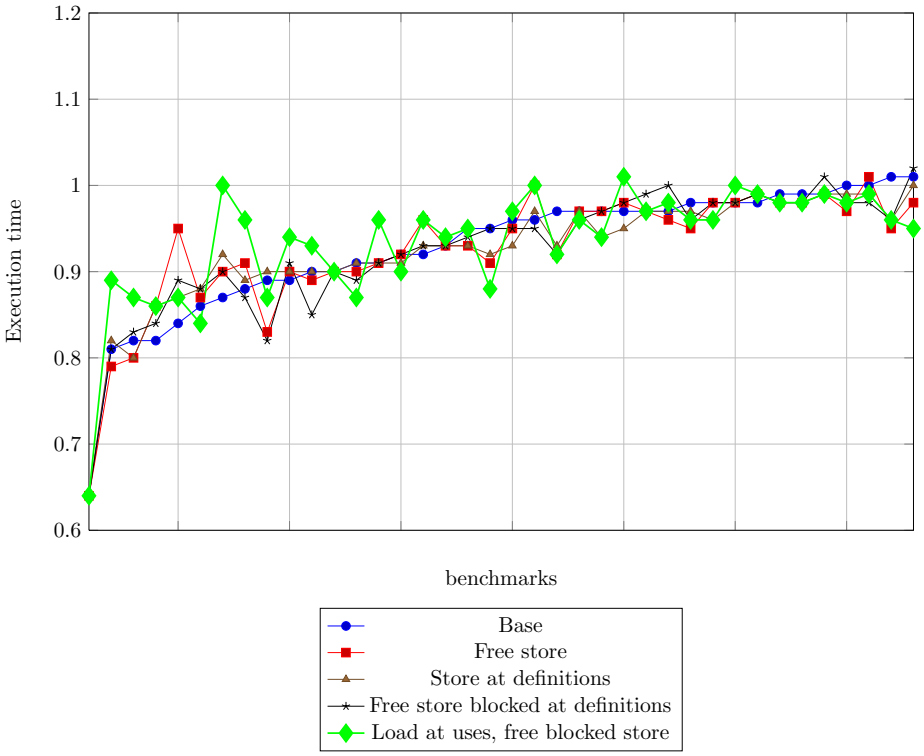
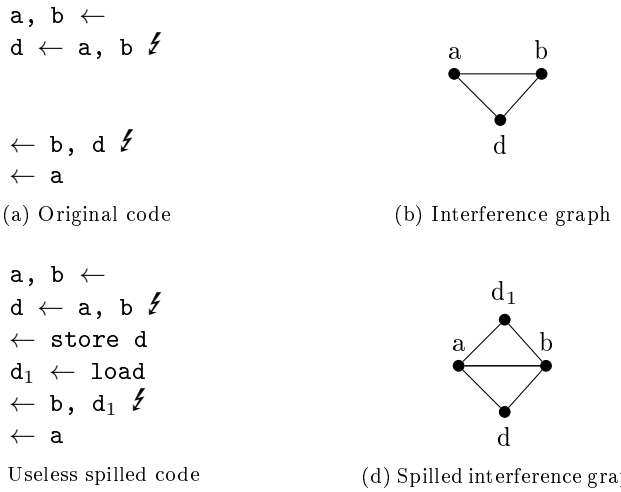


Figure 4.8: Impact of simplifying assumptions on the runtime for Pessimistic with profiling feedback and post latency optimization. The "Base" curve shows Pessimistic in that configuration without these simplifying assumptions. Numbers are normalized to Pessimistic with frequency estimate. (Lower is better)

and uses. These additional short live-ranges imply nodes that are not represented in the graph and require to rebuild and redo the whole approach after every spilling phase. This problem is known as spilling with *holes*, where holes represent chunks of the memory live-ranges that must reside in register. In other words, when a variable is spilled, a live-range with holes is placed in memory, and not the full live-range. Without holes, on chordal graphs, e.g., those generated by SSA programs, an optimal spill-everywhere strategy can be produced in polynomial time when the number of registers is fixed [15]. However, the problem on general graphs or with holes, i.e., the most common cases, are NP-complete [15]. With holes, the expected benefits of a spilled variable may completely vanish leading to overspill as illustrated in Figure 4.9. It is possible to build more complex examples where spilling a variable helps to reduce the chromatic number of the interference graph in a given iteration of the simplification process, but becomes useless later as another variable is spilled.

It is possible to emulate a spilling problem without holes. For that, one can reserve some registers to materialize the spilling code. Chow and Hennessy [34] used this simplification for their priority-based coloring. However, doing so decreases the number of possible colors that can be used when allocating the variables to registers, and thus possibly increases the amount of spill. Therefore, unless the target machine has a lot of registers, this is generally a bad idea.



(a) Original code

(b) Interference graph

(c) Useless spilled code

(d) Spilled interference graph

Figure 4.9: Spill-everywhere strategy coupled with graph-coloring model may produce useless spill code. All variables have the same degree, but d has fewer uses. Thus, the spill cost metric is the cheapest for d, which is chosen for spill.

To tackle the holes and the ineffective spill code problems, it is possible to explicit in the interference graph (IG) the parts of the live-ranges that reside in registers using (parallel) copy instructions. To our knowledge, this method is not used because it blows up the size of the IG, which is a major concern of graph-coloring-based approaches as it impacts their runtime and memory footprint as well as the quality of the produced results (see Chapter 6).

Since graph-coloring-based approaches perform everything in a single phase, post phases may have limited opportunities to optimize the generated code. Indeed, the allocated code is more constrained for scheduling than before allocation. In this situation, a post latency optimization may be blocked because of the reuse of registers (anti dependences also known as write after read dependence). Similarly, `load` elimination optimization has fewer opportunities to extend live-ranges, and so on.

4.3.2 Scan-Based Approaches

Scan approaches do not use an abstraction of the program, but work directly on the program. As already stated, spilled variables help to decrease the register pressure of at least one program point. This is a first advantage over graph-coloring-based approaches. The original linear-scan approach was proposed by Poletto and Sarkar [89]. Its main weakness is that it over-approximates the actual live-ranges, increasing the register pressure, hence the amount of spill code. Moreover, it reserves some registers for the whole program to emulate a target featuring memory operand, thus enabling spill everywhere without holes. These over-approximations are made for speed purposes.

Mössenböck and Pfeiffer [76] adapted the linear-scan approach to SSA-based program. The notable improvement in terms of spill compared to the original algorithm concerns the modeling of live-ranges. The live-ranges can contain holes in the linearization of the program. Later, Wimmer and Mössenböck [105]

introduced on-the-fly live-range splitting. This has two advantages. First, it reduces the number of registers needed globally as a variable can change its register on the fly. Second, they insert a split point before the next use of a variable before spilling it. This results in spilling only this sub live-range, thus yields better spill code than with a spill-everywhere strategy.

In 2007, Sarkar and Barik [95] in their extended linear-scan feature an extensive live-range splitting framework to take advantage of all possible split points for coloring. However, when they choose to spill, they do it in a spill-everywhere fashion. Pereira and Palsberg [85], in their puzzle-based allocator, used a similar approach but handle register aliasing.

Finally, to take spilling decisions, Barik [6, Ch.6] proposed a bipartite liveness graph that is more compact and more expressive than interference graphs. On the left-hand side of this graph, there are variables, on the right-hand side, the end point of each *basic interval*, i.e., a part of the live-range that is contiguous in the linear ordering. The variables are connected to the end points where they are alive. Then, each program point where the degree of the end points (i.e., the right-hand side points) is at least k (the number of registers) is considered to take a spilling decision: the heuristic starts from the point with the largest frequency and spills one of the connected variable with the smallest spill cost, in a spill-everywhere fashion. This way, the heuristic forces the variables that are highly executed to be kept in register.

All these approaches deal with global register allocation, in a non-decoupled fashion, thus the difficulties that we pointed out for graph-coloring-based approaches regarding post phases also apply here.

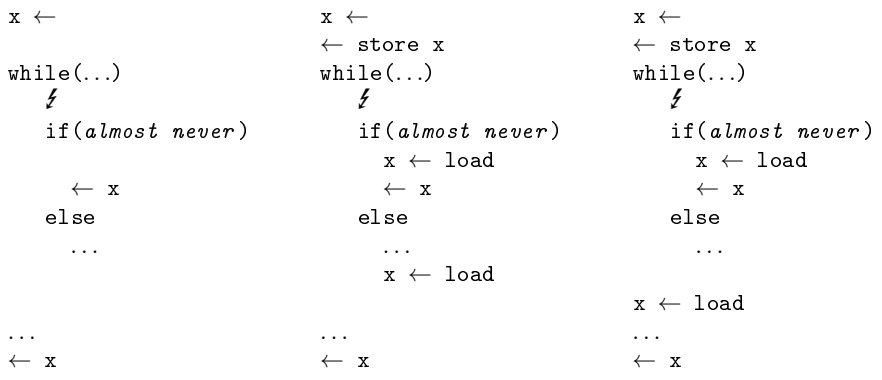
4.3.3 Decoupled Approaches

To our knowledge, excluding the exact approaches [3, 65], only two decoupled spillers were proposed so far and both work on SSA programs.

Hack et al. [58] defined an heuristic that performs a furthest-first analysis, as discussed in Section 4.1.2.3, on each basic block independently. At the end of this process, each basic block holds a set of variables that are in registers. Using this information and the input set chosen for each basic block, the required `load`s are placed on edges of the control flow graph (CFG) to match the occupancy sets. This approach lacks global information during the local analysis, resulting, potentially, in a lot of `load` instructions on CFG edges.

Braun et Hack [21] extended the furthest-first criterion for the whole program, as explained in Section 4.1.2.3. We already pointed out some of the weaknesses and advantages of their approach there. More generally, this approach is compile-time and memory-footprint efficient. Thus, it may be a good match for a decoupled register allocation in a just-in-time (JIT) compiler. Of course, as for all heuristics, it suffers some bad cases. One source of such behavior stems from the fact that this heuristic tries to saturate the register file, i.e., to reload variables as soon as there is the space to do so. Figure 4.10 shows how this can generate a bad spill code.

Both methods offer good opportunities to apply optimizing post phases as the resulting code is not yet constrained by register assignment, only the register pressure is guaranteed to be low enough everywhere.



(a) Original code (b) Braun and Hack spiller (c) Optimal spiller

Figure 4.10: Bad spilling decision in Braun and Hack spiller [21] due to the policy of saturating the registers. At the joint point of the `if-then-else` in the `while` loop, x is available in register on the path where it is used. Since the other path has room for this variable, it is reloaded there. Thus, x is available in register at the exit of the `while` loop and no reload is necessary for the final use. However, it would have been cheaper to reload x outside the loop as shown in (c). Here, reloading x outside the loop would trigger the possibility to eliminate the useless `load` by dead code elimination. This is not generally true.

4.4 Improving Runtime

This section is a collection of observations and ideas to help improving the runtime performances of the generated code.

4.4.1 Latency

As we pointed out in Section 4.1, hiding latency is a key point to improve runtime performances. Here we give the bases to account for latency in existing spilling heuristics based on a spill cost.

As explained in Section 4.1.1.1, the static spill cost (the metric that states how expensive a spill will be) is usually composed by the cost of the instruction plus its latency. It is tempting to change the accounted latency with respect to the minimal distance to the next use. This straightforward approach is not realistic for the following reason. Actually, the target machine stalls at the point of the use of the loaded variable. Hence, the remaining latency is paid at this particular point and not at the place where the `load` has been issued. Moreover, it remains some latency only if the path to the use issued a `load` and not each time the use is processed. In other words, the remaining latency is paid at the use point but not as frequently as the use is executed.

From this description, we can derive the following `load` cost: the base cost of the instruction multiplied by the frequency of the point where it is placed plus the maximum, over all next uses, of the remaining latency towards this use multiplied by the path frequency to this use. As we are accounting for the maximum couple (remaining latency, probabilities of its occurrence) in that formula, we have a worst-case perspective of the runtime, assuming a fixed

```
a ←
...
↯
l1
while(often)
    ...
    u1 : ← a
u2 : ← a
```

(a) Input code

Use	Remaining latency
u_1	1
u_2	2

(b) load-use distance if `load` at l_1

Model	Latency cost		Cost
	u_1	u_2	
Static spill cost	3		4
Latency at use	$f_{u_1} * 1 = 50$	$f_{u_2} * 2 = 2$	51
Path to use	$f_{l_1} * p_{l_1\ to\ u_1} * 1 = 0.5$	$f_{l_1} * p_{l_1\ to\ u_2} * 2 = 1$	2

(c) `load` cost at l_1 where f denotes the frequency and p the probability with $f_{l_1} = 1$, $f_{u_1} = 50$, $f_{u_2} = 1$, $p_{l_1\ to\ u_1} = 0.5$ and $p_{l_1\ to\ u_2} = 0.5$

Figure 4.11: Impact of the latency model in the optimized `load` cost. The same `load` in a given code (a) may have very different expected cost depending on the model (c). All these models work statically, using or not the static information of the remaining latency at a use point (b).

schedule, a perfect cache, and that the coloring phase will not remove or insert instructions. Figure 4.11 gives an example of the different models of latency.

In the proposed models, the remaining latency from a given point to each use is a critical information. This information is more complex to compute as it may appear at first glance. Indeed, the insertion or removal of instructions may change this information. With a fixed schedule, it is tempting to ignore these possibilities as things are not supposed to move around. In fact, even with this assumption, instructions may still appear/disappear because of live-range splitting or coalescing that are usually performed during the assignment phase of a decoupled register allocator. We now discuss these hard-to-model effects.

Unless a newly-inserted instruction uses a loaded variable, its impact on latency is always beneficial. Indeed, it increases the `load`-use distance of other variables or, said differently, it uses the remaining latency cycles to perform its computation, improving the overall *code productivity* (ratio stall/computation). However, if this instruction uses a loaded variable, the cost of the `load` may be worse than expected in the model. This is the case if a `move` is inserted just after a `load`. A possibility is to bias the coalescing process (the phase that tries to assign the same registers to the two variables involved in a `move`) by increasing the weight of such a `move` to take into account the remaining latency penalty. Note that even if this is not a newly-inserted instruction, this bias will improve the runtime behavior of the program, with respect to the accuracy of the frequency estimation, as it may reduce the expected remaining latency.

Another situation is that, in case of coalescing, the related copy instructions may disappear and thus may shorten some `load`-use distances as seen before spilling, resulting in a worse runtime behavior [2]. To avoid this degradation of

[2] More precisely, it impacts the productivity of the code (i.e., there are more empty slots

the performance, when computing the remaining latency for a given variable v, we choose to consider the copy instructions in a conservative way as follows: if the copy does not use v, we assume it will disappear, thus it does not decrease the remaining latency, and if it uses v, we assume it is a regular use and will produce a stall if the latency is not covered. This model is pessimist since it assumes that the copies using v will not be coalesced and other copies will not hide latency (this is the case if they are finally all coalesced).

Of course, this model is not perfect. In particular, due to its static nature, it does not capture the insertion of other spilling instructions. However, it remains conservative as inserting instructions will decrease the remaining latency for others and thus will produce better expected solutions. Moreover, a spilling heuristic that inserts spilling instructions on the fly may be able to account for these new instructions for other insertions. However, to be applicable for our experimental target, this model has to be refined as it does not take into account the instruction level parallelism available in the processor. Indeed, on our target, the latency is not eaten by instruction but by *bundles*. Thus, a heuristic must have the knowledge or, more likely, an estimation of the bundles, as bundles may change when spilling instructions are inserted. Moreover, it should take into account the density of the bundles, i.e., how much room it remains in bundles to hide the cost (not the latency) of spilling instructions. Indeed, when filling a hole in a bundle, a spilling instruction may be completely for free, regardless of the frequency of the bundle.

4.4.2 Helping the Scheduler

The runtime performance depends on the quality of the instruction schedule that is finally produced, which in part depends on the freedom that the scheduler has to place instructions. It is hard to tune the scheduler for register allocation and vice versa. The scheduling performed prior to register allocation, usually called the pre-scheduler, has to balance the actual schedule length and the expected impact on the spilling cost (with a metric based on register pressure). On the other hand, after register allocation, the post-scheduler has limited scheduling possibilities as the code is over-constrained by register reuse.

From that perspective, the post latency optimization, which moves `load` and `store` instructions after spilling but before register assignment, helps the scheduler to produce a shorter schedule length or a denser sequence of instructions. Following the same idea, we gave in the previous section a general cost model, which aims at hiding the latency at a more global scope than just basic blocks. However, none of these approaches is able to move other instructions around. Thus, if the pre-scheduler just tried to minimize the length of the live-ranges – a classical approach before register allocation – the allocated code may result in very bad runtime performances even if the spiller is very good. This problem is even more relevant on targets that feature instruction level parallelism as ours. Indeed, sequences of code that can be executed in parallel prior to register allocation may be completely sequentialized because of a too-tight register pressure that created register dependences between unrelated instructions. This is one beneficial side effect of spilling heuristics on runtime, compared to optimal approaches as studied in Chapter 3. Indeed, spilling heuristics usually

in bundles), not necessarily its global runtime behavior.

over spill compared to optimal approaches and, as an unexpected consequence, leave more freedom to the post-scheduler to hide latencies. Analyzing by hand some of the final codes produced by the different approaches, we indeed found cases where this situation occurs: the static spill cost is much better with the optimal solutions but the heuristics can still produce codes with an equivalent, or sometimes better, runtime performance, because the degradation in static cost is compensated by the post-pass scheduler, which has more freedom to hide latencies.

From our point of view, it may be interesting to have an estimation of the schedule length and the degree of parallelism, if any, during the spilling algorithm. Therefore, on hot spots, like loops, it may be interesting to spill more that the fixed k limit so that the post-scheduler has more freedom to schedule instructions. For instance, one can spill more live-through variables. An alternative would be to design a scheduler with "local" spilling/recoloring capabilities after register allocation or after the spilling phase. Fully integrating scheduling and register allocation is well-known to be expensive, if not intractable. However, in the context of a decoupled register allocation, integrating scheduling and spilling, before register assignment, seems an interesting option to explore.

4.5 Conclusion

In this chapter, we showed the limitations and scopes of the existing criteria used for spilling. In particular, we saw that in many existing spilling approaches, the related spilling criterion is used outside its scope, thus making difficult to predict the benefits on the runtime. In the case of the furthest-first criterion, we gave some hints on how to extend its scope to achieve better runtime performances.

We then empirically validated some simplifying assumptions. In particular, we showed that, if a post-spilling latency optimization (as explained in Chapter 3) is available, it is a reasonable simplification to restrict to `load`s placed just before the related uses, when optimizing the cost of the spill instructions. We believe that this simplification may be interesting to design an efficient heuristic, in particular for architectures with bundles where `load`s can sometimes be completely hidden. With an adequate live-range splitting, this can be seen as a spill-everywhere problem (on the variables created by this live-range splitting) where `store`s are free and fixed at the definition point of the original variable. This splitting can be obtained by adding, after each instruction, a new virtual definition of all its arguments, coupled with the static single information (SSI) representation [98, 2].

We pointed out that the abstraction of live-ranges induces an overestimation of the register pressure in the case of linear scan and of the benefits of spilling in almost all spill-everywhere approaches. For the latter, this assumption may require to reserve a register, or several registers depending on the algorithm, to emulate the spill-everywhere principle even when spilling is not necessary. We also emphasized the interest of decoupled register allocations due to their use of live-range splitting and the possibility to help the scheduler before assigning the registers to variables.

Finally, we discussed a latency cost model to be used as an objective function in spilling heuristics. This model is not intended to be used with the proposed simplifying assumptions but should be seen as another way to tackle the spilling

problem. We also proposed to explicitly spill more variables to help the scheduler to reduce the schedule length in the regions of the program that are frequently executed. All these aspects are still to be explored. Indeed, optimizing for runtime performance remains a difficult task. As we explained, in addition to the fact that, given a cost model, the different optimizations are in general NP-complete, the design of the model is itself difficult: how expensive a `load` will be is hard to measure, depending on the architecture and the post-pass scheduling.

Part III

Coloring with Affinities and Antipathies

The spilling phase is done. As a result, the register pressure is nowhere greater than the number of registers. The challenge now consists in assigning registers to the variables so that no additional spill code is generated. Moreover, this assignment must respect the encoding, application binary interface (ABI), and register aliasing constraints, while optimizing the performance of the code, in particular, optimizing the register-to-register moves (coalescing).

In this part, we show that neither complex algorithms nor extensive live-ranges splitting are required to handle such constraints, even in existing graph-coloring-based allocators. The first chapter of this part (Chapter 5) deals with encoding and ABI constraints. We present an extension of the interference graph (IG) model to tackle these constraints without inserting any additional split points. We show how to apply the same mechanism on scan-based approaches. As a bonus, we define a new allocator, called *tree-scan* , that may be suitable for just-in-time (JIT) compilation. In a second chapter (Chapter 6), we focus on register aliasing constraints. We show how they can be handled in a decoupled graph-coloring-based allocator without the size explosion of the intermediate representation (IR) usually implied by these constraints.

Chapter 5

Coloring with Encoding Constraints

An important detail of the register assignment process is register constraints imposed by the instruction set architecture (ISA) or the application binary interface (ABI). For example, the first integer argument of a function call on the ARM Linux ABI must be passed in register R_0. Similarly the division in IA32 requires the source/destination operand to reside in register %eax and %edx. Instruction sets may also impose two operands of the same instruction to use the same register (two-address mode). These constraints, referred as *operand pinning*, are local to instructions and are usually handled *prematurely* by the allocator by splitting live-ranges, i.e., by introducing copy instructions, prior to assignment. This places additional pressure on the coalescing to eliminate as many of these extra copies as possible. Moreover, coalescing is the most costly task of register allocation [23, 51] and is NP-complete (even with 3 registers) [17, 55].

This chapter proposes a new technique called *repairing* that deals with local register constraints without requiring preliminary live-ranges splitting. We emphasize that repairing is useful when certain instruction operands are restricted to a subset of registers, possibly a singleton [70, 3, 51, 105]. The idea is to *relax* register constraints during allocation and *repair* only afterward those that have been violated. This approach allows to handle register constraints without loosing the benefits of the elegant formalisms that have made graph coloring [33], linear scan [89], and decoupled register allocation based on static single assignment (SSA) form [15, 55, 27] appealing in the first place. Moreover, it saves the overhead of premature live-range splitting. Lastly, the cost of a potential repair can be integrated into a graph-coloring based register allocator, e.g., the iterated register coalescer (IRC) [51], through the introduction of *antipathies* (*affinities* of negative weight) that can be handled with minor changes in the implementation.

We also present how repairing approach can be applied to a linear scan [89, 95, 103, 76, 105] or to its SSA form based improvement a *tree-scan* . Those allocators use an approach that *decouples* the spilling to the coalescing phase [3]. SSA form enables the design of decoupled register allocation schemes very naturally as it provides to liveness and interferences nice properties [15, 27, 58] that

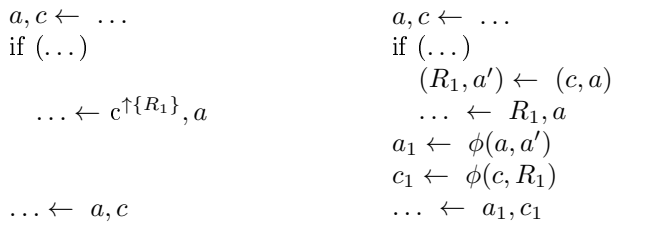
guarantee the register pressure of the program to *equal* its register demand. Thus, in SSA-based register allocation, the spilling phase simply decreases the register pressure to the number of available registers K. Then, a tree-scan that traverses the dominance tree can produce a register assignment in linear time *without* introducing further spilling [18].

Our repairing approach does *not* address register bank irregularities, such as aliasing [99] or register pairing; we will present in Chapter 6 a method that handles those constraints in the context of a decoupled register allocation scheme that tries to avoid inserting copies at every program point as in the elementary form [85] but still relies on live-range splitting. Handling aliasing constraints without excessive and preliminary live-range splitting remains an open problem, which we do not attempt to address here. Repairing is concerned with constraints that are local to individual instructions.

This chapter makes the following contributions:

- In Section 5.1, we extend the standard coalescing problem with *antipathies* between variables to express the fact that a variable should not be coalesced to another variable or register. Unlike *affinities* that have positive weight to express the potential gain of coalescing the corresponding variables (removal of a copy), antipathies can be seen as negative affinities that express the potential cost of assigning them to the same register (introduction of copies). While *coalescing* aims at merging as many affinity related variables as possible, *alienation* aims at making interfere as many antipathy-related variables as possible. Using affinities and antipathies, hints for register constraints can be modeled without significantly blowing up the size of the interference graph (IG). We first show how antipathies can be modeled by interferences and (positive weight) affinities and can thus be incorporated into existing allocators by only modifying the IG construction phase. We then present an elegant extension to the IRC that directly handles antipathies, so avoiding the modification and size increase of the IG.
- In Sections 5.2 and 5.3, we show how repairing can be used in scan like allocators and describe a tree-scan. We show how to minimize the number of repairing copies without the use of any graph-based coalescing. To this end, we present several *biased* heuristics for coloring.
- Related work is reported in Section 5.4.
- Section 5.5 presents an extensive experimental evaluation that shows the effectiveness of our techniques on the integer part of the Spec `CINT2000` benchmark suite. The use of repairing technique produces IGs that have 26% less nodes (33% less edges) compared to the state-of-the art solution with preliminary live-range splitting. Using antipathies and afterward repairing does not change the quality in terms of run-time of the compiled program. The base line tree-scan algorithm produces code of the same quality as the IRC while showing an allocation time speedup of 8.81x. Activating biasing techniques outperforms the run time performance of the best IRC configuration while the allocation time speedup compared to IRC is still 6.43x. These good results also carry against the recent preference guided scan allocator from Braun et al. [22] where our algorithm is 4.72x faster for a similar run-time quality.

Finally, Section 5.6 concludes the chapter.



(a) Initial code (b) Code with live-range splitting

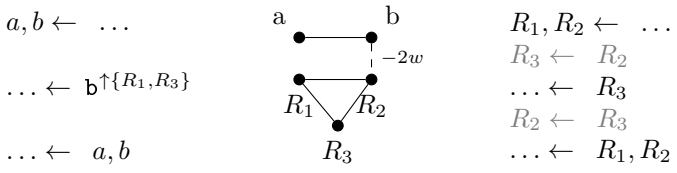
Figure 5.1: Effects of live-range splitting. $(R_1, a') \leftarrow \ (c, a)$ stands for parallel copies where $R_1 \leftarrow \ c$ and $a' \leftarrow \ a$ are done in parallel. $c^{\uparrow\{R_1\}}$ indicates that operand c has to be in the related subset of registers, here $\{R_1\}$.

5.1 Graph Coloring with Repairing

Many compilers use an IG to guide register allocation (see Chapter 2 for more details). In principle, any graph coloring register allocator can be modified to handle register constraints through the introduction of pre-colored vertices [51]. Any variable that should be assigned to register R is initially merged with the pre-colored vertex R. Any variable which assignment is constrained to a register subset is made interfering with any register not part of the subset. The fulfillment of operand constraints might require splitting live-ranges by inserting copies. Indeed, a given variable may appear in two operands which constraints are incompatible. Also, constraining at least two vertices to be assigned to some given colors can make a graph initially colorable not colorable anymore, thus causing additional spilling. To limit the lifetime of constrained variables, the allocator usually splits, prior to coloring, live-ranges by inserting copies around [76], or at least (for SSA code) just before [55] each constrained instruction. In general, this can reduce the amount of additional spilling, and for SSA form programs it guarantees the register pressure to equal its register demand. As illustrated in Figure 5.1 live-range splitting can be done through the use of *parallel copies* that correspond to set of copies to be executed simultaneously.

5.1.1 Model and restrictions

Register constraints have different variants. Commonly several registers are charged with a special meaning throughout the program such as the stack or frame pointer. Hence, they are usually not subject to register allocation and excluded from the set of available registers. In this chapter, we consider a more local constraint were an instruction dictates that *an operand* has to be in a specific (subset of) register(s), e.g., a register class. Such constraints often occur in calling conventions of the ABI. Each argument to a function call has to be put into a dedicated register. Figure 5.2 illustrates how this constraint is modeled using antipathies. In Figure 5.2a, $b^{\uparrow\{R_1,R_3\}}$ states that the corresponding operand that uses b is constrained to be in the register subset $\{R_1, R_3\}$. We say [70, 91] that operand b is *pinned* to $\{R_1, R_3\}$. If, for some reason, b is assigned to R_2 then some shuffle code has to be inserted prior to (and after) the



(a) Initial code (b) Interference graph (c) Allocated code

Figure 5.2: If a and b are respectively allocated to R_1 and R_2 some repairing code (in gray) is inserted. An antipathy (dashed lines) of weight $-2w$ is used to model this cost in the interference graph.

pinned operation to copy b to (and respectively from) either R_1 or R_3, as shown in Figure 5.2c.

As shown in Figure 5.2b, an *antipathy* of weight $-2w$ between b and R_2, where w stands for the weight of a copy instruction, indicates that assigning b to R_2 will require at least two repairing copies around the pinned operation. For coalescing, *affinities* that express the *benefit* of assigning two variables together are represented in the IG using dashed lines of positive weight. Similarly, antipathies that express the *repairing cost* of assigning two variables together are also represented using dashed lines but of negative weight. We say that an *affinity is satisfied* by a coloring if the two corresponding are given the same color (coalesced). Similarly, we say that an *antipathy is satisfied* by a coloring if the two corresponding nodes are given two different colors (made interfering).

5.1.2 Strategies

We have integrated support for antipathies into the IRC, a graph-coloring based register allocator by George and Appel [51]. The original IRC implementation performs spilling and coalescing together (see Figure 2.4); as our compiler uses a decoupled approach, and a different spilling algorithm, we focus on the coalescing part. In other words, the *potential spill*, *select*, and *actual spill* can be ignored at this stage of the discussion.

The IRC algorithm iteratively transforms the graph by merging (*coalescing*) some affinity related nodes. It also removes nodes of low degree (i.e., of degree smaller than the number of available registers) that are not affinity related (*simplification*). Every simplified node is pushed onto a stack. This is the coalescing-simplification phase. When all nodes are simplified it pops nodes from the stack and assigns a color. This is the color phase. The coalescing process uses an ordered (by decreasing weight) work-list of affinities (`worklistMoves` in [51]). For each affinity the algorithm checks by simple rules (namely Brigg's & George's) if both ends of the affinity can be coalesced conservatively (regarding the graph colorability). If it can, it merges the nodes, otherwise, put the affinity in some other lists. Optimistically, a judicious choice of color still has the possibility to satisfy some or all of the non-coalesced affinities when it is later popped from the stack and assigned a color; this is called *biased coloring*, as discussed by Briggs et al. [26].

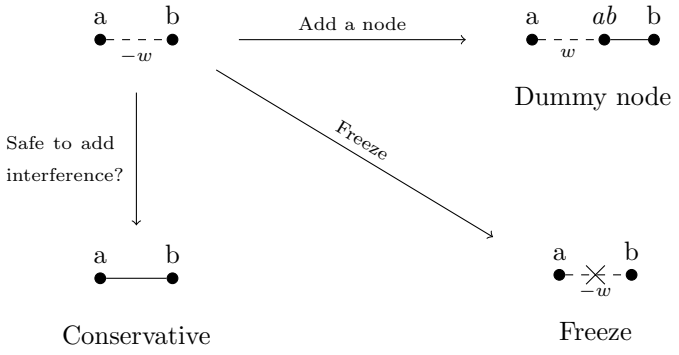


Figure 5.3: Strategies to deal with antipathies.

Our goal is to handle antipathies within this algorithm. As the notation (dashed lines) suggests, one may want to consider antipathies as affinities of negative weight. This allows the following formalism:

Definition 1 (Optimal coloring)**.** Consider an IG $G = (V, E)$ and a weighted function that associates to each couple $(x, y) \in V \times V$ a number $w(x, y)$ (positive for affinities, negative for antipathies, null for others). A k-coloring associates to each vertex $x \in V$ a integer (color) $\mathsf{col}(x) \in [1, \ldots, k]$ such that for each $(x, y) \in E$, $\mathsf{col}(x) \neq \mathsf{col}(y)$. The weight $w(\mathsf{col})$ of a k-coloring col is the sum over each $(x, y) \in V \times V$ such that $\mathsf{col}(x) = \mathsf{col}(y)$ of $w(x, y)$. A k-coloring is said to be optimal if there is no other k-coloring with bigger weight.

This formalism imposes to have at most one affinity per pair of nodes. Thus affinities and antipathies have to be summed during the build phase of the IG. This is however always a good idea to merge affinities and antipathies between nodes as coloring algorithms that aim at maximizing the overall weight are heuristics. Notice also that this formalism allows an asymmetry for the function w. In theory one can choose to set $w(x, y) = w(y, x) = \omega$ for each affinity of weight ω between x and y or set (for example) $w(x, y) = \omega$ and $w(y, x) = 0$. One should just be coherent in his choice.

Using the IRC described above, we propose three different strategies to address our generalized optimization problem.

5.1.2.1 Freeze

Representing antipathies by affinities of negative weight, and letting the IRC cope with it is definitely a bad idea: Even if the weight of an affinity is negative, it will try to satisfy it, in other words merge the two corresponding nodes. Given a graph with affinities of negative weight, the simplest solution to avoid this behavior is to ignore them during the simplification-coalescing phase. This is done by initially *freezing* all negative affinities, i.e., by putting them in the `frozenMoves` work-list of [51]. The biased coloring approach of the color phase is modified to take the antipathies into account.

5.1.2.2 Dummy Nodes

The second technique consists in transforming a graph with antipathies into an equivalent graph with only (positive) affinities. Every antipathy (x, y) of weight $-w$ is replaced by a sequence of an interference edge (x, xy), with a new vertex xy called a *dummy node*, which does not correspond to an actual variable in the program, and a (positive) affinity (xy, y) of weight w. Any existing graph coloring algorithm can directly assign color for the resulting graph. Any optimal coloring of this new graph will provide an optimal coloring of the original graph.

Definition 2 (Graph with dummy nodes)**.** Consider an IG $G = (V, E)$ and a weighted function w. The corresponding graph with dummy nodes $G' = (V', E')$ and its corresponding weighted function w' is defined and built as follow: (1) for each $x \in V$ create a vertex x in V'; (2) for each $(x, y) \in E$, create an edge in E'; (3) for each $(x, y) \in V \times V$ such that $w(x, y) > 0$ set $w'(x, y) = w(x, y)$; (4) for each couple $(x, y) \in V \times V$ such that $w(x, y) < 0$, create a node xy in V', an edge (x, xy) in E', and set $w'(xy, y) = -w(x, y)$; (5) for all remaining couples $(x, y) \in V' \times V'$ set $w'(x, y) = 0$.

Theorem 1 (Equivalence with Dummy Nodes)**.** *Let $k \geq 2$. Consider an IG $G = (V, E)$ and a weighted function w. Consider its corresponding graph with dummy nodes $G' = (V \bigcup D, E')$, with w' its weighted function, and D the dummy nodes.*
(1) if there exists a k-coloring for G, then there also exists a k-coloring for G';
(2) let col be an optimal k-coloring for G', then the restriction of col to V is an optimal k-coloring for G.

Proof. (1) Consider a k-coloring of G with $k \geq 2$. For each dummy node xy of D interfering with x, set $\mathsf{col}(xy)$ to any color different than $\mathsf{col}(x)$. Such a color exists as $k \geq 2$. This provides a k-coloring for G'.
If we force $\mathsf{col}(xy)$ to be equal to $\mathsf{col}(y)$ when possible, i.e., when $\mathsf{col}(y) \neq \mathsf{col}(x)$, then we have

$$\begin{aligned}
w(\mathsf{col}) &= \sum_{\substack{(x,y)\in V\times V\\ w(x,y)>0\\ \mathsf{col}(x)=\mathsf{col}(y)}} w(x,y) + \sum_{\substack{(x,y)\in V\times V\\ w(x,y)<0}} w(x,y) - \sum_{\substack{(x,y)\in V\times V,\\ w(x,y)<0,\\ \mathsf{col}(x)\neq\mathsf{col}(y)}} w(x,y) \\
&= \sum_{\substack{(x,y)\in V\times V\\ w(x,y)>0\\ \mathsf{col}(x)=\mathsf{col}(y)}} w'(x,y) + \sum_{\substack{(x,y)\in V\times V\\ w(x,y)<0}} w(x,y) + \sum_{\substack{(x,y)\in V\times V,\\ w(x,y)<0,\\ \mathsf{col}(xy)=\mathsf{col}(y)}} w'(xy,y) \\
&= \sum_{\substack{(x,y)\in V\times V\\ \mathsf{col}(x)=\mathsf{col}(y)}} w'(x,y) + \sum_{\substack{(x,y)\in V\times V\\ w(x,y)<0}} w(x,y) + \sum_{\substack{(xy,y)\in D\times V,\\ \mathsf{col}(xy)=\mathsf{col}(y)}} w'(xy,y)
\end{aligned}$$

In other words, by letting

$$W^- = \sum_{\substack{(x,y)\in V\times V\\ w(x,y)<0}} w(x,y)$$

we get

$$w(\mathsf{col}) = w'(\mathsf{col}) + W^- \tag{5.1}$$

(2) Consider an optimal k-coloring col of G'. First, the restriction of col to V provides a k-coloring of G. Indeed, given $(x, y) \in E$, by step (2) in the construction of G', $(x, y) \in E'$, so $\mathsf{col}(x) \neq \mathsf{col}(y)$.
Now, let us prove that for each $xy \in D$, we have $\mathsf{col}(xy) = \mathsf{col}(y)$ if and only if $\mathsf{col}(x) \neq \mathsf{col}(y)$. Indeed (by contraposition), if $\mathsf{col}(x) = \mathsf{col}(y)$, as $\mathsf{col}(xy) \neq \mathsf{col}(x)$ (xy interferes with x), this implies $\mathsf{col}(xy) \neq \mathsf{col}(y)$. Reciprocally, if $\mathsf{col}(x) \neq \mathsf{col}(y)$, $\mathsf{col}(xy)$ can be set to $\mathsf{col}(y)$ which satisfies the affinity between xy and y, and then provides a strictly better solution than if by absurd $\mathsf{col}(xy) \neq \mathsf{col}(y)$.
As equation 5.1 holds, this proves that if $w'(\mathsf{col})$ is maximal for G', $w(\mathsf{col})$ is maximal for G. □

5.1.2.3 Conservative Alienation

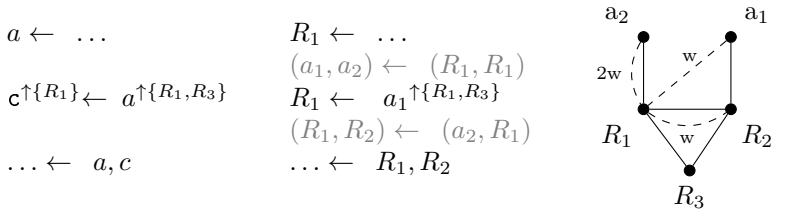
The basic idea of this third technique is to *conservatively* replace an antipathy (x, y) with an interference edge, when doing so does not affect the colorability of the IG. Recall that the work-list of affinities is sorted using their weight. Our first modification consists in putting both antipathies and affinities in this work-list and considering the absolute value of the weights in the way they are sorted. Whenever a (positive) affinity is popped from the work-list, the code is unchanged: The conservative coalescing tests [19] are performed and if successful the two corresponding nodes are merged. When an antipathy is popped from the work-list, the test consists in checking instead if the antipathy can by conservatively (regarding the graph colorability) replaced by an interference. If the test is successful the interference is actually added, the degrees of the corresponding nodes updated, and their position in the many work-lists handled by IRC updated also. The rule can be stated as follow:

Definition 3 (Conservative Alienation)**.** *let k be the number of available registers. Let (u, v) be an antipathy; (u, v) can be replaced with an interference edge if, u (or v) has at most $k - 2$ neighbors of high degree i.e., of degree at least k.*

This rule is conservative regarding the greedy-k-colorability [17] of the graph. A graph is said to be greedy-k-colorable if it can be reduced to an empty graph by successively eliminating (simplification process mentioned above) low degree nodes (degree less than k).

Theorem 2 (Preservation of greedy-k-colorability)**.** *The* conservative interfering rule *preserves the greedy-k-colorability. In other words, consider a greedy-k-colorable IG $G = (V, E)$. Consider two nodes u and v in this graph such that u has at most $k - 2$ high degree neighbors. Then the graph $G' = (V, E \bigcup \{(u, v)\})$ is greedy-k-colorable.*

Proof. Clearly a sub-graph of a greedy-k-colorable graph is also greedy-k-colorable: Any elimination order that fully reduces a graph can also be used to fully reduce any sub-graph, as nodes on the sub-graph have a lower degree than in the initial graph. Suppose u has at most $k - 2$ high degree neighbors. Adding an interference between u and v does not change the degree of nodes other than u and v. All originally low degree neighbors of u (excluding v) can still be eliminated. Remains at most $k - 1$ neighbors (including v), so u itself can then be eliminated. The obtained graph is a sub-graph of the initial IG. This proves that the introduction of such an interference does not change the greedy-k-colorability of the graph. □



(a) Initial code (b) Allocated code (c) Interference graph

Figure 5.4: a, c have been assigned R_1, R_2. Some parallel copies are introduced to repair the inconsistency. The new local variables a_1 and a_2 have to be allocated. The corresponding interference graph.

5.1.3 Repairing Code

When coloring is over, repairing code has to be inserted for each actual antipathies that have not been satisfied, i.e., whenever two antipathy-related nodes have been assigned the same register. Repairing can be understood as an allocation problem restricted to a very small region around the pinned operation. Consider the example of Figure 5.4a. Suppose that, despite the affinity of c with R_1 and the antipathy of a with R_1 (as a is live-through), c and a have been assigned respectively R_2 and R_1. To repair the inconsistencies, every variable live-in of the pinned operation (a here) is copied to a new local variable (a_1 here). Any use in that operation is replaced by the corresponding freshly created variable; hence the use of a is replaced by a use of a_1. If, as a, a live-in variable is both used in the operation and live-out of the operation then it is duplicated, i.e., copied to another new local variable (here a_2): This duplication will be the one that will traverse the pinned operation. Note that a_1 and a_2 are not made interfering here. Every defined variable (here c) is also replaced by a new local variable; in our example, as for any variable whose constrained subset is a singleton, we directly replaced this new local variable by the only possible register it has to be allocated to, i.e., R_1. Now, for every variable live-out of the pinned operation (here c allocated to R_1, and a carried by a_2) a copy back from the corresponding new local variable is inserted just after the pinned operation. In our example, R_1 (that carries the definition of c) is copied to c (allocated to R_2), and a_2 is copied back to a (allocated to R_1). This leads to the code of Figure 5.4b where assigned variables have been replaced by registers, and where the freshly created local variables remain to be allocated. We end up with a classical allocation problem where copies are affinities to be satisfied and interferences link variables that cannot share the same register. The corresponding IG is represented in Figure 5.4c. Affinities between interfering nodes that could obviously not be satisfied have been represented for completeness. a_1 and a_2 respectively assigned R_1 and R_2 would lead to a final code with a copy $R_2 \leftarrow \ R_1$ before the operation and a swap of R_1 and R_2 after. In practice, the allocation problem being very local, the IG is not actually built. A greedy ad-hoc heuristic, such as the one developed in Section 5.2.2, is designed instead.

After repairing, like for every approaches that use the parallel copies [3, 85, 22], we sequentialize them using `swap`, `move` and `xor` operations [55].

5.2 Tree-Scan

In the general graph-coloring setting, the minimum number of registers required to color the graph might very well exceed the maximum register pressure of the program. Recent results on SSA-based register allocation show [15, 27, 58] that if the program is in SSA form, its register demand equals its maximum register pressure. This allows for decoupling spilling and register assignment: once the maximum register pressure in the program is lowered to the number of available registers, a *scan* algorithm manages to assign registers without causing further spills.

To this end, the tree-scan algorithm traverses the dominator tree in pre-order, while processing the definitions and uses of variables in a manner similar to linear scan register allocation [89]. However, in contrast to the original linear scan algorithm, tree-scan does not over-approximate the live-ranges of variables by intervals but uses precise liveness information.

Spilling techniques [21, 55] for SSA programs are not in the scope of this chapter; we assume that spilling has already been performed and the register pressure is nowhere larger than the number of registers.

5.2.1 The Basic Algorithm

The control flow graph (CFG) is processed in reverse post-order (RPO) (in general any dominance-preserving order works). Each basic block is traversed from top to bottom. A bit set of occupied registers is maintained. At the entry of a basic block this should be set to the registers used by variables that are live-in. However, SSA form allows to avoid the cost of pre-computing liveness sets in favor of the fast liveness check developed by Boissinot et al. [14]. This technique provides a query system to answer questions such as "*is variable v live at location q*" but does not compute any set of live variables as the standard data-flow analysis technique would do. The reason why liveness sets can be avoided under SSA is that, a variable live-in of a block is also live-out of its already processed immediately dominating block: the scan algorithm can reuse the occupancy set of the end of the immediate dominator block, tests which of those variables are live-in, and release unused registers accordingly. During the scan of a basic block, whenever a definition of a variable is encountered, it is assigned the next free register. Whenever a *death point* of a variable is encountered (the variable is no more live after this program point), the corresponding register is released. For this last task, fast liveness check can also be used.

Main loop (Algorithms 2 and 3) The pseudo code of the main loop is given in Algorithm 2 and the details of the processing of a single operation is given in Algorithm 3. As register assignment is classically assimilated to graph coloring, the term colors will be used heavily in place of registers. In these algorithms, code in gray corresponds to repairing features explained in Section 5.2.2. The remaining code shows the basic algorithm that can be directly implemented as it if no repairing is involved or if repairing is done as a separate phase afterwards. In this case, the helper function ChooseColor called for each variables definition simplifies to providing the first available register.

The first task TreeScan does when processing a basic block *block* is to initialize its set of live-in variables *block*.allocatedVariables: checking if variable v

is live-in of basic block *block* is done through v.islivein(*block*). It is then updated, for each operation, *op*, by ProcessOperation along with the corresponding (not reported in the pseudo-code) bit-sets of occupied and available registers. To avoid checking the set of all allocated variables, dead variables, i.e., variables not live-out of the current operation (tested through u.isliveout(*op*)), are extracted from the set of variables used by the operation (*op*.arguments). At this point ϕ-functions need a special treatment as explained below.

As every definition dominates all its uses, once an operation have been fully processed, all its operands can be replaced by the assigned registers. This is done through the call of function AssignOperandsColor which implementation subtleties related to ϕ-functions arguments are explained at the end of the next paragraph.

Algorithm 2 Tree-scan main loop. Code in gray represents repairing code.

```
1: procedure TreeScan(Region region)
2:     for block in region.blocks using reverse post-order do
3:         // Initialize set of occupied registers
4:         block.allocatedVariables ← if block.isEntry then ∅
                                      else block.idom.allocatedVariables
5:         block.allocatedVariables ← {v ∈ block.allocatedVariables / v.islivein(block)}

6:         // Forward traversal of the operations
7:         for op in block.ops do
8:             ProcessOperation(block, op)
9:             If op.next=⊥ or op.next.isLateOperation then
10:                // Last point of the block where we can insert code
11:                FixGlobalColor(block, op.next)
12:        // If the late operation changes the global color, then the outgoing edges
13:        // have to be split and FixGlobalColor called on all created blocks.
```

Special treatments for ϕ-functions Even if the instruction used to represent a ϕ-function in the intermediate representation (IR) is usually placed at the beginning of a basic block, its uses should semantically be considered as being at the end of its corresponding predecessor basic blocks, or as here, on the corresponding incoming edges. This explains why line 5 of Algorithm 3 filters out ϕ-functions: dead arguments (and in particular dead ϕ-arguments) are released when entering the basic block thanks to line 5 of Algorithm 2. Another subtlety related to ϕ-functions is that the set of ϕ-functions of a given basic block should be executed simultaneously. As an example, consider two ϕ-functions written in sequence in the IR of the program as follow: $a_1 = \phi(a_2, a_3)$; $b_1 = \phi(b_2, b_3)$. Suppose a_1 is not used *anywhere* in the program. The code should *not* be understood as the sequence (1) assign a_1; (2) release a_1; (3) assign b_1. But as (1) assign a_1 and b_1; (2) release a_1. For that reason, the ϕ-functions of a basic block should be treated all together: lines 21 and 34 of Algorithm 3 should iterate over *all* ϕ-definitions, a_1 and a_2 in our example. Lastly, as already mentioned, ϕ-function semantics also impacts the implementation of AssignOperandsColor: when reaching a ϕ-function, the arguments that flow from a back-edge, are not yet assigned. To avoid a special treatment of ϕ-functions arguments at the end of each basic blocks, a list of use operands (v.unassignedUses), is attached to each variable v. Those will be replaced by the assigned color as soon as the definition is processed and the variable allocated.

Algorithm 3 Tree-scan operation processing. Code in gray represents repairing code.

Require: The set of all ϕ-functions of a basic-block should be encapsulated inside a single operation

```
 1: procedure ProcessOperation(BasicBlock block, Operation op)
 2:     dead ← ∅
 3:     parallelCopy ← []
 4:     // φ-function arguments are considered to be on the incomming edges, not here.
 5:     if op not is φ operation then
 6:         // Check arguments constraints and release last used colors
 7:         for u ∈ op.arguments do
 8:             // If current color does not match constraints, then repair
 9:             If u.ccolor ∉ op.constraints(u) then
10:                 success ← RepairArgument(block, op, u, &parallelCopy)
11:                 If not success then
12:                     // Repairing heuristic failed. Replay all using graph coloring
13:                     GraphColoring(block, op, &parallelCopy)
14:                     goto end_of_coloring
15:             // Check whether u is last used here or not
16:             if not u.isliveout(op) then  dead ← dead ∪ {u}
17:         // Release dead variables
18:         block.allocatedVariables ← block.allocatedVariables \ dead

19:     // Assign definitions
20:     for d ∈ op.results do
21:         [d.gcolor, d.ccolor] ← ChooseColor(block, op, d)
22:         If d.ccolor = ⊥ then
23:             success ← RepairResult(block, op, d, &parallelCopy)
24:             If not success then
25:                 GraphColoring(block, op, &parallelCopy)
26:                 goto end_of_coloring
27:         block.allocatedVariables ← block.allocatedVariables ∪ {d}

28:     label end_of_coloring:
29:     // Instanciate repairing
30:     InsertParallelCopy(block, op, parallelCopy)
31:     AssignOperandsColor(op)

32:     // Release dead definitions
33:     for d ∈ op.defs if not d.isliveout(op) do
34:         block.allocatedVariables ← block.allocatedVariables \ {d}
```

5.2.2 Repairing

The goal of this section is to describe how the tree-scan can be extended to handle register constraints and inline the repairing process during the traversal.

Each variable is assigned one *global* color, called *gcolor*. This is the color that the variable has *across* basic blocks: the assignment at the *entry* and *exit* of each basic block must obey the global coloring. On the other hand, so as to fulfill some operand constraints *inside* a basic block, a variable can take, *locally* to that basic block, different colors than its global one. This follows the spirit of repairing advocated in the previous section: just as the repairing approach in graph coloring context allows to reduce the size of the IG, the repairing approach in scan context avoids the storage of each basic block boundary register assignment.

In other words, as the tree-scan progresses, any allocated variable has a *current* color (called *ccolor*) that might be different than its global color. The current color of a variable can change (i.e., be different than at the immediately dominating operation) whenever a pinned operation is encountered. Note that its global color is not necessarily restored just after a constraining operation. This is done lazily instead: if live-out of the basic block, the variable can, later be allocated back to its global color when another pinned operation is encountered, or at least just before reaching the end of the basic block.

Repairing at the end of a basic block (Algorithm 2) In Algorithm 2, the repairing code inserted before a constrained operation is handled during the call to ProcessOperation. If, when reaching the end of the basic block, the current color of a variable is different than its global color, a copy is inserted to restore it by the call to FixGlobalColor (Algorithm 4). By "end of the basic block", we mean the last point where a copy can be inserted i.e., not necessarily at its really end but possibly just before an operation such as a jump (designed as a *late operation*). The repairing code of ProcessOperation (Algorithm 3) is detailed hereafter in the corresponding paragraph.

Algorithm 4 Tree-scan fix global color process. For all variables that are not in their global color, copy them in parallel to their global color.

Require: All allocated variables at this point have a different global color.

```
1: procedure FixGlobalColor(BasicBlock block, Operation op)
2:     parallelCopy ← []
3:     for var ∈ block.allocatedVariables do
4:         if var.ccolor ≠ var.gcolor then
5:             AddToParallelCopy(&parallelCopy, var, var.gcolor)
6:     InsertParallelCopy(block, op, parallelCopy)
```

Repairing at a constrained operation (Algorithm 3) When reaching a pinned operation, a parallel copy (parallelCopy in Algorithm 3) might have to be inserted just before the operation so as to match its register constraints. Recall that the restoring to the global color is not done just after the operation but lazily instead. The proposed heuristic that processes and fulfills constrained operands one after an other can fail in finding a coloring. Graph coloring is

Algorithm 5 Tree-scan local assignment process.

```
1: procedure AssignOperandsColor(Operation op)
2:     for i ← 0 to op.operands.length() do
3:         v ← op.operands[i].var
4:         if v.ccolor ≠ ⊥ then op.operands[i].color ← v.ccolor
5:         else v.unassignedUses ← v.unassignedUses ∪ (op,i)
6:     for u ∈ op.results do
7:         for (op',i) ∈ u.unassignedUses do op'.operands[i].color ← u.gcolor
```

used as a fallback solution. Procedure GraphColoring (Algorithm 10) is detailed further in the corresponding paragraph. As the operands are processed, if repairing is required, parallelCopy and the corresponding ccolor variables attribute are updated by RepairArgument (Algorithm 7) for arguments and RepairResults (Algorithm 9) for results (both procedures are detailed further in the corresponding paragraphs). There are two situations that motivate the insertion of repairing code: (1) if a pinned argument is not already in the required register class (line 9 of Algorithm 3); (2) if the colors of a pinned result are already taken by other variables (line 22). For a variable v and an operation *op*, *op*.constraints(v) returns the register class v is restricted to on *op*. If no restrictions apply, the whole register class of v, v.regClass, is returned. If GraphColoring is called, repairing is done for all operands at once, parallelCopy and variables attributes ccolor and gcolor are set accordingly. During the processing of operands, parallelCopy is represented as a map that associates copies to variables. It is instantiated as an actual parallel copy and inserted just before the operation, only once all operands are processed through the call to InsertParallelCopy.

Selecting a color for a variable (Algorithm 6) Repairing affects the color choice in several ways. ChooseColor is called in three different contexts. First, at the definition point of some variable v (line 22 of Algorithm 3), both its global color and local one have to be set. Here the global color to choose must be different from the *global* colors used by interfering variables i.e., not in block.allocatedVariables.gcolor (that abusively represents the set $\{$ *var*.ccolor $\mid$ *var* $\in$ block.allocatedVariables and *var*.ccolor $\neq \perp$ $\}$). However, it might be that a free global color is *locally* in use at v's definition (i.e., in block.allocatedVariables.ccolor). This happens because of repairing: Another variable took that color to fulfill a certain constraint. The algorithm first checks if a color is both locally and globally available. Here, for a set *colors* Pick(*colors*) returns one of its elements if none empty and $\perp$ otherwise. Color biasing techniques as addressed by Section 5.3 can be applied at this point. If none of the allowed global colors are locally available, global and local assignment have to be different. This *temporary* state will be automatically restored later in the block thanks to the repairing process described further. The second situation where ChooseColor is called is during repairing e.g., when a live-in variable has to be recolored because of some local constraints. In that case, the current color is preferably set to its global color (already set at its definition point) if in allowedCColors. The last situation where ChooseColor is called is right after the graph coloring of the current operation. The global color is preferably set to its current color (set by graph coloring) if in allowedGColors.

Algorithm 6 Tree-scan color choice.

Require: The register pressure does not exceeded the number of registers.
Ensure: Returns *ccolor* if called by repairing, *ggcolor* if called by graph coloring, *[gcolor,ccolor]* if called by the main tree-scan loop.

```
1: function ChooseColor(BasicBlock block, Operation op, Variable var, RegisterSet allowedCColors = op.constraints(var) \ block.allocatedVariables.ccolor)
2:     AllowedGColors ← var.regClass \ block.allocatedVariables.gcolor

3:     // Returns [gcolor, ccolor] (we have reached a definition point)
4:     if var.gcolor = ⊥ and var.ccolor = ⊥ then
5:         color ← Pick(allowedCColors ∩ allowedGColors)
6:         if color ≠ ⊥ then return [color, color]
7:         else return [Pick(allowedGColors), Pick(allowedCColors)]

8:     // Returns the new ccolor (required for repairing)
9:     if var.gcolor ≠ ⊥ and var.ccolor ≠ ⊥ then
10:        if var.gcolor ∈ allowedCColors then return var.gcolor
11:        else return Pick(allowedCColors)

12:    // Returns gcolor (required by graph coloring that only sets ccolor)
13:    if var.gcolor = ⊥ and var.ccolor ≠ ⊥ then
14:        if var.ccolor ∈ allowedGColors then return var.ccolor
15:        else return Pick(allowedGColors)
```

Repairing Arguments (Algorithm 7) RepairArgument procedure is called whenever an operand is pinned to a register subclass fully occupied by some other variables. So as to release a color for the pinned operand, a variable (we say a pawn) has to be moved out from its place. As moving out a variable might require moving another variable, the procedure is recursive. forbidden, initialized to the empty set, is used to avoid endless loop. All the colors the variable var is allowed to take, are considered as candidates for receiving var (line 3). The one used by unconstrained variables are considered first as they will avoid recursion (line 6). For a given candidate, if the occupant (pawn) can move to another place (line 12) the process succeeds and the move is committed (lines 13-14). If it cannot, RepairArgument is called recursively. The current color taken by var is made available for the recursively considered pawns, but the color taken by pawn is marked forbidden so as to avoid considering it again in the recursion (line 17). If the repairing succeeds, the procedure returns true. In that case, parallelCopy contains the appropriate permutation of colors, and the current colors of all involved variables are updated accordingly. Otherwise, nothing is modified.

Note that, because of the recursion, the worst case complexity of this greedy ad-hoc heuristic is exponential in the number of pinned operands even-though a bipartite matching (with lower worst case complexity) could probably do a better job in minimizing the amount of copies. We argue that repairing is rarely required, and that the exponential behavior (only pinned operands to more than one register impact the complexity) cannot appear at least for the architectures we are aware of.

Algorithm 7 Tree-scan argument repairing process. No color is available, so we take from a "pawn" already in place, that might itself move another pawn... If success, makes the moves and recolor accordingly. If not return false.

Require: All variables live in front of the operation are in block.allocatedVariables. No color is available for var.

Ensure: Performs the repairing if possible (update parallelCopy and ccolors accordingly). Returns false otherwise.

```
1: function RepairArgument(BasicBlock block, Operation op, Variable var, ParallelCopy& parallelCopy, RegisterSet available = allColors \ block.allocatedVariables.ccolor, RegisterSet forbidden = ∅)
2:     // Try out every possible moves
3:     ccolor ← ⊥
4:     allowed ← op.constraints(var) \ forbidden
5:     while ccolor = ⊥ and allowed ≠ ∅ do
6:         // Not used in op ⇒ not constrained. So start trying not in op.uses first
7:         if allowed \ op.uses.ccolor ≠ ∅ then
8:             ccolor ← ChooseColor(block, op, var, allowed \ op.uses.ccolor)
9:         else ccolor ← ChooseColor(block, op, var, allowed)
10:        pawn ← var ∈ allocatedVariables | pawn.ccolor = ccolor
11:        // Try to move out the pawn from ccolor
12:        pawnAllowed ← op.constraints(pawn) ∩ (available ∪ {var.ccolor})
                         \ forbidden
13:        if pawnAllowed ≠ ∅ then
14:            pawnColor ← ChooseColor(block, op, pawn, pawnAllowed)
15:            AddToParallelCopy(parallelCopy, pawn, pawnColor)
16:            success ← true
17:        else
18:            success ← RepairArgument(block, op, pawn, &parallelCopy,
                                        available ∪ {var.ccolor},
                                        forbidden ∪ {pawn.ccolor})
19:        // Failed. Continue.
20:        if not success then
21:            allowed ← allowed \ {ccolor}
22:            ccolor ← ⊥
23:    // Commit if successed
24:    if ccolor ≠ ⊥ then
25:        AddToParallelCopy(parallelCopy, var, ccolor)
26:        return true
27:    return false
```

Algorithm 8 Tree-scan parallel copy update. Parallel copy structure maps a variable to a pair of colors (source → destination).

```
1: procedure AddToParallelCopy(ParallelCopy& parallelCopy, Variable var, Color color)
2:     if parallelCopy[var] = ⊥ then
3:         set: parallelCopy[var] ← var.ccolor → color
4:     else
5:         replace in parallelCopy[var]: src → dst by src → color
6:     var.ccolor ← color
```

Repairing Results (Algorithm 9) In theory repairing a result is similar to repairing an argument. However, a cascading strategy with recursion would requires a costly handling of sets of available colors depending on whether the variable to move is a last use, a definition or a live through. The proposed solution considers only the colors taken by live-through variables (designed as the pawn) as candidates for receiving var. If pawn finds an available spot (line 9), then the repairing succeeds. If not, the idea is to look for a last-use variable (designed as arg) to be swapped with pawn. To be possible, (1) as moving arg frees arg.ccolor only for the upper part of pawn's live-range (arg is a last-use), the lower part should already be free (line 14); (2) pawn should be allowed to take arg's color (line 15); (3) finally arg should be allowed to take pawn's color (line 16). If those three conditions are meet, the swap is committed (lines 18, 21), and as arg occupies only the upper part, the lower part becomes free for var that can take it without further ado (line 27).

Algorithm 9 Tree-scan result repairing process.

```
Require:
 1: function RepairResult(BasicBlock block, Operation op, Variable var, ParallelCopy&
    parallelCopy)
 2:     // Trying a move among all live-through only variables
 3:     allowed ← op.constraints(var) \ op.defs.ccolors) \ op.uses.ccolors
 4:     while allowed ≠ ∅ and ccolor = ⊥ do
 5:         ccolor ← ChooseColor(block, op, var, allowed)
 6:         pawn ← var | var.ccolor = ccolor
 7:         pawnAllowed ← op.constraints(pawn) \ op.defs.ccolors) \ op.uses.ccolors
 8:         if pawnAllowed ≠ ∅ then
 9:             // There is an available spot for pawn
10:             pawnColor ← ChooseColor(()block, op, pawn, pawnAllowed)
11:         else
12:             // pawn's color could be free (for var) by swapping pawn with a last use
13:             for arg ∈ op.uses if not arg.isliveout(op) do
14:                 if arg.ccolor ∉ block.allocatedVariables.ccolor and
15:                                     arg.ccolor ∈ op.constraints(pawn) and
16:                                     pawn.ccolor ∈ op.constraints(arg) then
17:                     pawnColor ← arg.ccolor
18:                     AddToParallelCopy(&parallelCopy, arg, pawn.ccolor)
19:                     break

20:         if pawnColor ≠ ⊥ then
21:             AddToParallelCopy(&parallelCopy, pawn, pawnColor)
22:         else
23:             allowed ← allowed \ {ccolor}
24:             ccolor ← ⊥

25:     // Commit if success
26:     if ccolor ≠ ⊥ then
27:         var.ccolor ← ccolor
28:         return true
29:     return false
```

Fallback: Graph Coloring of the Operation (Algorithm 10) The repairing process has a fall-back mechanism as soon as one of the heuristic fails to find a coloring that fulfills the constraints. These failures mostly occur when the register pressure is exceeded, which is unlikely unless the spilling phase gets it wrong, or when there is a need for duplications. As opposed to a live-range

splitting that has the effect of moving a value from a resource to another, a *duplication* is a copy that lets the source variable alive. There are cases, such as for variable a in the example of Figure 5.4c, where a duplication cannot be avoided. In our register allocation scheme, such patterns are detected by the spilling phase and required duplications are inserted prior to the coloring/coalescing.

The fall-back mechanism, based on a graph coloring, corresponds to the repairing technique described in Section 5.1.3. First, every live-through variables are duplicated (lines 10-16). Then the IG is built. Every live-in variable should interfere with one another but for a variable with its duplicate (line 18); every variable live at the definition point should interfere with one another (line 19). Next operand constraints are expressed through interferences to non allowed colors (line 21). Affinity setting presents two subtle differences with the description of Section 5.1.3. First, as the tree-scan restores the global color lazily instead of right after the pinned operation, the affinity of a live-through variable is 1 with its current color (line 23) plus 0.5 with its global color (lines 25). Second, as the global color of definitions are not set yet, antipathies with the global color of interfering variables are added (line 26). Once a coloring has been found, duplicated variables that have been assigned the same color than their respective parent can be deleted (lines 29-30). If not, the parallel copy could contain twice the same copy, which should be detected when sequentialized.

Our register allocation scheme is fully decoupled, meaning that no spilling is required during coloring/coalescing. However, a non fully decoupled approach using an optimistic *lightweight* spilling phase could be considered. In that case, Algorithm 10 should be able to perform spilling. `load`s and `store`s for some live-through variables would be inserted around the current operation. So one iteration of the IRC would do the job.

5.3 Biased Coloring

The goal of coalescing/alienation is to remove as many copies as possible. Some are already present in the original code, some come from the use of SSA form (through the form of ϕ-functions), and the largest source of copies come from the accommodation of register constraints (through preliminary live-range splitting or repairing). Coalescing is a hard problem (it is already NP-complete for SSA programs without register constraints [17]) and efficient coalescing algorithms are too slow (see Section 5.5) in a context of just-in-time (JIT) compilation.

The goal of this section is to present several heuristics to *bias* the color choice of the tree-scan algorithm to give `move`-related variables the same color in the first place. As our experimental evaluation shows, these heuristics suffice to waive the coalescing pass completely. Hereafter, we quickly review the adoption of Mössenböck and Wimmer's register hints [105] for tree-scan and then present new biasing approaches.

Register hints This technique can be considered as a copy propagation during the scan process. When assigning a color to the result of a `move` or parallel copy, if the color of the argument is available, the algorithm takes it. We also apply this technique for ϕ-functions results. In a ϕ-function, we have to chose among multiple source variables: One for each incoming edge. We select the color with the highest execution frequency (either determined by static analysis or profile information) over all already allocated sources.

Algorithm 10 Tree-scan fall back repairing process.

Ensure: op is colored with respect to coloring constraints (current repairing is discarded)
Ensure: availableColors and block.allocatedVariables are updated

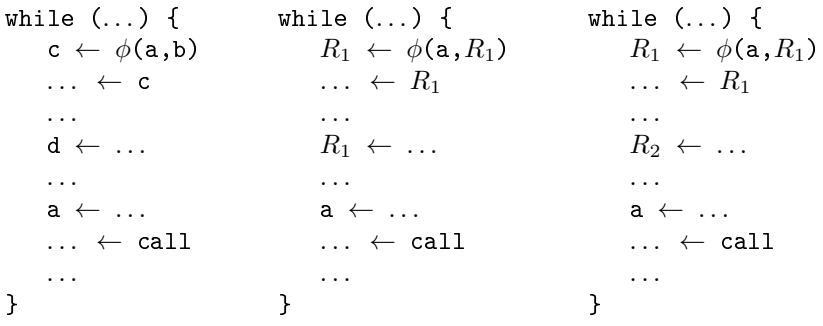
```
1: procedure GraphColoring(BasicBlock block, Operation op, ParallelCopy& parallelCopy)
2:     // Backtrack failed repairing
3:     for var: src → dst in parallelCopy do var.ccolor ← src
4:     parallelCopy ← []
5:     block.allocatedVariable ← block.allocatedVariables \ op.defs
6:     for var ∈ op.defs do  (var.gcolor, var.ccolor) ← (⊥, ⊥)

7:     // Build live-sets
8:     lastUses ← {var ∈ op.uses | not var.isliveout(op)}
9:     liveThrough ← block.allocatedVariables \ lastUses

10:    // Duplicate variables that are both used and live-through
11:    duplicates ← []
12:    for i in op.arguments.indices if op.arguments[i].var ∈ liveThrough do
13:        dup ← Duplicate(op.arguments[i].var)
14:        duplicates[var] ← duplicates[var] ∪ {dup}
15:        op.arguments[i].var ← dup
16:        lastUses ← lastUses ∪ {dup}
17:        dup.ccolor ← op.arguments[i].var.ccolor

18:    // Build the interference graph and do graph coloring potentially with local spill
19:    interferenceGraph.addCliqueButForDuplicates(lastUses ∪ liveThrough)
20:    interferenceGraph.addClique(defs ∪ liveThrough)
21:    for var ∈ op.operands do
22:        interferenceGraph.addInterferences({var}, allColors \ op.constraints(var))
23:    for var ∈ lastUses ∪ liveThrough do
24:        interferenceGraph.addAffinity(var, var.ccolor, 1)
25:    for var ∈ liveThrough do
26:        interferenceGraph.addAffinity(var, var.gcolor, 0.5)
27:    interferenceGraph.addAntipathies(op.defs, block.allocatedVariables.gcolor, 0.5)
28:    coloring ← interferenceGraph.color(op)

29:    // Remove useless duplicates and apply the coloring result
30:    for var ∈ liveThrough do
31:        for dup ∈ duplicates[var] if coloring[var] = coloring[dup] do
32:            Delete(coloring[dup])
33:            Delete(dup)
34:    for var: color in coloring if var.ccolor ≠ color do
35:        var.ccolor ← color
36:        if var ∉ defs then  AddToParallelCopy(&parallelCopy, var, color)
37:        else block.allocatedVariables ← block.allocatedVariables ∪ {var}

38:    // Set greedily a global color to definitions
39:    for d in op.defs do d.gcolor ← ChooseColor(block, op, d)
```

Aggressive pre-coalescing An aggressive coalescing merges as many copy and ϕ-function related variables as possible. It is easier than conservative coalescing as colorability of the resulting graph is not a concern. In particular there exists very fast and efficient algorithms that exploit SSA properties and do not even require the built of an IG (e.g., Boissinot et al. [13] and Budimlić et al. [30]) . Instead of actually merging variables, our aggressive pre-coalescing phase puts as many copy and ϕ-function related variables into interference-free sets (called *equivalence classes* by Sreedhar in [100]). Classes are then used during the tree-scan to bias the coloring of a variable to the "color" of the class it belongs to. The "color" of a class (initially undefined) corresponds to the global color of its last assigned variable. In other words, when assigning a color to a variable, the tree-scan checks if the color of the class is available, if so, it takes it. If not, it picks a different color (based on the other heuristics presented here) and updates the class' color.

Caller-saved registers This technique tries to put variables that are live across a call site into registers that are saved by the *callee*. Thus, it tries to avoid *caller-saved* registers for these variables. The fast liveness check method used by the original tree-scan algorithm is not very helpful, as the question that arises at the definition point of the variable is to know whether that variable is live across a call: In that case every call site dominated by the variable's definition should be tested. Instead, when using the caller-saved heuristics, we resort to a classic liveness analysis. If aggressive pre-coalescing is used as well, the across-a-call information is also propagated to the equivalence classes.

Round robin assignment The usual choice for a fresh register is to take the first available color, usually in the order of the bit set that tracks the registers in use. However, this paradigm usually leads to an unequal distribution of the colors used. Freed registers are immediately reused by the variable defined next. Hence, some registers are more frequently used than other ones. This has two negative effects. First this usually decreases the chance that a `move`-related variable can reside in the same register. Second, the allocated code contains more anti-dependences, making the job of a post-pass scheduler much harder. A round-robin strategy that affects registers in a cyclic manner aims at making a more balanced assignment. Consider the example in Figure 5.5. The result of the ϕ-function c is colored before its operand a. The variable d interferes with a. Hence, assigning the register of the class $\{a, b, c\}$ to d is bad because a cannot get it anymore. With the classic allocation strategy this might easily be the case. However, using round robin, the register of c will only be reused after K definitions, where K is the number of available registers. This increases the chances that c's register is available for a. Round robin assignment also has a positive effect on post-allocation scheduling because it decreases the locality of false dependencies. Thus, a post-allocation scheduler might have more freedom to reorder the instructions while keeping the register allocation.

Move related To further increase the chance for `move`-related variables to get "their" color (the one of their equivalence class),register file is divided into two parts (of equal size in our case but could be tuned): The first part is reserved for `move`-related variables and is only used by non-`move`-related variables if registers



(a) Initial code (b) Classical color choice (c) Round robin

Figure 5.5: Benefits of round robin on the color choice. Classical color choice reuses c's color for d and blocks the usage of that color for a. Round robin increases the chances that c's color will be available at a's definition.

of the second part are exhausted. Inside the `move`-related part, round-robin strategy is used to assign registers.

Figure 5.6 summarizes all presented bias techniques. It shows the different allocation results for each technique on an example.

5.4 Related Work

Graph coloring and register constraints Chaitin et al. [33] showed that every graph is the IG of a particular program, hence proving by reduction to k-Colorability the NP-completeness of register allocation. In this situation, there was no interest in properties of the graph structure. Thus, register constraints were represented as interferences.

More recently, it was shown that the IGs of SSA-form programs are chordal, which allows for coloring in polynomial time [15, 27, 58]. However, checking the k-colorability of a chordal graph with at least two pre-colored nodes is not polynomial anymore. Thus, early SSA-based allocators [58] used premature live-range splitting in front of constrained instructions as well. Moreover, Odaira et al. [79] show that live-range splitting implies an overhead of 20% on average in the compile time of IRC.

Scan approaches The idea of linear scan register allocation goes back to Traub et al. [103] and Poletto and Sarkar [89]. Allocation is done with a linear scan over the assembly code. Poletto and Sarkar do not take control flow into account and over-approximate the live-range of a variable by an interval on the linearized assembly code. Thus, variables might occupy a register where they are not live and might provoke unnecessary spill code. This method is simple and fast, but gives worse results than standard graph-coloring approaches. Traub et al. perform liveness analysis before and allow for holes in the intervals to avoid the over-approximation of live-ranges.

Mössenböck and Pfeiffer [76] proposed a modification of the original linear

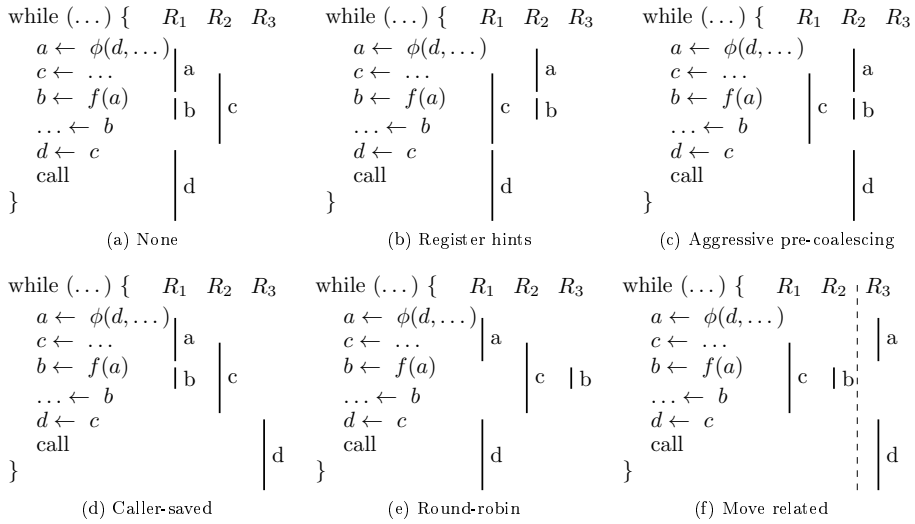


Figure 5.6: Different bias coloring strategies during tree-scan. For each technique, the left part represents the source code; the right part shows the allocation of the variables with their live-range in the related column. The second argument e of the ϕ-function is supposed to be already assigned to R_2. R_1 and R_2 are caller saved registers. For the aggressive pre-coalescing strategy, equivalence classes are supposed to be $\{b\}$, $\{c\}$, and $\{a, d, e\}$. For the `move`-related strategy the reserved set for `move`-related variables is supposed to be $\{R_3\}$.

scan to work on SSA. Unlike our *tree-scan,* they do not take advantage of SSA properties to allow for an optimal register assignment. Like Traub et al., their live-ranges have holes.

Mössenböck and Wimmer [105] further improved linear scan. In particular, they improved spill code placement and added on demand live-range splitting to avoid spilling in some context. In 2007, Sarkar and Barik [95] extended the linear scan. They explicitly split at basic block boundaries to avoid spilling and to handle register constraints at the cost of shuffle code. In our setting, the program is in SSA. Thus, introducing live-range splits in addition to ϕ-functions will *not* save any further spills [58]. In 2009, Rong [93] proposed the tree register allocation, which generalizes linear scan approaches. However, this algorithm needs global liveness information, in particular for the handling of pre-colored constraints. The same year, Barik in his thesis [6, Ch.6] proposed a linear scan approach that colors the basic intervals, i.e. part of the live-range that is contiguous in the linear ordering, independently. His algorithm tries to use the same color for the global interval, i.e. composed by several basic intervals, to minimize shuffle code between basic blocks. To minimize `move` cost, it builds another graph with all basic intervals and all `move` instructions, also the one expected to be inserted on edges, and uses this graph to get the preferred color of a basic interval when assigning its color. Overall, our approach is simpler as it does not require to build an additional graph for coalescing nor it requires to handle the shuffle code on the edges. Regarding coloring

constraints, Barik proposed two different approaches: (1) Choose an order of the register class and assign them separately, starting with the most constrained one. (2) Do everything at the same time with a register pressure by register class. In that context, coalescing of basic intervals composing a variable may be incompatible with already chosen color, thus creating a lot of `moves`. To circumvent this bad behavior, instead of a top to bottom approach, i.e. basic interval sorted by start date, Barik defined a bucket sorted list. With our approach, the global interval is assigned a color, thus all basic intervals have the same color a priori. Then, repairing makes the proper adjustments. The frequency of these adjustments depend on the time invest in pre passes analysis. In 2010, Wimmer and Franz [104] pointed out the interest of relying on SSA to deal with liveness in linear scan. Finally, the same year, Braun et al. [22] proposed a preference guided register allocator. Like our tree-scan, it works on SSA. But unlike our approach, it processes the program using a linear ordering of the basic blocks. This ordering is defined by a complete cover of the program by traces. Moreover, it has to insert shuffle code at join point if all predecessors have not been proceeded, using ϕ-functions. Regarding coloring, it uses a new bias technique, the preference sets, that gives the liked and disliked colors for each variables. Like us, their allocator repairs the register constraints on the fly but does not handle duplications, which must be set a priori. It splits all live variables when at least one of them does not match the instruction constraints and fixes the color for all split variables. It then solves a bipartite matching problem for all the new variables. Overall, the preference guided allocator is more complex than our approach.

Interestingly, already in 1999, Yang et al. [106] proposed a fast scan based register allocation than uses the CFG. This allocator presents some similarities with both the preference guided allocator and our tree-scan. Like the preference guided, it performs a two phases allocation. First, it computes some preference set as well as the last uses points[1] and second, it allocates the program. The allocation uses a RPO ordering of the basic blocks, like us, and splits the CFG in single entry multiple exits regions, i.e. it deals with tree like live-ranges. Like the preference guided, at the end of each region it stores the result of the allocation. If there is a mismatch between several predecessors of one region, it inserts some shuffle code. When this process failed, it reallocates the related region. Overall, again, this is more complex than our tree-scan, since we do not have to handle shuffle code between region.

Coalescings In graph-coloring register allocation, many different coalescing techniques have been developed. They fall into three categories: Aggressive, conservative, and optimistic coalescing. *Aggressive coalescing* removes as many copies as possible, regardless of the colorability of the IG [32]. While it removes many copies, it may also increase the register demand of the program which potentially causes spilling. Since we never want to trade a spill for a copy, aggressive coalescing has to be used with caution. *Conservative coalescing* uses conservative tests [26, 51, 15, 19] that ensure that the chromatic number of the graph is not increased, before a copy is coalesced. *Optimistic coalescing* uses aggressive coalescing and de-coalescing if the k-colorability was violated [80, 81]. On the other hand, *Biased coloring* tries to remove copies by giving the

[1] Since it does not use SSA, this information is not directly available

source and the target of the `move` the same color in the first place. Chow and Hennessy [34] rely on copy propagation to remove `move`s in the priority-based allocator. Briggs et al. [26] integrate biased coloring into graph-coloring allocation. Mössenböck and Wimmer [105] use "register hints" in their linear scan allocator to propagate copy information to the definition points of the variables. They gave also a technique based on register next use distances to assign caller-saved registers to local temporaries.

5.5 Experiments

The algorithms described earlier in the chapter were implemented in the back end of a production compiler developed by our industrial partner, STMicroelectronics for their commercial media processor based on the Lx architecture [42]. This static C compiler uses open source version of the SGI Pro64 compiler [49] (OPEN64) as the code generator, linear assembly optimizer (LAO) as the register allocator, and OPEN64 for post-allocation optimization and assembly code emission. LAO can be used both in a static and dynamic compilation context. While the funding of those developments was motivated by dynamic compilation constraints, the industrial partner does not provide us with access to the dynamic compilation tool-chain. The target processor is 4-issue very-long instruction word (VLIW) with 32 general-purpose registers, 8 of which are callee-saved. Compared to IA32, for example, the Lx architecture [42] has relatively few register constraints. That being said, our results show significant improvements compared to allocators that do not effectively handle register constraints; consequently, the disparity is likely to be even greater in our favor for target architectures with more constraints.

Our experiments use a decoupled register allocation approach. The spilling algorithms used is described in [55]; the purpose of the experiments is to compare coalescers.

Our experiments use a subset of the Spec `CINT2000` benchmarks compiled using `-O3` optimization level; our compiler cannot handle *eon*, which is written in C++, and *gcc*, which requires a frame pointer that our compiler does not support. To give a better idea on how the different configurations may apply to different targets, we made our experiments with three different numbers of allocatable registers: 32, 16 and 8 registers.

5.5.1 Graph Coloring and Repairing

These experiments establish the efficacy of our approach to repairing on five different coalescing configurations:

- `IRC`: The IRC algorithm without live-range splitting, but no repairing; this algorithm is not guaranteed to find a k-coloring of the IG, so it is allowed to spill, when necessary.
- `Split`: The IRC algorithm with live-range splitting, but no repairing.
- `Freeze`, `Conservative`, and `Dummy Nodes`: The IRC algorithm without live-range splitting but with repairing implemented as described in Section 5.1.2.

Figure 5.7 reports the normalized execution time of the code generated by each configuration. Figure 5.8 reports the normalized number of vertices and number of edges of the IG for each benchmark.

Finally, Figure 5.9 reports the normalized number of dynamically executed copies for each configuration. We used frequency estimate [5] to find the number of times each basic block executed. For each copy operation occurring in basic block b, we use the weight assigned to b to estimate the number of times the copy executes. These numbers are then summed to produce a per function cost, and these costs are summed to produce a per benchmark cost. This metric is architecture agnostic, as it ignores, for example, the possibility to hide the copies by scheduling them in parallel with one another or with other operations, or to schedule them in a branch delay slot.

Due to the number of configurations, these Figures depict just geometric means. Detail per benchmark are given in the appendix, tables A.1 to A.6. All numbers are normalized to `IRC` with 32 allocatable registers.

The baseline approaches are `IRC` and `Split`, denoted IRC Split in the Figures. Between those approaches, `Split` produces better quality code (Figures 5.7 and 5.9), but with a noticeable increase in the size of the IG (Figure 5.8). In its favor, `Split` is the only existing technique that can deal with register constraints in a decoupled register allocation context. Our goal is to identify a coalescer that achieves the code quality of `Split` but without increasing the size of the IG.

We compare `IRC` and `Split` against three approaches to handle antipathies:

`Dummy Nodes` is the naive approach to extend graph coloring to deal with antipathies. It represents antipathies using dummy nodes. As shown in Figures 5.7 and 5.9, this approach produces good quality code, but the dummy nodes that are added significantly increase the size of the IG. Although `Dummy Nodes` is not shown in Figure 5.8, its IGs are larger than `Split` in virtually all instances. For this reason, we do not consider `Dummy Nodes` to be a realistic approach.

`Freeze` only considers antipathies during the biased coloring phase at the end of the coloring process. As shown in Figure 5.9, the quality of code generated by `Freeze` is inferior to that offer all other graph approaches: `IRC`, `Split`, `Dummy Nodes`, or `Conservative`. In particular, it has big worse cases (more than 28 times more `moves` than the `IRC`) when using with 32 and 16 registers. This becomes better when less choice are possible for the color of each variable, i.e. using only 8 registers, its quality competes with `IRC`. However, these bad performances on the number of `moves` barely show up in runtime numbers as reported in Figure 5.7. In that case, `Freeze` is slightly worse that `Split` but still better than `IRC`. Hence, inserting `moves` to repair is cheaper than having additional spill code. In terms of IG size, `Freeze` is comparable to `IRC` (Figure 5.8); the difference in size (due to a small number of negative-weighted affinity edges) is negligible, and is not shown in Figure 5.8.

`Conservative` converts antipathies into interference edges using criteria similar to conservative coalescing. `Conservative` generates code that quality is comparable to `Split`, `Dummy Nodes`, and `Freeze` for runtime (Figure 5.7). It is one of the best regarding the dynamic number of `moves`

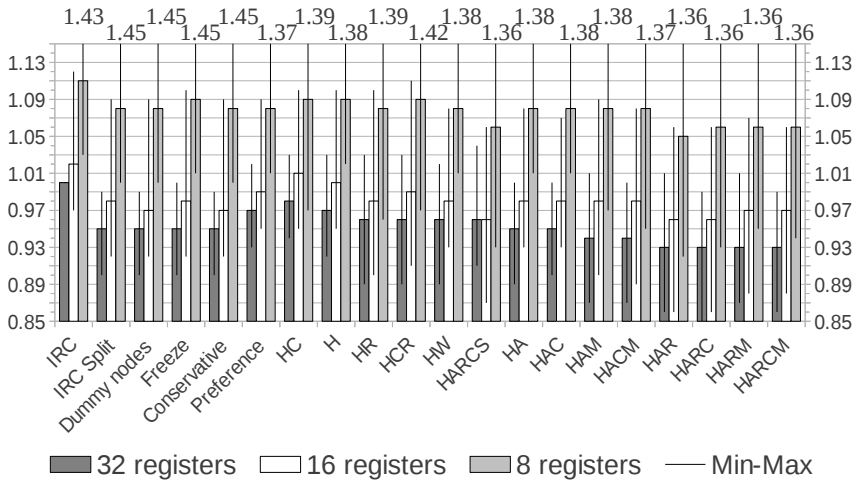


Figure 5.7: Geometric means over all benchmarks of the execution time of the generated code. Each bar represents the runtime for the given number of allocatable registers. The black lines in the middle of each bar represent the variation, i.e. minimum and maximum, over all benchmarks. All numbers are normalized to IRC with 32 registers (y=1). IRC stands for the iterated register coalescer in a decoupled fashion. IRC Split is IRC plus live-range splitting. The three next configurations are graph approaches with repairing as depicted in Section 5.1.2. Preference reports preference guided numbers. Then, letters stand for the mix of the bias coloring configurations applied to tree-scan. H: Hints; R: Round-robin; C: Caller; M: Move related; A: Aggressive; W: Web; S: Split. For tree-scan configurations, the results are sorted in increasing improvement with 32 registers. (Lower is better)

(Figure 5.9), while the size of the IG is comparable to that of `IRC`, and is not shown in Figure 5.8. Among the three antipathy-based coalescers considered here, `Conservative` is the only one to achieve the code quality of `Split` with an IG size comparable to `IRC`.

5.5.2 Tree-Scan

This section evaluates the allocation time, i.e., the compile time dedicated to register allocation, and the number of copy operations that execute dynamically when coalescing is performed by the tree-scan algorithm with different biased color assignment techniques, as discussed in Section 5.3. As this technique is intended to be applied in a JIT context, but not restricted to, this section also reports its memory footprint. To ease the comparison with existing approaches, we included the most recent, to our knowledge, scan approach, the preference guided register allocator [22] to our results.

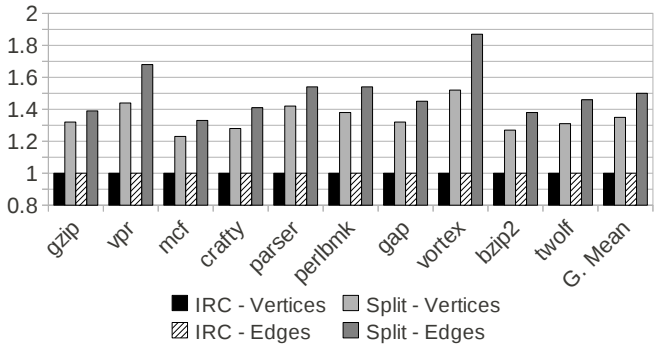


Figure 5.8: The normalized number of vertices and interference edges in the interference graph for each benchmark. For a given benchmark, the sizes of the interference graph of each function are summed. `IRC`, `Freeze`, and `Conservative` have the same interference graph sizes, while Split's interference graphs are noticeably larger. Dummy nodes is not a realistic solution, so we do not report its interference graph sizes, which would be large.

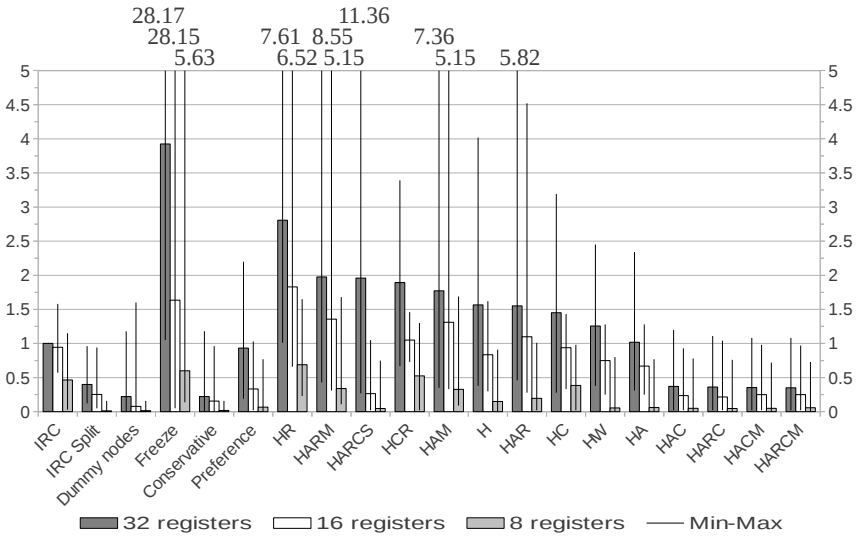


Figure 5.9: Geometric means of dynamic number of `moves`. See caption Figure 5.7 for the explanation of the configurations. The tree-scan configurations are sorted in increasing improvement with 32 registers. (Lower is better)

5.5.2.1 Allocation Time

Figure 5.10 reports the normalized compile time of the different color assignment approaches. The compile times reported include all memory allocation/deallocation, structure initialization/destruction, and liveness analysis; however, they do not include the time required to translate out of SSA, which is not part of the coloring process. For each benchmark, the runtime is the sum taken over all functions. Due to the number of configurations, only the geometric means over all benchmarks is reported. Detailed numbers are given in appendix, Tables A.7 to A.9.

As expected, the introduction of `Register Hints` to bias the color assignment process during the tree-scan incurs no measurable overhead, while `Round Robin` color assignment incurs an overhead of 10%. Pre-coalescing comes at a higher price, 12% overhead for the `Web` strategy [30] and 27% for `Aggressive` coalescing [13]; `Move Related` coalescing costs an additional 11% as we use a pre-coalescing phase to know which variables are move related. The most expensive technique, however, is `Caller`, whose overhead is 50%; this overhead is due to the data flow analysis required to compute liveness information and a traversal of the CFG to identify variables that are live across calls. Lastly, we also report the allocation time of the tree-scan with a `Split` strategy, where all live-ranges are split prior to constrained instructions [55]; the overhead of this technique is 71%.

Regarding the evolution of the different bias technique compile time with respect to the number of allocatable registers, we see that tree-scan is more or less not impacted by this number. The slight gain with 8 registers comes from the way we chose the set of allocatable colors, here they are all callee-saved. Thus, repairing on call site never occur anymore. The same observation applies to `Register Hints`. For `Round Robin`, the compile time follows this number. Nothing surprising as it traverses this set to find the next available color. The pre-coalescing techniques depend on the program structure not the number of registers. Thus, no patterns come out. Regarding `Caller`, the number of call sites is not impacted by the number of registers. Thus the numbers are almost the same. The slight gain with 8 registers comes again from choice of the allocatable registers. When choosing the color for a variable having the caller flag, the operations which restrict the possible colors will never end in an empty set, thus error case is never reached. Finally, the `Split` strategy depends on the number of live variable, which is directly linked to the number of register in decoupled register allocation approach.

On average, the baseline tree-scan runs 8.81 times faster than `IRC`, which respectively represents 4% and 26% of the whole back end compile time (17% for preference guided). In contrast, even the slowest-running variant of tree-scan has a runtime of less than 2 times than of the baseline version. Compared to the preference guided allocator, tree-scan is 4.72 times faster.

For a JIT compiler, it is clear that tree-scan runs much more efficiently than register allocation based on graph coloring. Moreover, it also beats the preference guided allocator whatever the number of allocatable registers. However, the gap is smaller with few registers and tree-scan is finally 2.96 times faster with 8 registers[2]). This is because the preference guided repairing process is faster

[2]The reported 2.82 is against the baseline tree-scan with 32 registers as all numbers use the same base to normalize

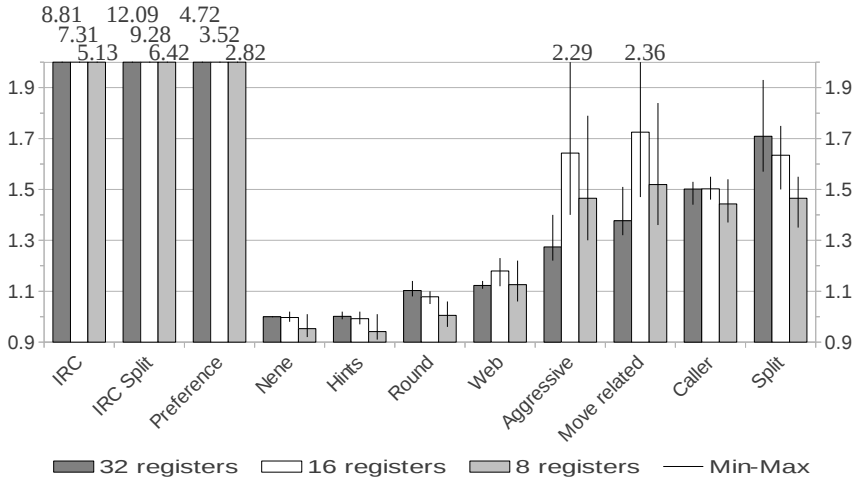


Figure 5.10: Normalized geometric means of allocation time. Numbers are normalized to tree-scan baseline (None) with 32 registers. Preference reports preference guided numbers. Configurations to the right of None are tree-scan algorithm with the related bias technique. (Lower is better)

when less variable are involved, whereas the tree-scan is more or less linear in the number of instructions. Next, we look at the quality of the code generated by the coalescers.

Note that by adding the techniques overhead, you get the allocation time of the related composed method. For instance, caller plus web have composed overhead of $(1.5-1)+(1.12-1) = 0.62$ on average. Thus, this composed method is 1.62 times slower than the baseline.

5.5.2.2 Number of Dynamically Executed Copies

Figure 5.9 reports the number of dynamically executed copy operations that result from different combinations of color assignment enhancements to the tree-scan algorithm. See Section 5.5.1 to know how these numbers are computed. We consider that tree-scan using register hints (H) is the most realistic baseline implementation for tree-scan, due to its low runtime overhead. Thus, we did not test tree-scan without any bias technique.

Let us first focus on the differences between the tree-scan configurations, using register hints (H) as baseline. As the trends are the same whatever the number of registers, we comment the numbers with 32 allocatable registers. The impact of the caller heuristic (HC) in isolation is minimal: The compiler inserts less repairing code, but fewer copies are coalesced. In many cases, two `move`-related variables cannot be coalesced because one crosses a call and the other does not; as we will see, the caller heuristic becomes more effective when combined with better coalescers.

Round-robin (HR) increases the number of dynamically executed copies by 79%. It does not have any information about future uses of variables, e.g.,, as operands of ϕs. Consequently, the likelihood of eliminating the copies that

result during SSA destruction is quite low. Thus, the potential benefit of round-robin, is possible only with a control on how it spreads the allocation over the available colors. For instance, when combining round-robin with caller (HCR) the negative impact is reduced from 79% to 20%.

The techniques that employ pre-coalescing (HW, HA) perform quite well. `Web` and aggressive strategies respectively reduce the number of dynamically executed copies by 20% and 35%. When combined with round-robin (HAR) and `move`-related (HAM), the negative impacts observed for the round-robin strategy, as described above, manifest themselves, but in a more limited fashion, as the pre-coalescer gives a better guide for assigning registers.

Combining the pre-coalescer and caller heuristic (HAC) is beneficial, because variables that are `move`-related to others that cross procedure call boundaries are biased using callee-saved registers. Compared to register hints alone (H), HAC reduces the number of dynamically executed copies by 76%. Augmenting HAC with the round-robin strategy (HARC) achieves an additional percent of improvement. Similarly, combining the `move`-related heuristic with the caller heuristic and a pre-coalescer (HACM) achieves 78% of improvement.

Lastly, we wish to establish that pre-splitting is not necessary when using repairing; the best result achieved with pre-splitting (HARCS) increasing the number of dynamically executed copies by 25% for 32 registers. This bad result comes from the way parallel copies inserted by split are handled in tree-scan. The algorithm does not have any special care for such instruction. Thus, it assigns the result in a sequential order. If the first result should not reuse the color of the first argument because of some bias information, like the caller flag[3], the used color may not be available to coalesce one of the other result. In the worse case, this error can propagate through all results of a given parallel copy, whereas it would have been limited to few variables in the repairing case that the bias information helps to avoid. However, with less registers, this problem is less likely to occur and this configuration competes with the best configurations. Nevertheless, the impact of pre-splitting on allocation time does not justify its use in a JIT compiler.

Compared to graph based and preference guided approaches, tree-scan variants perform quite well. In particular, HAC, HARC, HACM and HARCM, are better than `IRC` using live-range splitting and are close to the best achieved quality: `Conservative` repairing strategy. For 32 registers, a tree-scan using only an aggressive pre-coalescer (HA) achieves results almost as good as preference guided. Thus, it competes with preference guided whereas it is 3.72 times faster according to Figure 5.10. With 16 registers, this tree-scan configuration has to be combined with at least the caller (HAC) technique to catch up the gap in code quality against preference guided. In this case, it is still 1.64 times faster. Finally, with 8 registers, the web pre-coalescer technique is sufficient for tree-scan to beat the preference guided. In that configuration, it is 2.5 times faster.

5.5.2.3 Run Time Performance

We compare the quality of the execution of the code generated by tree-scan using the different biasing techniques. Figure 5.7 reports these results.

[3]Without other bias techniques, since split points are just a renaming of the variable, it is always possible to reuse the color of the related argument.

Due to the advantage of a fully decoupled register allocation against IRC decoupled approach, all programs compiled with tree-scan are always faster than their counterparts compiled with IRC. Although tree-scan is approximately nine times faster in allocation time than IRC.

The base tree-scan with register hints (H) generates code that is 3% faster than IRC. More surprisingly, with the additional caller heuristic (HC), code is only 2% faster than IRC even if the number of dynamic copies is less than register hints (H). This is because of the VLIW processor we use for evaluation. When the caller heuristic is *not* active, repairing often occurs at call sites. However, at call sites there are usually enough free slots in the VLIW bundles to hide the repairing code. Hence, this repairing code comes for free. If the caller heuristic *is* active, the repairing move instructions occur at different places where they are no longer easy to hide because of saturated VLIW bundles.

Round-robin (HR) gives an additional percent of improvement. This benefit comes from the additional freedom for the post scheduler. On our machine, post scheduling is very important because it places moves, stores, and loads in unused slots of near bundles.

Using a pre-coalescing approach (HW, HA), tree-scan achieves 4% of improvement. This is almost as good as IRC with splitting or repairing technique. Combining these approaches with caller heuristic (HAC), tree-scan gets an additional percent of improvements and is as good as the best graph coloring algorithms reported here. We achieve an additional percent by combining pre coalescing with round robin (HAR), having tree-scan generated code running faster than the best graph based approach.

Surprisingly, preference guided is just slightly better than tree-scan with just register hints (H), despite the fact that it has far less dynamic moves than this configuration. The reasons are twofold. Preference guided biases his color using the same metric as the one used to count the dynamic number of moves. However, this metric is based on an heuristic of frequency estimate and may not reflect the actual runtime behavior. Moreover, like for the caller heuristic, the repairing code is placed on saturated VLIW bundles, in that case, the edges.

In summary, we draw the following conclusions: Register hints should be always used. Then, if there is a post-scheduling phase, round robin should be applied. Although it does not help coalescing, the post scheduler has more freedom and can hide more shuffle code in empty slots. On the other hand, it might increase the number of moves. Here, the choice has to be made dependent on the architecture. On our machine and our benchmarks, there were enough empty slots in the VLIW bundles to hide those additional moves. The benefit from relaxed post scheduling outweighed those extra copies.

Pre-coalescing has a non-negligible overhead but gives very good results and can improve other heuristics, too. This is the main source of tree-scan's performance gain. The caller heuristic is quite expensive and gives bad results if used alone. It should be avoided, unless pre-coalescing is enabled. Together, they are more powerful in avoiding caller-saved registers for move-related variables that are live across calls. We show that splitting before coloring does not give any benefits in terms of run time. As it increases allocation time significantly, it should be avoided in the JIT context.

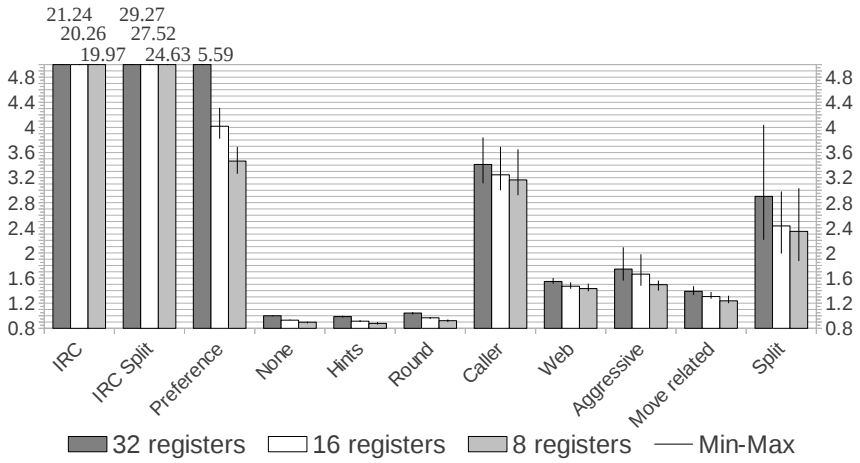


Figure 5.11: Normalized geometric means of memory footprint. Numbers are normalized to tree-scan baseline (None) with 32 registers. Preference reports preference guided numbers. Configurations to the right of None are tree-scan algorithm with the related bias technique. (Lower is better)

5.5.2.4 Footprint

So far we show that tree-scan approach runs faster, produces faster code with a comparable number of dynamic `moves` than `IRC` decoupled approaches and preference guided. However, to be suitable for JIT, it also must have a small memory footprint. This is what we show in that section.

Figure 5.11 reports the normalized footprint of the main classes of approaches. The numbers are normalized to tree-scan baseline (None) using 32 registers. The footprint measure all memory specifically allocated to perform the related algorithm. Thus, it does not take into account the footprint of the program representation, which is the same for all but `IRC` with live-range splitting approach, but it does take into account, for instance, the footprint of a liveness analysis and its results if this approach needs liveness sets information. For a given benchmarks, footprint is summed over all its functions. Due to the amount of reported configurations, only geometric means over all benchmarks are reported. Detailed numbers are available in appendix, Tables A.10 to A.12.

As expected, `IRC` with live-range splitting has the biggest memory footprint as its IG have more variables than `IRC` and uses far more memory, almost 30 times more for 32 registers, than a tree-scan approach. Preference guided allocator consumes also more memory than tree-scan. For 32 registers, this consumption is 5.59 bigger. It decreases rapidly, to 3.46 (3.84 against None with 8 registers), according to the number of registers, whereas tree-scan is not much affected by the number of registers (1 to 0.9). Preference guided is sensible to these numbers because it has to record for each variable its preference set, which size depends on the number of registers. The other difference in size comes from the fact that it needs the liveness sets to build these preference sets.

For the different bias approaches as for tree-scan baseline, the number of registers has a limited impact on the footprint. `Caller` heuristic is the most

expensive bias technique, with 3.41 times the size of the base algorithm. It has to perform a liveness analysis to know which variables are crossing calls and then has to store that information. `Split` also has to perform that analysis but it does not need to store any additional information. As already stated, the `move` related technique uses a pre-coalescer heuristic to know which variable are `move` related. However, it uses less memory than the used coalescer technique, here aggressive, because it uses the result of the analysis, but does not have to store the color information per set nor which variables are in one set.

Like allocation time, to know the overhead of a composed bias technique, the overhead of each technique has to be added. Compared to `IRC Split`, tree-scan has to use the HAC variant to achieve comparable coalescing quality, thus it uses 4.15 more memory than baseline. Consequently it uses still 7.05 times less memory than this technique for the same code quality. Compared to preference guided, we have to consider the number of registers. For 32 registers, HA variant is the closest. In that case, tree-scan uses 3.21 times less memory than preference guided. For 16 registers, HAC is the closest cheapest variant. In that case, tree-scan uses about the same amount of memory as preference guided. Finally, for 8 registers, HW is the closest cheapest variant and it uses 2.42 times less memory than preference guided.

5.6 Conclusion

This chapter has introduced repairing to handle register constraints during register coalescing. Repairing has been shown to be compatible with graph coloring-based coalescers and a new type of SSA-based coalescer called a tree-scan, that does not build an IG and improves significantly upon past linear scan allocators. Our evaluation has shown that a graph coloring coalescer that employs repairing can generate code whose quality is comparable to the most effective prior techniques that handle register constraints. The tree-scan, moreover, runs more efficiently than the graph coloring-based coalescer with repairing because it does not require an IG, while producing code of comparable quality. Moreover, this is also true in the case of the recent scan allocator preference guided. Consequently, we believe that the most reasonable choice for JIT compilers having a decoupled register allocation is tree-scan. Finally Table 5.1 sums up the properties of the proposed approaches for people interested to invest in repairing.

Approach	Pre-condition	Implementation effort	Compile time	Footprint
Freeze + ad-hoc repairing	Have IRC	Low	Medium	High
Conservative + graph coloring repairing	Have IRC	High	High	High
Conservative + ad-hoc repairing	Have IRC	High	Medium	High
tree-scan + bias technique according to time budget	-	High	Low	Low

Table 5.1: Properties of repairing approaches

Chapter 6

Decoupled Graph-Coloring Register Allocation with Hierarchical Aliasing

Although less studied than classical register allocation, register allocation with hierarchical aliasing is a common problem in actual architecture. Aliasing is present in four general purpose x86 registers: AX, BX, CX and DX. Each of these registers has two aliases, e.g., the 16-bit register AX is divided into two eight-bit registers: AH and AL. Aliasing is also found in floating point registers of many architectures typical of the embedded world, such as ARM and PowerPC, where single precision registers combine to make double precision ones. Architectures such as ARM Neon go further, allowing the combination of two doubles into a quad-precision register. There exist also more irregular architectures, such as the Carmel model, used in digital signal processors, showing overlapping registers of 16, 32 and 40 bits [96].

As already stated, when aliasing is involved static single assignment (SSA) split point are not sufficient to be able to decouple the spilling and coloring phases when this latter uses graph-coloring. Indeed, in that configuration the coloring is NP-complete [9, 69], thus even if a coloring exists, graph-coloring heuristics may not find it. In such configuration, more split points need to be inserted. Although fundamental, this notion of live-range splitting makes it difficult to extend decoupled algorithms to architectures with aliased register banks. Previous solution would split live-ranges between each pair of consecutive instructions [85], creating a program representation called *Elementary Form*. However, this level of live-range splitting makes traditional register allocators, like those based on partitioned Boolean quadratic programming [96], integer linear programming (ILP) [67] or graph coloring [33], impractical, because the number of program variables increases too much.

In this chapter we solve this problem introducing a program representation that we call *Semi-Elementary Form*. Programs in this format provide the essential property required by a decoupled register allocator: the local register pressure at any given program point equals the weight of variables alive at that point. Because the semi-elementary form does much less live-range splitting than the original elementary form, it fosters decoupled allocators that are faster,

require a smaller memory footprint and, as a side effect, yield better register coalescing when submitted to traditional coalescing heuristics. We also introduce a way to merge the live-ranges of variables – the *local merging test* – which reduces even more the size of the program's interference graphs, and speeds-up allocation time considerably. Finally, we provide as a bonus an improved spilling test, that might produce less spilling than the simplification heuristics traditionally used in graph-coloring based register allocation.

The semi-elementary form speeds up register allocation; however, it is not a new register allocation algorithm. Hence, it does not increase the performance of the assembly code produced in any substantial way. Although it has the side effect of reducing the number of copies in the final assembly code, this reduction is too small to provide performance gains. Nevertheless, it considerably simplifies register allocation, and we believe that this is the best way to handle register aliasing in decoupled allocators. To substantiate this claim, we have adapted two different graph coloring-based register allocators to run in a decoupled fashion: George and Appel's iterated register coalescer (IRC) [51] and Bouchez *et al.*'s brute force coalescer (BF) [19]. We show, via experiments, that building semi-elementary form programs is fast. Furthermore, allocators working on semi-elementary form programs consumes much less memory than elementary-form based approaches, and are much faster. In our experiments we compile the SPEC CPU 2000 benchmarks to miniIR assembly, using 8, 16 and 32 aliased registers.

The rest of this chapter is organized as follows: Section 6.1 gives more background on the register allocation with aliasing. Section 6.2 introduces a spilling test in face of aliasing. Section 6.3 presents our semi-elementary form, and Section 6.4 provides experimental data supporting our techniques. Finally, Section 6.5 concludes this paper.

6.1 Background

Figure 6.1 illustrates the traditional graph coloring approach. Figure 6.1(a) shows an example program, and Figure 6.1(b) outlines its interference graph. In this example, we assume that lower-case names denote 32-bit floating-point variables, while upper-case names denote 64-bit doubles. If we assume an architecture with two 64-bit registers, each having two 32-bit aliases, then the graph in Figure 6.1(b) is not *colorable*. That is, no register assignment keeps all the variables simultaneously alive in registers. The register allocator normally solves this problem via *spilling*. In Figure 6.1(c) we have sent variable `a` to memory; thus, creating two new variables, $\mathtt{a}_0$, at the definition point of `a`, and $\mathtt{a}_1$ at its use point. The new interference graph, given in Figure 6.1(d) is now colorable.

This program does not match the essential property required for decoupled register allocator:

Property 1. *The maximum* register pressure *at any program point equals the global register pressure.*

Lee *et al.* [69] have proved that register allocation with two-level aliasing is NP-complete even for SSA form programs without branches. Thus, in face of

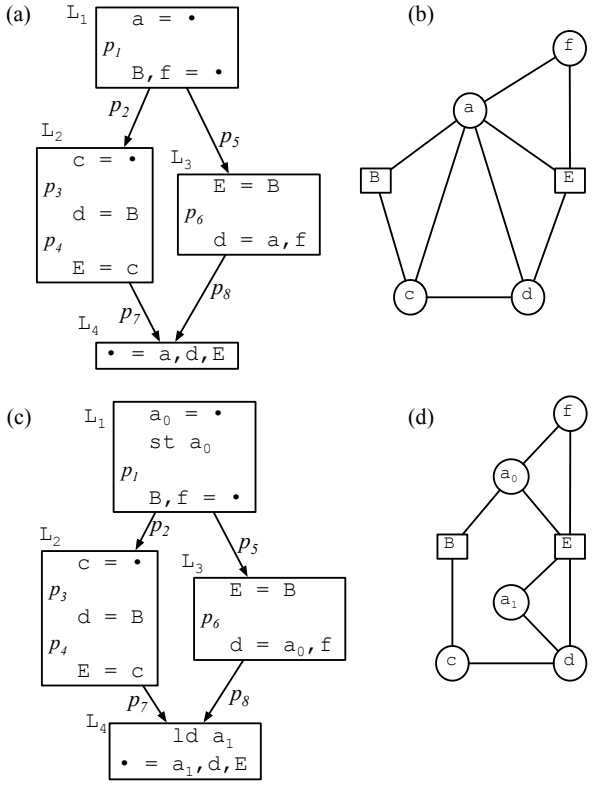


Figure 6.1: Traditional graph-coloring-based register allocation. (a) Example program. (b) Program's interference graph; square nodes plus upper case letters denote double precision values. (c) Program after spilling variable a. (d) New interference graph.

aliasing, the SSA form conversion is not extensive enough to guarantee Property 1; instead, *elementary form* can be used. We convert a program to elementary form via the insertion of parallel copies between each pair of consecutive instructions. Figure 6.2(a) shows our running example in elementary form. The interference graph of the new program, conveniently called an elementary graph, is given in Figure 6.2(b). Elementary graphs have very simple structure; thus, determining the local register pressure usually has a polynomial time solution, even when nodes are allowed to have weights 1, 2 or 4, as in our case of aliased register allocation. The variables in the program given in Figure 6.2(a) can be allocated into our register bank made of two 64-bit registers and four aliased 32-bit registers; an improvement on the original program seen in Figure 6.1(a). This result is not a coincidence: any program can be transformed into the elementary form, and the elementary form program never requires more registers than the original code.

A heavy price incurred by the conversion into elementary form is the growth in the program size. For instance, the interference graph in Figure 6.1(b) has

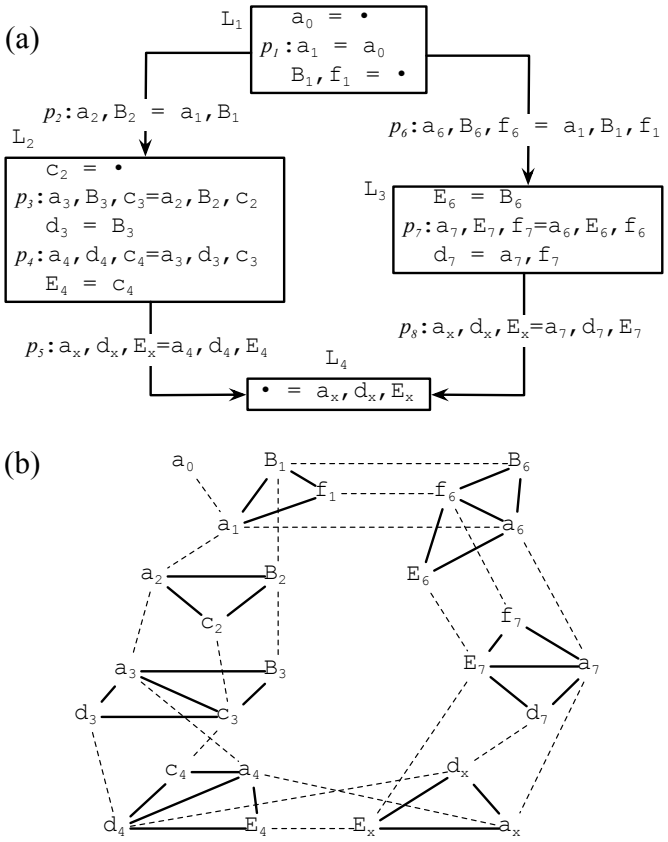


Figure 6.2: (a) The program from Figure 6.1 in elementary form. (b) The interference graph of the elementary program.

six nodes, but the corresponding elementary graph seen in Figure 6.2(b) has 26. This explosion is observed in actual benchmarks. Figure 6.3 compares the size of program functions taken from SPEC CPU 2000 before and after the conversion into elementary form. This transformation tends to increase quadratically the number of variables in the intermediate representation.

In order to explain our ideas, we have adapted two different graph coloring based register allocators to run in a decoupled fashion in face of register aliasing. The first is IRC of George and Appel [51], and the other is BF of Bouchez *et al* [19]. Figure 6.4 shows our version of these algorithms. Comparing Figure 6.4(a) and Figure 2.4 it is easy to notice that the decoupled version has less iterations between its phases. Both these algorithms use the extensions of Smith, Ramsey and Holloway [99] to deal with aliasing, which we re-introduce later. Most of the phases that constitute each algorithm, i.e, simplify, coalesce, freeze and select have been thoroughly described in previous works [51, 19]. Decoupled register allocators in general also use a phase called *patch*, related to the implementation of parallel copies. After register allocation, the compiler must

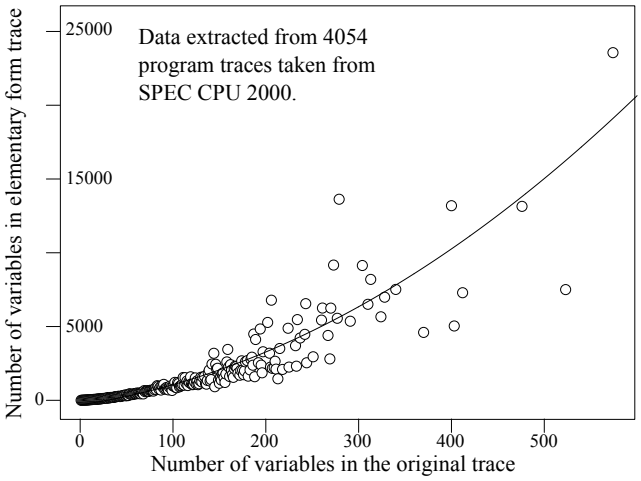


Figure 6.3: The growth in the number of program variables due to the conversion to elementary-form.

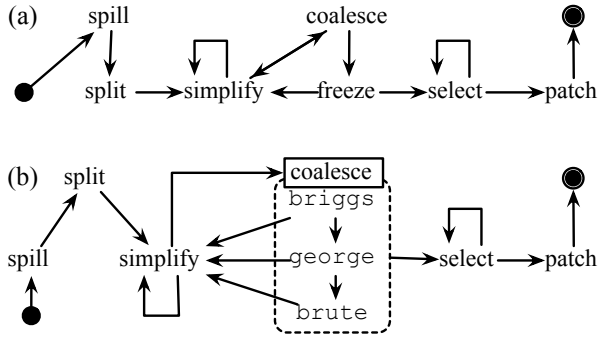


Figure 6.4: (a) A decoupled re-implementation of IRC. (b) A decoupled re-implementation of Bouchez's BF that handles aliasing.

implement these parallel copies, using the instructions present in the target architecture. Parallel copy patching has been thoroughly described by Pereira and Palsberg [86].

The brute-force algorithm, outlined in Figure 6.4(b), has a more modular design than IRC. After spilling is performed, BF orders the copies in the source program according to their *profitability*, and try to coalesce them following this ordering. The profitability of a copy is a measure of how much improvement its elimination can bring to the target code. Copies inside deeply nested loops tend to be more profitable than copies outside loops. We say that the coalescing of vertices a and b is *conservative* if the interference graph that we obtain after collapsing these nodes into a single node ab can still be allocated with the available registers. Brute Force uses one of the following three tests, in order,

to guarantee that the coalescing of copy $a = b$ is conservative:

1. `Briggs`(a, b) [26]: the merging of a and b will create a node ab with fewer than k neighbors with squeeze greater than k.

2. `George`(a, b) [51]: assuming that a is a pre-allocated variable, then every neighbor of a already interferes with b, or has squeeze less than k. Notice that we must also try `George`(b, a), as this rule is asymmetric.

3. `Brute`(a, b) [19]: the graph that results from merging a and b can be colored with k colors. We do this check in polynomial time, given Property 1.

6.2 Spilling Test in Face of Aliasing

Decoupled register allocation is interesting as long as it does not cause more spilling than traditional graph-based register allocators do. The elementary form is an easy way to provide this guarantee. Given that the conversion to elementary form divides the source program in regions that are very small and simple, the problem of determining the local register pressure for each region has polynomial time solution, at least for architectures with quad, double and single registers, such as x86, ARM, PowerPC and SPARC. The polynomial time solution still holds in face of pre-allocation, a phenomenon caused by architectural constraints that force variables to be assigned to particular registers [85].

6.2.1 Checking Colorability via Smith's Simplification Test

In the case of both IRC, and BF, the spilling phase must guarantee that the program it passes forward to the other phases of the register allocator has an interference graph that is greedy k-colorable, where k is the number of registers. In the presence of aliasing, the simple test based on the node degree is not enough to check for greedy k-colorability. A correct test has been devised by Smith *et al.* [99], using Fabri's idea of squeeze factor [41]. In Smith *et al.*'s framework, the computer architecture provides a number of register classes, which might alias in several ways. Each variable must be assigned to registers in a specific register class. The squeeze of a variable is the maximum number of registers, in its class, that could be denied to it, given a worst case allocation of its neighbors. Thus, a node v can be simplified if the worst case allocation of all neighbors of v is less than v's squeeze factor. Figure 6.5 illustrates this idea, assuming an architecture with double (R) and single (r) precision register classes. Figure 6.5(a) shows a subgraph of the graph given in Figure 6.2(b). Each vertex has been augmented with the squeeze factor of the variable that it represents, as determined by Smith *et al.*'s simplification criterion. For instance, variable $\mathtt{B}_6$ needs a double precision register, and has two neighbors, which could be assigned to aliases of different double-precision registers; thus, its squeeze factor is 2. We use the suffix R in $\mathtt{B}_6$'s squeeze factor to indicate its register class. The squeeze of a variable is bounded by the number of registers in this variable's class; hence, the squeeze of $\mathtt{a}_6$ or $\mathtt{f}_6$ is 4, although the worst case allocation, assuming an unbounded number of registers in class r would be 5 for any variable. Notice that the interference graph of the variables alive between two consecutive instructions is very simple: it consists of two cliques only. Thus, we can compute the squeeze factor of each variable simply counting variables simultaneously alive.

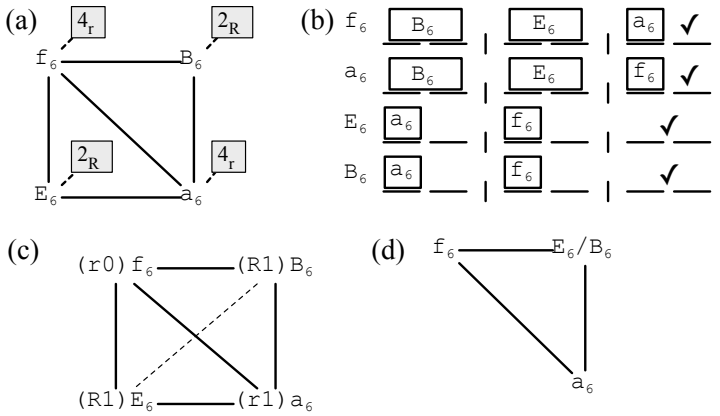


Figure 6.5: Smith *et al.* Simplification test. (a) A connected component of the graph in Figure 6.2(b). The nodes are labelled with their squeeze factors, e.g., the worst case allocation of E_6's neighbors takes off two registers of class R. (b) Worst case allocation for each variable. (c) A tight allocation produced by a puzzle solver [85]. (d) Variable merging guided by the puzzle solver. Architectural definition: r = $\{r_0, r_1, r_2, r_3\}$, R = $\{R_0, R_1\}$. Aliases: $\{(r_0, R_0), (r_1, R_0), (r_2, R_1), (r_3, R_1)\}$.

6.2.2 Correct Spilling Test Handling Aliasing and Precoloring

A fundamental question that concerns a decoupled register allocator is "which spilling test should we use to ensure that after spilling we will be able to color the program's interference graph using the algorithm's graph coloring technique?" To answer this question one must be aware that after spilling and live-range splitting no more spilling must be necessary. The choice of the graph coloring technique is an important player in this game because a given heuristic may fail to color a graph that is actually colorable, after all, graph coloring is a NP-complete problem. Many allocators use Kempe's simplification test as the coloring heuristics. We are no exception.

The notion of greedy k-colorability, based on Kempe's test, is an over-approximation of colorability; however, this approximation is tight if we do not have to handle register aliasing. That is, in the absence of aliasing, if G is an elementary graph, then G is greedy k-colorable if, and only if, G is k-colorable. The proof of this statement follows from Bouchez's result for SSA-form programs without aliasing [17]. Therefore, without aliasing, answering the initial question is very simple: the spilling test is as simple as counting the number of variables alive at each program point.

In the presence of aliasing, greedy k-colorability is different than colorability, as the example in Figure 6.5(a) shows. Furthermore, a combination of precoloring and aliasing may lead to situations in which every connected part of an elementary graph is greedy k-colorable, but the global graph is not, as the example in Figure 6.6 illustrates. We call the interference graph formed by the

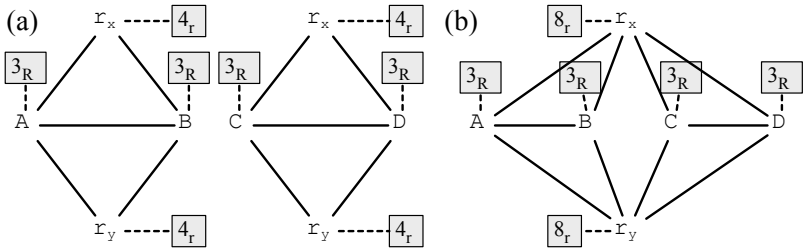


Figure 6.6: (a) Two greedy k-colorable elementary graphs. (b) The whole graph is non-greedy k-colorable.

live-ranges live in, out and across an instruction *local*. Pre-colored nodes bind many local graphs together. Thus, the global squeeze factor of pre-colored nodes may be larger than their squeeze factor taken into consideration at each local graph. The consequence of this observation is that for a decoupled approach that performs the coalescing/coloring steps via Smith *et al.*'s method to be correct in the presence of aliasing and pre-coloring, we need to perform the spilling test carefully. In other words, we must start the simplification process from the uncolored nodes, leaving the pre-colored nodes to the end. Theorem 3 proves the correctness of this procedure.

Theorem 3. *If every connected component of an elementary graph is greedy k-colorable starting the simplification process from the uncolored nodes, then the whole graph is greedy k-colorable.*

Proof. Any non-pre-colored node interferes only with nodes in its connected component, even taking the whole graph into consideration. Hence, the squeeze factor of these nodes is the same in the local and global interference graph. After these nodes are simplified, we are left with pre-colored nodes only. These nodes must be simplifiable, because they represent the registers in the actual architecture. □

Therefore, a possible spilling test is to check the colorability of each local graph via this method. Note that it does not mean that all split points will be necessary as we will show in Section 6.3. Figure 6.7 illustrates that.

6.2.3 Improving Smith's Test with Live-Range Merging

Assuming only two double-precision registers, the squeeze-based simplification test would fail to simplify any node in Figure 6.5(a), and some variable would have to be spilled. On the other hand, there exists a register assignment that accommodates all the variables, as Figure 6.5(c) shows. In order to improve Smith *et al.*'s simplification test, we do live-range merging whenever we are unable to simplify any variable. To be able to do that, the live-ranges to be merged must not interfere globally. Here we assume that the initial code is in SSA form, therefore the local test is enough to get this information. Algorithm 11 gives the pseudo code of the related spilling test.

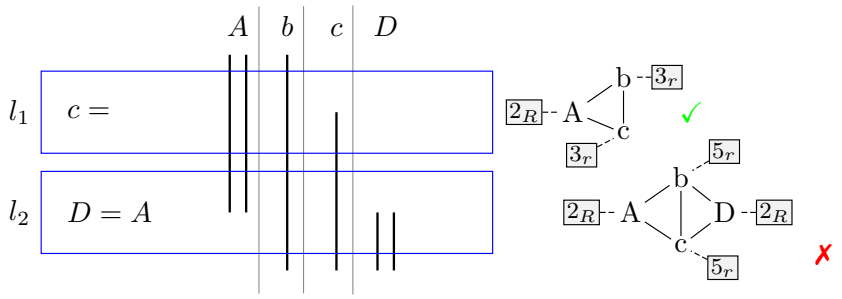


Figure 6.7: Spilling test based on Smith's simplification test. In that example, the register file has two levels, with 4 registers at the lowest level and 2 registers at the highest level. The first instruction l_1 has a local interference graph fully simplifiable, thus no spill is required. The second instruction l_2 has a local interference graph not simplifiable as all squeeze factors are bigger than or equal to 4 for low level variables (resp. 2 for high level). Hence, for l_2 the spiller has to choose one of the variable to spill.

Algorithm 11 A correct spilling test in face of aliasing and precoloring featuring live-range merging. Given sets represent the related information for the given instruction.

```
1:  procedure LiveRangeMerging(Instruction inst, Set last_uses, defs, live_through)
2:      live_local ← last_uses ∪ defs ∪ live_through
3:      while live_local ≠ ∅ do
4:          if ∃v ∈ live_local : v is simplifiable then
5:              simplify(v)
6:              live_local ← live_local \ {v}
7:          else if ∃o ∈ defs and i ∈ last_uses: size(o) = size(i) then
8:              let i, o be the largest pieces that fullfilled the condition
9:              a ← merge(i, o)
10:             live_local ← live_local \ {i, o} ∪ {a}
11:         else
12:             let v ∈ live_through : v ∉ inst.uses
13:             spill(v)
14:             live_local ← live_local \ {v}
```

When merging variables, we start with pairs of variables in register classes with the largest size, Line 8, because this strategy reduces more drastically the squeeze factor of the other variables alive in that program point. Another important detail of our algorithm is the fact that we use live-range merging with discretion. If we are stuck in the simplification process, then we choose only one pair of pieces, merge them, and re-try the simplification test. We proceed in this careful fashion because merged variables will be assigned the same register. This restriction might have the undesirable side effect of constraining too much the register coalescer that will run after spilling takes place. In fact, this merging can be completely virtual. The spiller knows that there is a valid coloring and does not spill anything. In this case, the coloring phase has to provide the same feature. Going back to Figure 6.7, with live-range merging enabled, variables A and D can be merged, thus no spilling is required.

We do no apply live-range merging at program points that contain pre-allocated variables. Pre-allocation might prohibit the merging of live-ranges, and, in face of this phenomenon we fall back to Smith *et al.*'s simplification test.

6.3 Semi-Elementary Form

Traditional coalescing tests, such as George's [51] or Briggs's [26] have a number of disadvantages if used on elementary graphs. The first disadvantage is in terms of runtime. Each of these tests would have to be invoked once for each affinity edge in the elementary graph. The second disadvantage concerns the quality of the code produced. In the presence of aliasing, the traditional coalescing techniques may fail to eliminate copies, even though they are not necessary. For instance, Figure 6.8 shows a program in elementary form, in which every copy could be completely coalesced away. However, neither George nor Briggs rules would be able to coalesce the inner copies. This limitations happen because these rules are applied *sequentially*. Coalescing would be possible if all the affinity edges were analyzed in parallel.

The rational behind the elementary form is to reduce the amount of spilling during register allocation. With such purpose, the conversion to elementary form splits the live-ranges of the variables at every program point. However, most of these splits are unnecessary. We have developed two techniques to reduce the size of the program's interference graph. The first technique, that we call *the critical node test*, is based on a criterion that avoids splitting live-ranges whenever possible. We call the program representation that results from this method *semi-elementary form*. The second technique merges variables, whenever it is conservative to do so. In order to perform this merging we rely on a method that we call the *local merging test*. We explain these two strategies in the rest of this section.

6.3.1 Criterion to Avoid Live-Range Splitting

An elementary graph if formed by many unconnected components, which represent the live-ranges of variables at some particular program point. Therefore, we expect a lot of redundancies between graphs formed from consecutive instructions. Given two instructions, the guider and the follower, all the vertices

```
byte f(int A, byte b) {
  while (true) {
    int C;
    byte e;
    C, e = div (A, b);
    A = C + e;
  }
  ret b;
}
```

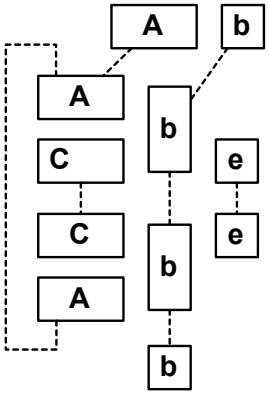


Figure 6.8: Example showing the deficiencies of traditional coalescing techniques.

that correspond to variables live-in at the follower are connected through affinity edges to the vertices in the guider. The only vertices in the follower's graph which have no affinities for vertices in the guider's are those nodes that represent variables defined in the follower instruction. We call them *critical nodes*. Usually an instruction defines at most one variable so as the number of critical node. We except then to perform this test quickly. In light of these observations, our criterion to avoid live-range splitting is as follows:

> **The critical node test:** if every critical vertex in the follower's graph has a squeeze factor less than k, then it is not necessary to insert a parallel copy between guider and follower to achieve Property 1.

Theorem 4. *Let G_g and G_f be the interference graph at the guider and the follower, as previously defined. If G_g is greedy k-colorable, then the graph that results from merging G_g and G_f via the critical node test is greedy k-colorable.*

Proof. The proof is straightforward: if the merging is done, the resulting graph is formed by all the nodes from G_g plus the critical nodes in the follower. Because of our criterion we know that every critical node can be simplified. Once they are simplified, we fall back into G_g, which, by hypothesis, is greedy k-colorable. □

Figure 6.9 illustrates our method when applied to the sequence of instructions from program point p_1 to p_5 in Figure 6.2(a). We have augmented the graphs in Figure 6.9(a) with the squeeze factor of each node, and we have highlighted the squeeze factor of each critical node in the next figures. Considering two double precision registers available, we can avoid all the parallel copies but the last, because the squeeze of $\mathtt{E}_4$ is 4. On the other hand, if applied on the program in Figure 6.8, the critical node test would avoid every live-range splitting. In this case, the semi-elementary form program equals the original program converted to SSA form.

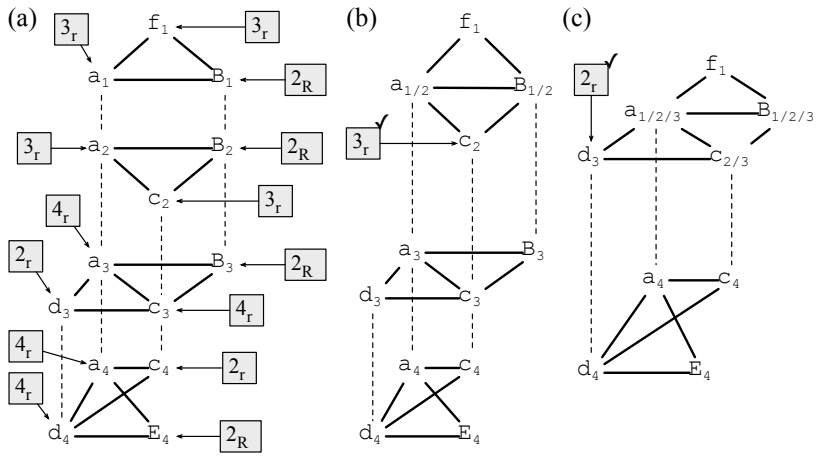


Figure 6.9: Construction of semi-elementary form. (a) a subgraph of the graph in Figure 6.2. (b) and (c) Subgraphs that result from avoiding the insertion of two parallel copies. We cannot avoid the last parallel copy, otherwise we would build a graph that is non greedy k-colorable.

The critical node test avoids splitting live-ranges unnecessarily. As the experiments from Section 6.4 show, the interference graphs of semi-elementary form programs that we found are about eight times smaller than the corresponding graph of elementary-form programs; however, the former graphs are still approximately twice as big as the interference graphs of the original programs. In order to avoid this growth, we can go even further, merging live-ranges of non-affinity related variables whenever it is conservative to do so. We call this type of preprocessing the *local merging* of live-ranges, and explain it in the next section.

6.3.2 Local Merging of Live-Ranges

To further reduce the size of the interference graph, we can merge some non-affinity related variables, using a technique based on a coloring oracle. The oracle says for two variables whether they may be colored the same way or not. For instance, performing linear-scan [89] on the program assign each variable a color. This color can then be used to coalesce nodes that are not affinity related. In this work, we based our oracle on punctual coalescing [87]. We made this choice because this technique works locally, is fast and has been specifically designed to coalesce in face of aliasing.

Punctual coalescing is a strategy used in conjunction with puzzle-based register allocation to remove copies in the target code. Puzzle-based register allocation performs the coloring by solving a set of puzzles where the pieces are the variables and the board is the register file. Each puzzle represents the register allocation local to one instruction. It is composed by the pieces of the variables locally alive and a board which shape is a rectangle. This rectangle is defined by two lines as large as the register file. The first line denotes the space that will be

occupied by live-in variables and the second line the space for live-out variables. From this description, the shape of the pieces involved for a puzzle is straighforward. The width of the piece that represents a variable equals the width of its register class. Its height equals one for last used and defined variables, two otherwise. Then, the goal is to place as much pieces in the board as possible, depending on a cost function, while preserving the semantic of the program, i.e. a live through variable must be on both lines, definitions on the second line, last uses on the first line and the alignement. In our case, the spilling test ensures that all variables will fit the board. In that model, pre-colored variables discard the related places in the rectangle. Moreover, it features hierarchical aliasing, thus each piece have an alignment constraints.

The punctual coalescer traverses the dominator tree of the source program, analyzing one instruction at a time. The algorithm processes the interference graph formed by variables alive around this instruction, remembering the allocation of the previous instruction. It is a locally optimal approach; that is, given only the knowledge of the variables alive across two consecutive instructions, it finds the largest number of matches between variables that do not compromise Property 1. We use the results that we get from the punctual coalescer to design a local live-range merging method. Our live-range merging technique based on punctual coalescing is given below:

- For each pair of consecutive instructions, *guider* and *follower* inside a basic block, let G_g and G_f be the local interference graphs that denote the register allocation problem for each instruction.
- We let the punctual coalescer [87] place in the same registers the vertices that have affinities. The punctual coalescer tends to maximize the number of matches between two consecutive instructions.
- For each pair of same-size variables $u \in G_g$, and $v \in G_f$, that have been assigned the same register r:
    1. If the vertex uv that results from merging u and v does not interfere with any vertex w that has been assigned r or an alias of r by the punctual coalescer, then replace u and v by uv. This type of interference might happen if u and v have non-contiguous live-ranges, and w is alive between the kill site of u and the definition site of v.
- For each s denoting a variable defined in the *follower*:
    1. If s has a squeeze factor greater than the number of registers in the register class of s, then undo every merging of the previous step.

We only merge live-ranges inside the same basic block, because, by merging non-affinity related variables, we may eliminate coalescing opportunities. As we show in Section 6.4, punctual merging decreases the capacity of both, IRC and BF to eliminate copies in the final assembly code. Figure 6.10 illustrates punctual merging. We have used a different example this time, because our running example from Figure 6.2 is not complex enough to exercise the interesting aspects of punctual merging. Notice that the critical node test, when applied on Figure 6.10(b) would not insert the paralle copy on p_1, thus only merge the

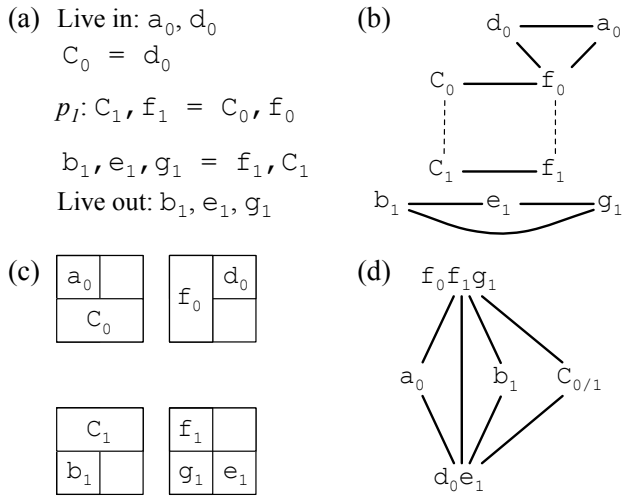


Figure 6.10: A constructed example showing punctual merging. (a) The elementary-form program. (b) The interference graph. (c) The solution of punctual coalescing. (d) The solution of punctual merging.

variables `C`'s and `f`'s. However, assuming a solution of punctual coalescing that places variables f_0, f_1 and g_1 into the same column, we can also merge these pieces. The same happen with – non-contiguous – variables d_0 and e_1. On the other hand, we cannot merge variables a_0 and b_1, because C_0 and C_1 have been allocated to aliased of the registers assigned to a_0 and b_1. If we merged a_0 and b_1, then the resulting variable would interfere with both C_0 and C_1.

6.4 Experiments

Testing Environment The algorithms were implemented in Python, producing code to a prototype architecture called MiniIR [1], which is based on the YAML [2] serialization format. YAML is used by STMicroelectronics *Inc* to quickly prototype hardware. MiniIR provides a minimalist textual machine level intermediate representation to be used for experimental tools. We report numbers for the x86 architecture, which we described in miniIR (Figures 6.13, 6.14 and 6.15), and for an artificial architecture with 8, 16 and 32 registers, also described via miniIR (Figure 6.12). In this case, each register has 32 bits, and is divided into two 16-bit aliases. We have checked the validity of each register allocation using the type-system of Nandivada *et al.* [77]. We chose to run our experiments on SPEC CPU 2000, which we have compiled into MiniIR using LLVM 2.7 [68].

[1] `http://www.assembla.com/wiki/show/bE6Ve4RQir36HF eJe5cbLr`
[2] `http://www.yaml.org/`

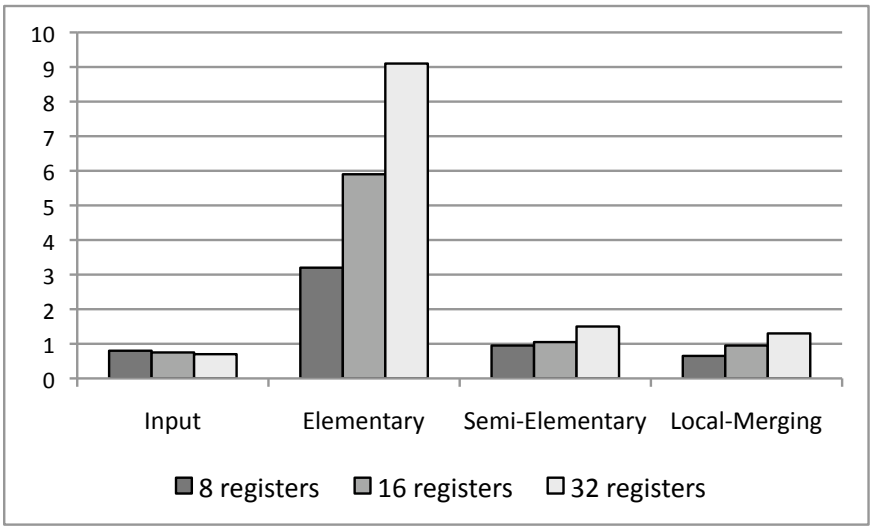


Figure 6.11: Number of nodes, in millions, of the interference graphs of different program representations. ***Input:*** input graph passed to the original iterated register coalescing algorithm – the number of nodes is the number of variables in the source program.

The Spilling Approach All the numbers in this section refer to greedy k-colorable interference graphs. That is, because we do decoupled register allocation, we perform spilling before doing register assignment. We are using a simple spill heuristic: whenever the register pressure is too high at some program point, we choose the variable that has the furthest use in a linearization of the control flow graph, and spill it using spill everywhere.. We check the register pressure via the improved Smith test that we have described in Section 6.2. To spill we replace the definition of the variable by a store, and each use by a load. We do not try to re-use loads.

Size of Interference Graphs The chart in Figure 6.11 illustrates the effectiveness of the conversion into semi-elementary form in order to reduce the size of the interference graphs. The interference graphs of elementary-form programs, on average, are 8 times larger than the interference graphs of the original programs. This difference falls down to 2 times if we use semi-elementary form instead. If we do local merging, then we obtain interference graphs that are even smaller than the graphs produced for the original program, when we use eight registers only. The graphs tend to become larger as more registers are taken into consideration, because the amount of spilling decreases, but the proportion of variables having different sizes remains the same. Thus, although we spill less when more registers are available, we still must perform live-range splitting to avoid spilling, as we do in Figure 6.1. This fact also explains why the input graphs are smaller when we have more registers available: in this case, less spilling will happen, and fewer variables will be created to hold the source/destination of stores/loads.

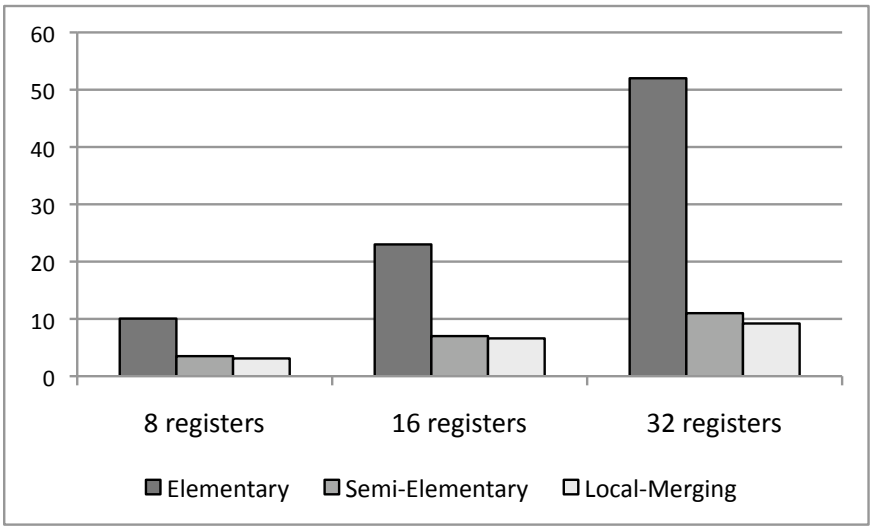


Figure 6.12: Allocation time, in hundreds of seconds, for different kinds of interference graphs.

Allocation Time The size of the interference graph has a direct impact on the allocation time, as Figure 6.12 shows. Considering eight registers only, allocation in semi-elementary form is 3.1 times faster than in elementary form. This different increases to 3.3 times if we perform local live-range merging. With 32 registers the difference is even larger. Semi-elementary form speeds-up register allocation by a factor of 4.7x, and local merging moves this factor to 5.5x.

To put these results in perspective, let's consider the largest function that we found in our benchmark, assuming eight registers. This function, in SSA-form, has 10,163 variables. The interference graph of the elementary program contains 99,364 nodes. On this graph, IRC takes 4,544 seconds. This is 49.40x slower than the punctual coalescer [87], which does not build an interference graph. The semi-elementary form program has an interference graph with 19,024 nodes. In this case, IRC is 5,30 times slower than the punctual coalescer. After local live-range merging we have a graph with 13,764 nodes, in which IRC is 1.82x slower than punctual coalescing. Thus, IRC is 27.2x faster on a graph after local live-range merging than on an elementary graph.

Effectiveness of Live-Range Merging on Coalescers The semi-elementary form not only reduces the size of the interference graph, but also improves the effectiveness of copy coalescing, as we show in Figures 6.13, 6.14 and 6.15. Figure 6.13 shows only the result of IRC using Smith *et al.*'s [99] extensions, implemented according to Figure 6.4(a). The algorithm executing over elementary graphs left 3418 copies on the SPEC CPU 2000 programs. On semi-elementary form this number falls down to 3221 copies. If we perform local live-range merging on the source programs, then IRC leaves 4133 copies on the assembly code. In these experiments we count only copies inserted into the program to do live-range splitting; that is, we do not count the copy instructions that were part of

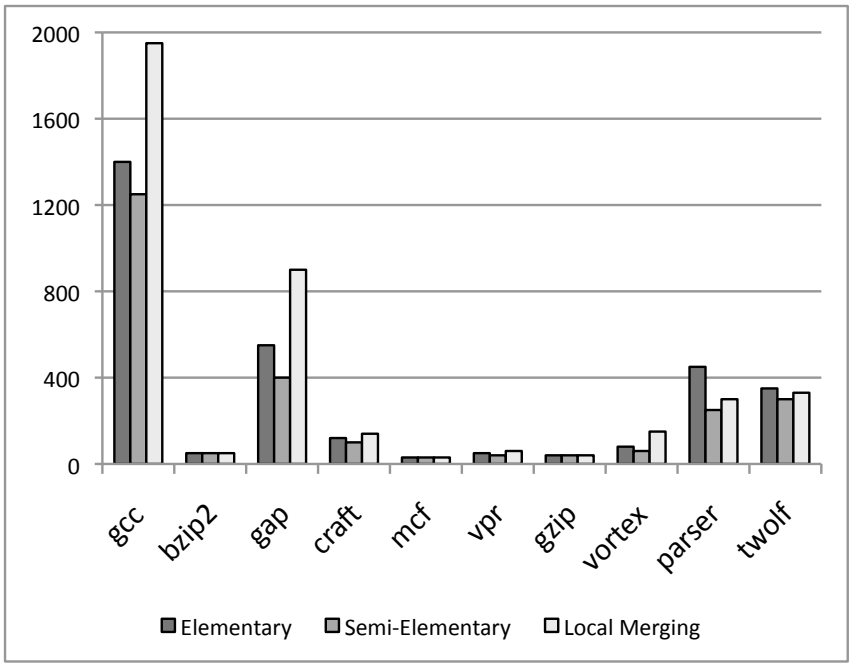


Figure 6.13: Number of copy instructions left by IRC when running on different program representations.

the source program, even though many of them were coalesced too. In this way, we focus on the ability of the allocators to handle aliasing. The local live-range merging decreases the coalescing power of IRC, when compared to the results that we obtain by using semi-elementary form. We speculate that this fact happens because the punctual merging constraints too much the interference graph, creating nodes with larger squeeze factors.

Figure 6.14 shows the effectiveness of BF implemented using the modifications from Section 6.1, following Figure 6.4(b). Running the algorithm directly on elementary-form or semi-elementary form programs leaves 2803 copies on the final assembly code produced by the algorithm. Local live-range merging degrades the ability of the coalescer to eliminate copies. In this case, BF leaves 3503 copies on the assembly programs.

Figure 6.15 compares the three algorithms: BF, IRC and punctual coalescing [87] in terms of the number of copies that each algorithm eliminates via coalescing. We have chosen the best configuration for each algorithm: IRC and BF run on semi-elementary graphs, while punctual coalescing can only run on elementary-form programs. Confirming previous results [19, 87], BF is the most effective algorithm, followed by IRC. Also, as reported by Pereira and Palsberg [85], the punctual coalescer increases the final assembly programs by 6.8% on average. This relatively bad result of the punctual coalescer is due to the fact that it is a local approach, which does not attempt to eliminate copies between basic blocks.

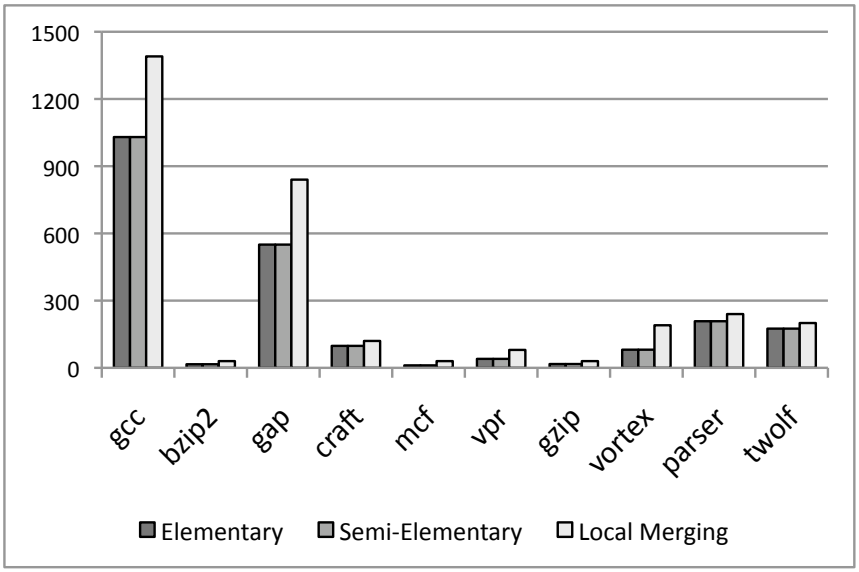


Figure 6.14: Number of copy instructions left by BF when running on different program representations.

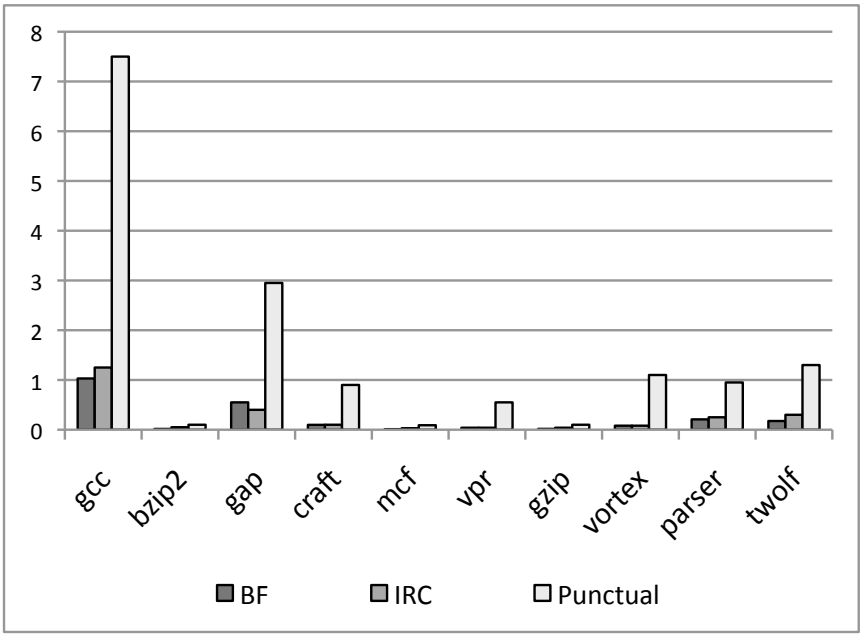


Figure 6.15: Number of copy instructions left by three different register coalescers, in thousands. IRC and BF run on semi-elementary form, and the punctual coalescer runs in elementary form programs.

6.5 Conclusion

We have shown how to decouple graph coloring-based register allocators in face of aliasing. In particular, we described a spilling test that is compatible with the simplification scheme used for the coloring in those approaches. Moreover, we have introduced a number of techniques that make these allocators more practical and effective in the presence of live-range splitting. Live-range splitting helps to decrease the number of variables spilled during register allocation. However, in order to produce code to architectures with aliased register banks, previous register allocators use a very aggressive form of live-range splitting – the elementary format – which would increase too much the size of the program's interference graph, in addition of potentially causing the insertion of extra copies into the final assembly code. Our new techniques allows the register allocators to use all the power of the elementary format, while at the same time avoiding the size explosion, and decreasing the amount of copies into the assembly program. Finally, these techniques can be applied either directly on the program to remove or not to insert some split points or directly on the graph to merge several move related or not move related variables that incremental conservative coalescing technique would fail or not consider to merge.

Part IV

Post Phases

At this point of the compilation flow, we performed the register allocation in a decoupled fashion. We saw that the spilling phase can be decoupled from the coloring even when register constraints are involved. However, this came at the cost of some live-ranges splitting. Thus, at this stage, (parallel) copies instructions may remain. This amount depends on the quality of the coalescing.

This part of the manuscript details two approaches to move or remove more copies. The first one, parallel copy motion, focuses on eliminating the copies from control flow graph (CFG) edges as they may appear for instance using static single assignment (SSA). It features a nice formal model to move copies from one place to another. Using this approach, we are able to avoid to split a CFG edge, otherwise required to instantiate the copies, when this edge splitting turns out to produce poor code. The second approach eliminates copies directly on a data dependence graph (DDG). The idea remains the same than the previous approach but it is more powerful as it can reorder the instructions. For both approaches, we detail the basics. These foundations can then be extended to lead to the design of more sophisticated heuristics.

Chapter 7

Parallel Copy Motion

In back-end code generators, register coalescing means allocating to the same register two variables involved in a `move` instruction so that the copy can be removed. The register coalescing problem is the corresponding optimization problem, i.e., how to map variables to registers so as to reduce the cost of the remaining copies. Before quite recently, this issue was not very important because, usually, the codes obtained after optimization did not contain many `move` instructions. Even if they did, register coalescing algorithms, such as the iterated register coalescer (IRC) [51], were good enough to eliminate most of them. Today, the context of static single assignment (SSA) but also just-in-time (JIT) compilation has put the register coalescing problem in the light again and raised new problems.

The time and memory footprint constraints imposed by JIT compilation have led to the design of light-weight register allocators, most of them derived from a "linear scan" approach [89, 103, 76, 105, 95]. These algorithms perform a simple traversal of the basic blocks, without building any interference graph, in order to save compilation time and space. To make the technique work, `move` instructions may need to be introduced on control flow edges so that the register allocations made in previous predecessor blocks match. Since these register allocators are designed to be fast, they usually use cheap heuristics that may lead to poor performance. In particular, many `move` instructions can remain, which, in addition, can lead to edge splitting, i.e., the insertion of a new basic block where register-to-register copies will be performed. Similar situation occurs in the design of register allocators based on two decoupled phases as we showed in Chapter 2.

These two situations illustrate the need for a better way of handling parallel copies, not only in a JIT context: some JIT algorithms perform coalescing poorly, so a fast and better coalescing scheme is needed, and some algorithms (JIT or not) rely on the insertion of basic blocks, i.e., on edge-splitting, which is not always desirable:

- edge splitting may add one more instruction (a `jump`), a problem on highly-executed edges;
- splitting the back-edge of a loop may block the use of hardware loops as found on some DSPs (e.g., [102, 45]);
- compilers may insert "abnormal" edges [75], i.e., control flow edges that

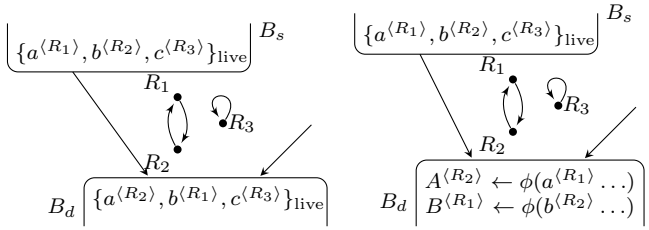
cannot be split (for computed goto extensions, exception support, or region scoping);
- copies inserted on critical edges cannot be scheduled efficiently without additional scheduling heuristics (speculation, compensation), especially on multiple-issues architectures.

The goal of this chapter is to propose a general framework for moving around parallel copies in a register-allocated code. Section 7.1 illustrates the concept of parallel copy motion inside a basic block and out of a control flow edge. For a critical edge, moving copies is more complicated, as some compensation on adjacent edges must be performed, then possibly propagated. Section 7.2 describes more formally our method, which is based on moving permutations of register colors (possibly with compensation). In Section 7.3, we develop simple heuristics to optimize the placement of moved parallel copies and address our initial problems, i.e., parallel copy motion for better coalescing and to avoid edge splitting. Section 7.4 shows the results of our experiments on SPECint benchmark suites. We show in particular that it is better *not* to split edges everywhere, but to move some copies instead. The simplicity of the technique make us believe it could be applied for JIT compilation. We conclude in Section 7.5.

7.1 Parallel Copy Motion

7.1.1 Parallel Copies

As recalled at the beginning of the Chapter, register-to-register parallel copies can be generated by some live-range splitting phase done before or during register allocation. In particular, in most extensions of the linear-scan register allocator, the assignment of a variable between two consecutive basic blocks might be different, which leads, implicitly, to a register-to-register parallel copy *on the edge* between the two basic blocks. Figure 7.1a illustrates such a case: the registers assigned to a and b (in this figure, the notation $a^{(R1)}$ means that variable a is assigned to register R_1 at this point) in basic block B_d are swapped compared to the assignment in B_s, hence, the values contained in R_1 and R_2 must be swapped on the edge from B_s to B_d. On the contrary, variable c is assigned to R_3 on the two basic blocks so the value of R_3 should remain there. The parallel copy is represented in the figure by a graph (whose semantics is given hereafter) along the corresponding edge. A similar situation arises when performing SSA-based register allocation: ϕ-functions are removed after the register assignment phase, which leads, due to the semantics of these functions, to the introduction of register-to-register parallel copies on the edges leading to the ϕ-functions. Figure 7.1b shows an example where R_1 and R_2 must be swapped on the left edge, because the left arguments of the ϕ-functions are in different registers than the variables defined. Also, some less-conventional register allocation frameworks need to insert shuffle code either before, during, or after the allocation. Parallel copies represent shuffle code, encoding data movements in registers so that assignments in different program regions match. Examples of such frameworks include: graph coloring with insertion of split points [23]; combined local, global coloring, and on demand split points, as in priority-based graph coloring [34]; region-based approaches such as hierarchical graph coloring [31]; or graph-fusion allocators [73].



(a) Linear scan (b) SSA

Figure 7.1: Parallel copies on edges.

In these contexts, a parallel copy means that values must be transferred between registers from one program point to another. For this reason, it is handy to represent, in the parallel copy, the registers that keep their value in place. In other words, we enforce a parallel copy to represent the liveness because all "interesting" values, i.e., those of live variables, are referenced in the parallel copy. A parallel copy can be represented as a graph in which live registers are nodes and directed edges represent the flow of the values [55, 92, 86]. Self-edges are necessary to represent unmodified but live registers. In short, R_i is in the live-in (resp. live-out) set of the parallel copy iff there is an edge leaving (resp. entering) the node R_i in the graph representation. For simplicity, we consider that any register in the graph representation of the parallel copy has at most one entering edge. Otherwise, this would mean that the two source registers carry the same value. In such case, we should modify the code so that it uses only one of the registers at this point.

Finally, we also consider only *reversible parallel copy*. The advantage of this restriction will appear clearly in the next section. Actually, when going out of SSA, it is possible that the removing of ϕ-functions creates duplications in parallel copies, thus breaking the assumption of an reversible parallel copy. This happens for instance if, at the beginning of a basic block, the same variable is used twice as argument, as in $[b \leftarrow \ \phi(a, \ldots); c \leftarrow \ \phi(a, \ldots)]$, or if two arguments have been coalesced and renamed into one variable. In practice, the duplications can be extracted from the parallel copies and placed in the predecessor basic block. But this task may lead to additional spilling and we chose for clarity not to treat this case here. Also, none of the existing linear-scan register allocators would lead to parallel copies with duplications on edges. For SSA-based register allocators, the aforementioned situation can be avoided beforehand by inserting a new variable for each duplication on the predecessor edge. In the example above, this would give a copy $[a' \leftarrow \ a]$ in the predecessor block and the original ϕ-functions would be replaced by $[b \leftarrow \ \phi(a, \ldots); c \leftarrow \ \phi(a', \ldots)]$. This is less constraining than enforcing SSA to be conventional static single assignment (CSSA) [100], but CSSA would do the job [86] too.

A parallel copy can be defined as a transfer function from the registers of its *live_in* set to the registers of its *live_out* set. With the additional constraints above, i.e. the parallel copy is reversible, the transfer function is one-to-one:

Definition 4. A reversible parallel copy $/\!/c$ is a one-to-one mapping from its *live_in* set $\{s_i\}$ to its *live_out* set $\{d_i\}$. We use the notation $/\!/c : (d_1, \ldots, d_n) \leftarrow (s_1, \ldots, s_n)$ where $/\!/c(s_i) = d_i$ and $/\!/c^{-1}(d_i) = s_i$.

The *live_in* and *live_out* sets are subsets of the register set. In Figure 7.1a, the *live_in* and *live_out* sets are both equal to $\{R_1, R_2, R_3\}$. The reversible parallel copy is $/\!/c : (R_2, R_1, R_3) \leftarrow (R_1, R_2, R_3)$. An equivalent sequence of move instructions is $[R_x \leftarrow R_1; R_1 \leftarrow R_2; R_2 \leftarrow R_x;]$ if R_x is a register such that $R_x \notin$ *live_out*. For $R_i \notin$ *live_in*, we abusively write $/\!/c(R_i) = \perp$ and, for $R_i \notin$ *live_out*, $/\!/c^{-1}(R_i) = \perp$.

7.1.2 Moving a Parallel Copy Out of an Edge

It is *usually* obvious to move a parallel copy out of a non-critical edge. It can indeed be placed at the bottom (resp. top) of the source (resp. destination) block, if this block has only one successor (resp. predecessor). The term "usually" refers to the register pressure problem exposed for these moves in Section 2.3.3.2. When this problem occurs, we treat the related edges the same way as critical edges. For critical edges, these moves are not directly possible. Hence, the parallel copy would then be executed on other undesired paths. However, it is possible to *compensate* the effect of a reversible parallel copy on other edges.

This is similar to the idea introduced by Fisher [44] for trace scheduling, later called "compensation code", but it concerns general code and deals only with duplicating the code when moving instructions above a join point or below a split point. According to Freudenberger et al. [47], code could be inserted to undo any effects on "off-trace paths". It is not done in practice because, even if it would be possible for simple register operations, it is too complex for general operations. We present in this section a way to "undo" the effects caused by reversible parallel copies.

When trying to move a reversible parallel copy away from a critical edge $E : B_s \rightarrow B_d$, there are two possibilities: either move it *down*, i.e., to the top of B_d, or move it *up*, i.e., to the bottom of B_s, as done in Figure 7.2b. As illustrated by this example, when moving a parallel copy up, it might be expanded to reflect the change of liveness between the critical edge and the end of the predecessor basic block. In our example, a possibility is to make the reversible parallel copy grow with a self edge on R_4 and an edge from R_3 to save its value in R_1. Indeed otherwise, the transfer from R_2 to R_3 would overwrite the value of a live variable, stored in R_3 and needed in B'_d.

Once the parallel copy has been moved up, its effect should be compensated on the other outgoing edges. The compensation is roughly the reverse of the parallel copy. This explains why we restricted initial parallel copies on edges to be reversible. In Figure 7.2b, the values of R_2 and R_3, which are alive on B'_d must be restored.

This example shows that a reversible parallel copy can be moved out of a critical edge, at the price of some compensation code, expressed as a parallel copy too. This parallel copy can possibly be reduced or even removed by further parallel copy motion inside basic blocks. Also, the copies inside a block can be scheduled with the other instructions of the block. This is true in the example for both the parallel copy moved up in the block B_s and the compensation parallel copy moved down in the block B'_d. However, we need a model that takes into account the cost of the critical edge splitting and the cost of the compensation code so as to help the compiler make a decision between moving the parallel copy as explained above or leaving it on the edge and splitting the edge. The precise mechanism to perform this transformation correctly is

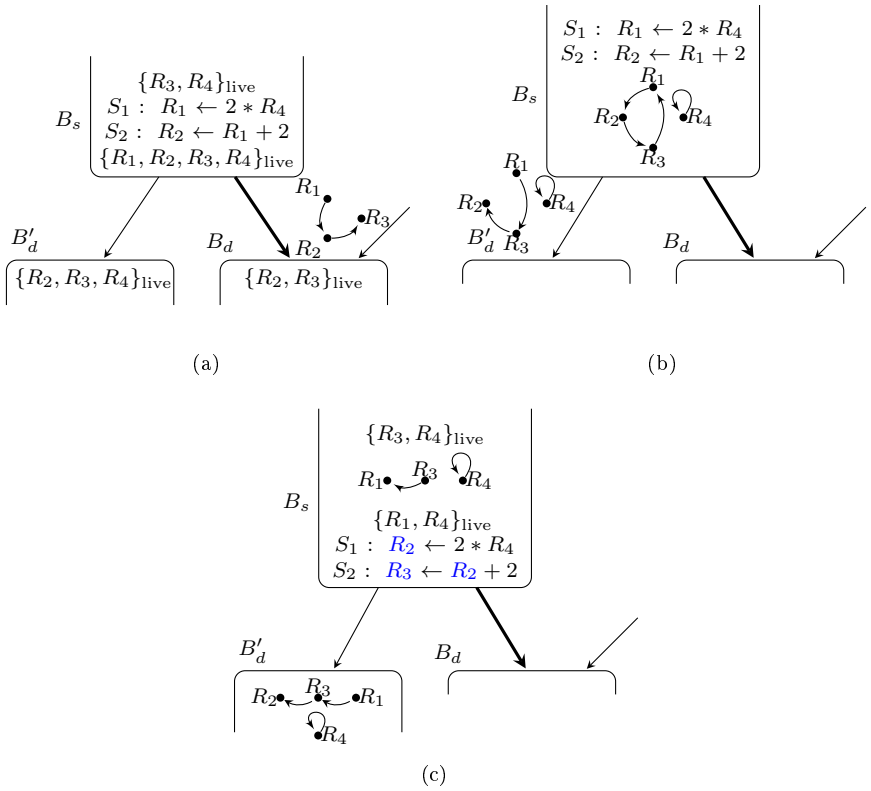


Figure 7.2: On a critical edge (a), parallel copies can be moved if compensated; (b) the parallel copy is augmented to include the liveness of the top basic block, and is compensated on the other leaving edge; (c) the permutation is moved higher in the basic block and its size may shrink (here it does), the compensation code is put at the beginning of basic block B'_d.

explained in Section 7.2 using the notion of permutation motion.

7.1.3 Parallel Copy Motion Inside Basic Blocks

Consider the example of Figure 7.2b again. Because of the presence of data dependencies on R_1 and R_2, the parallel copy at the end of B_S cannot be scheduled before S_2 using standard techniques. Still, it is possible to move a parallel copy inside a basic block. The trick is to consider the parallel copy as a reassignment function and not as a general instruction. This is of course possible only by reassigning operands of "traversed" instructions. Figure 7.2c shows an example where, after having moved a parallel copy up from an edge, the copy is further moved inside the basic block. The operands have been reassigned accordingly. Here, the resulting parallel copy is smaller after being moved up because R_1 and R_2 are not live before, thus their values do not need to be transferred.

The details for performing this transformation will use the *permutation motion* and *region recoloring* concepts. As illustrated by this example, one of the

benefits of moving a reversible parallel copy inside a basic block is that its size may shrink down because the liveness changes. Another potential advantage of this technique, not developed in this chapter, is the ability to place part of the reversible parallel copies on empty slots of a scheduling.

One restriction to the motion inside basic blocks concerns registers constraints. Indeed, some instruction operands cannot be reassigned, for example for function calls. So, unless $/\!\!/c(R_i) = R_i$ for all constraints of an instruction S, $/\!\!/c$ cannot be moved beyond S as it is. Still, it does not mean that we are blocked. It is in fact possible to decompose $/\!\!/c$ into $/\!\!/c' \circ /\!\!/c_{id}$ where $/\!\!/c_{id}$ is the identity for all register constraints of S. Then, $/\!\!/c'$ stays on its side of S while $/\!\!/c_{id}$ can be moved further.

7.2 Permutation Motion and Region Recoloring

To take liveness into account when moving reversible parallel copies, we propose a solution based on *permutations*.

Definition 5. A *permutation* is a one-to-one mapping from the whole set of registers to the whole set of registers.

As seen previously, moving a reversible parallel copy should be done carefully because of liveness. A permutation is a transfer function that does not have to cope with liveness, as it concerns all registers. Because of that, it is much easier to move. The idea here is to extend a reversible parallel copy into a permutation (we call this operation *expansion*), then to move the permutation, and finally to transform the permutation back to a reversible parallel copy (we call this operation *projection*).

7.2.1 Reversible Parallel Copies & Permutations

A permutation π at a program point has the effect of moving each register R_i into $\pi(R_i)$. However, only live registers need to be considered as other registers contain useless values. We can then define a (reversible) parallel copy $\texttt{Project}(\pi)$, the projection of π, as the restriction of π to the live registers. In other words, $\texttt{Project}(\pi)$ is a one-to-one mapping from $\mathit{live\_in}(\pi)$ to $\mathit{live\_out}(\pi) = \pi(\mathit{live\_in}(\pi))$, and such that, for all $R_i \in \mathit{live\_in}(\pi)$, $\texttt{Project}(\pi)(R_i) = \pi(R_i)$. In the graph representation, all edges leaving registers that do not contain any live-in value of the permutation can be safely removed. All remaining edges move data of a live variable and hence must remain in the projected permutation. For convenience if live is the live-in set (resp. live-out set) of π, the projection of π is denoted as $\texttt{Project}_{in}(\pi, \mathit{live})$ (resp. $\texttt{Project}_{out}(\pi, \mathit{live})$). Of course $\texttt{Project}_{in}(\pi, \mathit{live}) = \texttt{Project}_{out}(\pi, \pi(\mathit{live}))$. The projection mechanism is illustrated in Figure 7.2c. Projected before statement S_1, the permutation $\pi : (R_2, R_3, R_1, R_4) \leftarrow (R_1, R_2, R_3, R_4)$ must match its live-in set $\{R_3, R_4\}$: $\texttt{Project}_{in}(\pi, \{R_3, R_4\})$ is $(R_1, R_4) \leftarrow (R_3, R_4)$.

Expanding a reversible parallel copy amounts to find a permutation whose projection is the initial reversible parallel copy. First, the $\mathit{live\_in}$ set must be augmented to be the whole set of registers. Second, since a permutation contains only cycles, the chains of the reversible parallel copy must be closed to

Algorithm 12 Expands given parallel copy into a permutation. The permutation is available as the result of this function.

```
1: function EXPAND(Parallel copy ⫽c)
2:     π ← ⫽c // Make π a copy of ⫽c.
3:     for R_i ∈ Registers do
4:         if π^{-1}(R_i) = ⊥ then
5:             current ← R_i
6:             while π(current) ≠ ⊥ do
7:                 current ← π(current)
8:             π(current) ← R_i // Close the chain to form a cycle
9:     return π
```

form simple cycles. There are more than one way to expand a parallel copy. A possibility is to proceed as in the pseudo-code in Algorithm 12.

For every register that still has no predecessor (Line 4), i.e., every beginning of a chain, the loop Line 6 finds the register at the end of the chain. It then connects this register to the first one so as to form a cycle. Free registers are made cycles of length one (self-loop) by this process. This way, π is the identity for as many registers as possible. Another possibility is to turn all chains into a unique cycle so that it can be "sequentialized" [13] with a minimum number of swaps. Note that the algorithm in [13] can be used to sequentialize any reversible parallel copy using the minimum number of copies.

7.2.2 Region Recoloring

We can define the permutation motion mechanism in a more formal way: for any program region, it is possible to add a permutation π at every entry point of the region, to add its inverse π^{-1} at every exit point of the region, and to reassign every operand in the region according to π: textually replace in the region each occurrence of R_i by $\pi(R_i)$. However, there are still limitations to this, as for the motion of parallel copies in a basic block described earlier: some instructions have register naming constraints, e.g., arguments of a `call`, that cannot be recolored. So, unless $\pi(R_i) = R_i$ for all such constraints, these instructions cannot be part of such a region.

We call this alternative view of permutation motion *region recoloring*, since the variables of the regions get reassigned to different registers. Using this formalism, it is easy to understand how to move a permutation in a basic block, and more generally how the whole parallel copy motion works. On Figure 7.3, the reversible parallel copy $/\!/c$ will be moved up into basic block B_s by recoloring the gray region with an expansion π of $/\!/c$: on the right edge, the composition of `Project`(π) followed by $/\!/c$ simplifies to the identity.

Let us illustrates the process on the example of Figure 7.2 with 4 registers R_1 to R_4 and the same region recoloring as in Figure 7.3. A possible expansion of the reversible parallel copy $(R_2, R_3) \leftarrow (R_1, R_2)$ is to extend it with $\pi(R_3, R_4) = (R_1, R_4)$, i.e., $\pi : (R_2, R_3, R_1, R_4) \leftarrow (R_1, R_2, R_3, R_4)$. The projection of π at the top of B_s is $(R_1, R_4) \leftarrow (R_3, R_4)$ as the initial live-in of the region $\{R_3, R_4\}$ must match the live-in of the reversible parallel copy. The projection of π^{-1} on B'_d is $(R_2, R_3, R_4) \leftarrow (R_3, R_1, R_4)$ as the initial live-out of the region

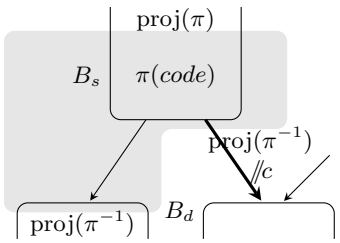


Figure 7.3: Region recoloring, starting with $/\!/c$ on the critical edge.

$\{R_2, R_3, R_4\}$ must match the live-out of the reversible parallel copy. Within the region, R_1 is replaced by $\pi(R_1) = R_2$, R_2 is replaced by $\pi(R_2) = R_3$, there is no occurrence of R_3, and R_4 is unchanged.

To conclude, while trying to move directly reversible parallel copies seems awkward and mind twisting, the detour through permutation motion and region recoloring shows that parallel copy motion is, in fact, not a difficult task to perform. The last task is then to *sequentialize* the parallel copies using actual instructions of the target architecture. This is a standard operation, see for example [13]. The only critical case is when the parallel copy permutes all registers, in which case a swap mechanism is needed.

7.3 Applications

We now detail some applications of parallel copy motion.

7.3.1 Removing Parallel Copies from Critical Edges

The problem with parallel copies on edges is that there is no basic block there. So, in order to actually add code, such an edge must be split and a new basic block must be created to hold the instructions. However, as mentioned in the beginning of the Chapter, there is a folk assumption that splitting edges is a bad idea. The main reasons are both performance reasons (possible additional jump instruction, prevents the use of hardware loops, interaction with basic block scheduling) and functional reasons (compilers may forbid some edges to be split).

We now show how to optimize the removal of parallel copies out of control flow edges. Section 7.3.1.1 gives a heuristic based on a local cost to decide if an edge should be split or if the parallel copy it contains should be moved. A simple propagation mechanism along critical edges is proposed. This mechanism can fail if parallel copies are moved out of an unsplittable edge whose neighboring edges are also unsplittable. This case is addressed in Section 7.3.1.2.

7.3.1.1 A local heuristic

The input of the heuristic is a control flow graph (CFG) with a reversible parallel copy, possibly the identity, on each control flow edge and at the top and bottom of each basic block. The principle of the heuristic is to deal first with edges that cannot be split, and then to deal with the others in decreasing order of

frequency.[1] For each edge in a sorted worklist (initialized with all edges with a parallel copy different than the identity), the heuristic evaluates the impact of parallel copy motion (moving it up, moving it down) by computing a local gain (possibly negative) compared to the solution that keeps the parallel copy on the edge, i.e., compared to edge splitting. Then, the heuristic selects the best feasible solution, applies the modifications, and removes the edge from the worklist. When the content of another edge is modified (because the parallel copy was moved and compensated as explained in Section 7.2), it is added (if not already) in the worklist unless its new parallel copy (the initial copy composed with the compensation) is the identity. The heuristic continues until the worklist gets empty, i.e., it stops when no reversible parallel copy motion leads to a positive gain. (The heuristic terminates as the cost of moves strictly decreases at each step; another cheaper solution is to prevent compensation on edges already examined.) Of course, staying on the edge is forbidden for non-splittable edges. Likewise, a copy motion is not feasible if it produces a parallel copy, different than the identity, on a non-splittable edge. If no choice is feasible, the heuristic fails. This case is discussed in Section 7.3.1.2.

To evaluate the gain, the heuristic should simulate the motion and the compensation on neighboring edges using a performance model. To illustrate the heuristic, let us describe a toy model for a very-long instruction word (VLIW) architecture with 4 issues:

- The cost of an instruction in a basic block B with frequency W_B is approximated to $\widehat{\text{inst}} = \frac{1}{4} \times W_B$.
- The cost of splitting an edge E depends on the linearization of the basic blocks in memory. Inserting some code between two basic blocks placed consecutively in memory can be done for free. However, if the edge corresponds to an actual jump, a new basic block has to be created. The initial jump should point to this new basic block, which itself ends up with an unconditional jump to the target of E. In this case, if E has frequency W_E, the cost of splitting is the cost of a jump, $\widehat{\text{inst}}$, plus the branch penalty W_E, thus $\widehat{\text{split}} = \frac{5}{4} \times W_E$.
- The number of instructions (copy or swap), $\|/\!/c\|$, necessary to sequentialize a parallel copy $/\!/c$ as described in [13].
- We slightly favor the placement of copies in blocks (as they can be scheduled with other instructions), with $\widehat{/\!/c} = \frac{\|/\!/c\|}{4} \times W_B$, than on edges, in which case we let $\widehat{/\!/c} = \left\lceil \frac{\|/\!/c\|}{4} \right\rceil \times W_E$.

Of course this model is far from being perfect, but the effect of further optimizations (e.g., post-pass scheduler), in addition to the approximation made on edge frequency, makes it difficult to model more precisely. What we need is just a model to drive the heuristic in the right direction. Consider as an example the code of Figure 7.5(a). If we leave the parallel copy in place, the local cost will be evaluated as $\left\lceil \frac{1}{4} \right\rceil \times W_{(AB,B)} + \frac{5}{4} \times W_{(AB,B)}$. If we move it down, the cost will be evaluated as $\frac{1}{4} \times W_B + \left\lceil \frac{1}{4} \right\rceil \times W_{(BC,B)} + \frac{5}{4} \times W_{(BC,B)}$. If we move it up, the cost will be evaluated as $\frac{2}{4} \times W_{AB} + \left\lceil \frac{1}{4} \right\rceil \times W_{(AB,A)} + \frac{5}{4} \times W_{(AB,A)}$. Suppose that moving it down leads to a positive gain. At this point, there should be

[1] This frequency can be obtained with any frequency-estimation algorithm or by simple static considerations on the nesting of loops.

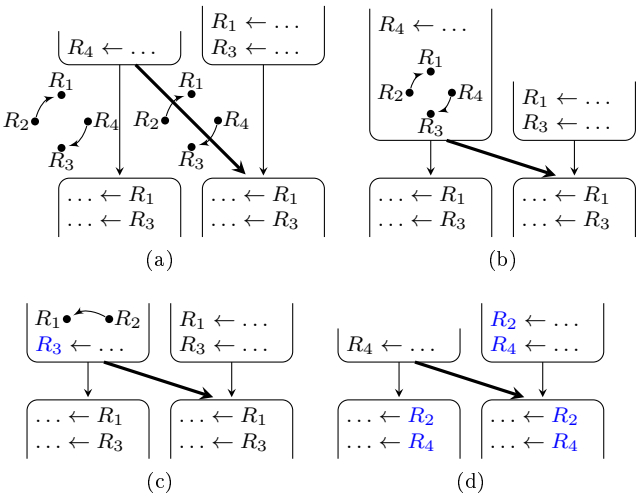


Figure 7.4: Local heuristic and motion in basic blocks. (a) Initial code, 4 moves; (b) Local heuristic, 2 moves; (c) Local heuristic followed by parallel copy motion in basic block, 1 move; (d) All together, no move.

$(R_2) \leftarrow (R_1)$ on the edge (BC, B) and $(R_1) \leftarrow (R_2)$ at the beginning of basic block B. The content of (BC, B) is modified with a non-trivial parallel copy, so (BC, B) is added to the worklist.

The heuristic itself is not local, as copies can move, progressively, further than to neighboring edges. But the decision to move down, to move it up, or to split the edge, is made by a local computation of gain. Algorithm 13 describes this in pseudo-code.

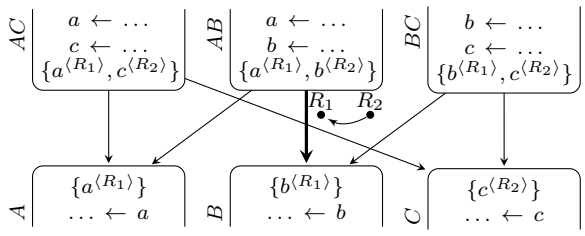
Figure 7.4b illustrates its principles, assuming that R_2 and R_4 are not live beyond the control-flow edges. Here, the local heuristic considers the parallel copy on the critical edge first and computes the cost of leaving it on the edge. It then computes the cost of moving it down. This would produce a compensation on the right edge with two copies and two other copies in the destination basic block. It finally computes the cost of moving the parallel copy up. This would produce a compensation on the left edge, which, composed with the parallel copy already in place, gives the identity, plus two copies in the source basic block. The best local choice is the latter, moving the parallel copy up, as depicted in Figure 7.4b.

7.3.1.2 Parallel Copy Motion Might Be Stuck

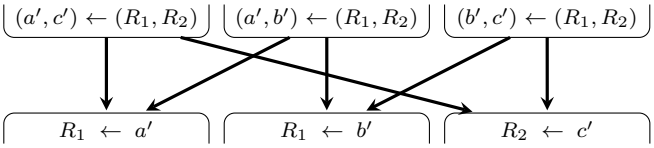
Suppose that, in Figure 7.5(a), the edges $AB \to A$, $AB \to B$, and $BC \to B$ are marked as unsplittable. If the first considered edge is the bold edge (from AB to B), the heuristic fails as it cannot split the edge and it cannot move the copy up (resp. down), as a compensation would be needed on the unsplittable edge $AB \to A$ (resp. $BC \to B$). In such a case, a recursive heuristic that tries to move the compensation further is necessary. For example, the reversible parallel copy can be moved down on B; its compensation on $BC \to A$, can then be moved up on BC; the compensation of this motion on $BC \to C$ can be moved

Algorithm 13 Evalutes the gain of moving in the given direction the reversible parallel copy carried by the given edge if possible. Performs the changes implied by the considered moves if *simulate* flag equals FALSE. Returns whether the given direction is a valid move and the gain of such a move.

```
1: function LOCAL-HEURISTIC(Edge e, Direction direction, Boolean simulate)
2:     //c ← e.//c
3:     gain ← 0
4:     π ← EXPAND(//c)
5:     if simulate = TRUE then
6:         save current state
7:     // Move parallel copy in the related basic block
8:     if direction =↑ then
9:         edges ← B_s.leaveEdges                          // edges with compensation
10:        gain ← B_s.//c_bottom.cost                      // initial cost at end of block
11:        // Move the parallel copy up
12:        //c ← Project_in(π, B_s.liveOutSet)
13:        B_s.//c_bottom ← //c ∘ B_s.//c_bottom            // compose to get new copy
14:        gain ← gain − B_s.//c_bottom.cost                // subtract new cost
15:    else if direction =↓ then
16:        edges ← B_d.enterEdges                          // edges with compensation
17:        gain ← B_d.//c_top.cost                         // initial cost at start of block
18:        // Move the parallel copy down
19:        //c ← Project_out(π, B_d.liveInSet)
20:        B_d.//c_top ← B_d.//c_top ∘ //c                  // compose to get new copy
21:        gain ← gain − B_d.//c_top.cost                   // subtract new cost
22:    else
23:        // We want to split e
24:        if simulate = FALSE and e.isSpittable then e.split = TRUE
25:        return e.isSpittable, gain
26:    for e_i ∈ edges do
27:        gain ← gain + e_i.//c.cost                      // initial cost on e_i
28:        // Apply compensation on the edge.
29:        if direction =↑ then
30:            // Compensation's liveout must match the livein of e_i.//c
31:            //c_tmp ← Project_out(π^-1, e_i.//c.liveInSet)
32:            e_i.//c ← e_i.//c ∘ //c_tmp                  // compose to get new copy
33:        else
34:            // Compensation's livein must match the liveout of e_i.//c
35:            //c_tmp ← Project_in(π^-1, e_i.//c.liveOutSet)
36:            e_i.//c ← //c_tmp ∘ e_i.//c                  // compose to get new copy
           gain ← gain − e_i.//c.cost                       // subtract new cost
37:    if simulate = TRUE then
38:        restore current state
39:    return TRUE, gain
```



(a)



(b)

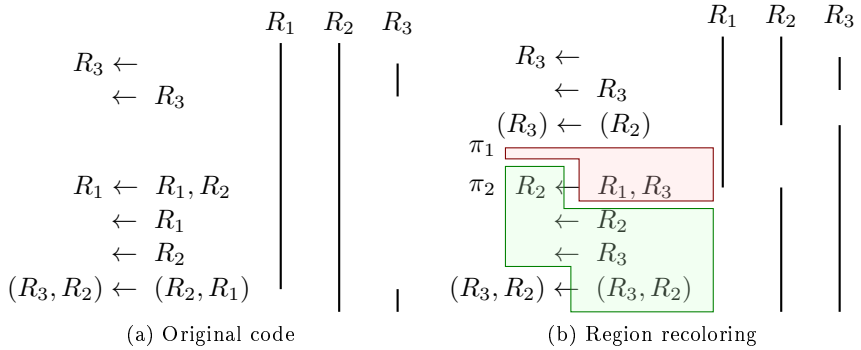
Figure 7.5: Complex multiplexing region. (a) The local heuristic can be stuck; (b) an ultimate solution involves Chaitin-like graph coloring.

down on C, and so on until a splittable edge is reached. Even though, in the extreme case where all edges of the figure are unsplittable, such a propagation process will loop and will not manage to eliminate the parallel copy.

When the parallel copy motion is stuck, the ultimate solution is to consider the whole region (that we call multiplexing region) formed by the maximal set of connected edges and to view the problem as a standard (NP-complete in general) graph coloring problem. This situation is depicted in Figure 7.5(b). Live-ranges are split at the frontier of the multiplexing region using parallel copies between registers and variables (a', b', and c'). The interference graph is a 3-clique. If available, 3 different registers can be assigned to the 3 variables, otherwise, some spilling is required. Here, whatever the number of available registers, the parallel copy motion is stuck. We point out that, although theoretically possible, we never encountered such a case requiring a global graph-coloring approach in all our benchmarks: the local heuristic always succeeded.

7.3.2 Shrinking Parallel Copies in a Basic Block

As mentioned earlier, moving parallel copies out of control flow edges is not the best we can do. We can still use the parallel copy motion mechanism to move the parallel copy further in the block, either up if it comes from an outgoing edge of the block, or down if it comes from an incoming edge. In a fully-scheduled code, one could look for an empty slot to hide the parallel copy. But even without knowing the schedule, the parallel copy motion can be interesting. Indeed, depending where the parallel copy is placed, the number of moves it implies may vary as the parallel copy is projected on the live variables. For example, the extreme situation is when no variables are live at some program point: placing the parallel copy there means simply recoloring the whole region below (if the copy is moved up), with no move: the parallel copy vanishes. Another



(a) Original code (b) Region recoloring

Figure 7.6: Composition of several recoloring regions allows to change the considered permutation within an instruction. This is possible when the composition projects to identity within the instruction. In that example, $\pi_1 = (R_1, R_2, R_3) \leftarrow (R_1, R_3, R_2)$ and $\pi_2 = (R_1, R_2, R_3) \leftarrow (R_3, R_1, R_2)$. Thus, the projection of $\pi_1^{-1} \circ \pi_2$ at the joint point of the region equals the identity. Likewise, $\pi_2^{-1} \circ /\!\!/_{c_{id}}$ projects to identity at the end of the second region

side effect is that some remaining moves in the code (with no duplication) can also be absorbed along the way.

The region recoloring mechanism can cover arbitrary contiguous region. In particular, it can consider a region starting (resp. ending) between the definitions and uses of an instruction. At those spots, the number of live variables may be even smaller than regular program point. Indeed, if a recoloring region starts (resp. ends) at one of these point, the last used (resp. defined) variables are not considered in the recolored region. However, the parallel copy resulting from the projection of the liveness at that point cannot be inserted here. This is not a problem as recoloring region can be composed at will[2]. Thus, another recoloring region can be used to move this parallel copy copy at a valid program point. In this case, the composition of the projection of the recoloring region just have to end up in the identity on the invalid point. This is illustrated by Figure 7.6. In other words, this technique allows to apply different permutations to different regions. The challenge is then to choose the best permutations and the best regions.

Using this idea, we developed a heuristic for parallel copy motion within a basic block. For that heuristic, we fixed the regions we considered. Algorithm 14 gives the pseudo-code for the motion up (direction $\uparrow$) from the bottom of a basic block. The input of the heuristic is a basic block with a reversible parallel copy on its top and on its bottom. These parallel copies may have been composed with the local heuristic of Section 7.3.1.1. We proceed in two phases. First, we simulate the motion of the parallel copy in the basic block and we record where the parallel copy is the cheapest. In a second time, we do the motion to the previously-selected position. For both the simulation and the motion, we proceed instruction after instruction. For each program point (before an instruction, inside an instruction between the use and the def operands, after an instruction), we project and re-expand the permutation that we are moving

[2] In particular, they can overlap.

up. This has the effect of potentially shrinking the permutation. If we cannot traverse an instruction due to coloring constraints, we stop the process (although we could split the parallel copy, as explained in Section 7.1.3). Moving a parallel copy down in a basic block is similar to the pseudo code of Algorithm 14. The only subtlety is to mark the last uses, i.e., the uses of variables that are not live-out of the instruction so as to update liveness during the traversal.

Algorithm 14 Evaluates the best position to project in the given basic block the parallel copy at the end of that block. Applies the changes if *simulate* is FALSE. Given *position* indicates where to stop the motion if *simulate* = FALSE. Returns the minimum cost for that motion and the related position.

```
1: function MOTION-UP-FROM-BOTTOM((BasicBlock block, Boolean simulate, Operation position))
2:     minPosition ← block.bottom
3:     minCost ← block.//c_bottom.cost                    // Sequentilization cost
4:     π ← block.//c_bottom
5:     Live ← block.//c_bottom.liveInSet
6:     for op ∈ block's operations in reverse order do
7:         if simulate = FALSE and current position = position then
8:             exit loop
9:     if π can traverse op then
10:        for result in op's results do
11:            live ← live − result
12:            if simulate = FALSE then
13:                substitute result by π(result)
14:        π ←EXPAND(Project_in(π, live))
15:        for arg in op's arguments do
16:            live ← live ∪ arg
17:            if simulate = FALSE then
18:                substitute arg by π(arg)
19:        if Project_in(π, live).cost < minCost then
20:            minPosition ←before op
21:            minCost ← Project_in(π, live).cost
22:        else
23:            exit loop                                  // Happens only when simulating.
24:    if simulate = FALSE then
25:        SEQUENTIALIZE(position, Project_in(π, live))
26:        // Reset block's parallel copy with the identity on live out set
27:        block.//c_bottom ← Id(block.liveOutSet)
28:    return minCost ∗ block.frequency, minPosition
```

Figure 7.2c illustrates a motion in a basic block after the local heuristic. One copy remains before the definitions of R_2 and R_3. Here, the parallel copy motion is performed after the decision made to move copies out of edges. But, we can integrate the possibility of moving parallel copies inside basic blocks in the cost function given in Section 7.3.1.1. With no change to the local heuristic, we can achieve better performance. For example, Figure 7.4d shows how the

new cost function modifies the algorithm decision. Now the parallel copy on the critical edge is moved down, which produces a compensation on the right edge. The resulting parallel copies shrink to identity. The same happens for the parallel copy on the left edge. In this example, all copies can be removed thanks to parallel copy motion.

7.4 Experiments

We implemented parallel copy motion on top of the linear assembly optimizer (LAO), a production compiler for a commercial target architecture from STMicroelectronics. For these experiments, we used the LAO code generator as a static compiler for C code, connected to the code generator of the open source version of the SGI Pro64 compiler [49] (OPEN64). Using aggressive optimization level `-O3`, the OPEN64 compiler generates the code up to register allocation , at which point the LAO performs register allocation and implements parallel copy motion. The compilation is completed by the OPEN64, which does post-allocation optimizations and emits executables. We did not make experiments using JIT configuration as this setting was not available to us.

Our target processor is a commercial media-processing embedded VLIW architecture from the Lx [42] family of processors issuing up to 4 instructions per cycle over 6 functional units consisting of 1 load-store unit, 1 branch unit, and 4 arithmetic units. We made our experimentation on the C subset of Spec2000 integer benchmarks and benchmarks from STMicroelectronics (KERNELS). The *eon* C++ benchmark is not included due to the limited support for C++ in our code generator version. Also the *crafty* benchmark is excluded due to a yet unsolved functional problem with our compiler configuration. The KERNELS are a set of computation-intensive kernels like fft, jpeg, and quicksort algorithms, supposed to be representative of embedded media applications as found in firmware code such as audio, video codecs, or image processing.

For this study, we compared the parallel copy motion algorithm against a split-everywhere strategy for critical edges. Both are run after our tree-scan register allocator, see Chapter 5, with biased coloring on ϕ-functions and moves. We used Hack's heuristic [55] for the spilling phase. The performance of the allocator with the split-everywhere strategy is comparable to an extended linear scan coloring heuristic [76]. The colored ϕ-functions nodes are left in the program, which is thus in colored-SSA form, and the ϕ-functions are then interpreted as parallel copies on the edges before the parallel copy motion heuristics are run. We evaluated the parallel copy motion heuristics under three modes: motion on edges alone (mode *edge motion*), motion on edges followed by motion inside basic blocks (mode *block motion*), and motion on edges where the motion inside basic blocks is taken into account in the cost model (mode *all*). When not specified otherwise, *edge motion* is thus done without motion in blocks. The split-everywhere strategy only splits critical edges when some move operation remain. Other edges are not split as their parallel copies can always be moved, with no compensation, to their source or destination basic block.

We show different kind of results, based on the cost model with static or profile-based basic block frequency estimations.

First, we give some figures corresponding to the cost model used in the heuristics so that we can verify its efficiency out of the execution context. At

Benchmark	#edges	Benchmark	#edges
164.gzip	0	175.vpr	0
176.gcc	16	181.mcf	0
197.parser	0	253.perlbmk	15
254.gap	3	255.vortex	2
256.bzip2	0	300.twolf	0

Table 7.1: Number of non-splittable edges with moves, hence not solvable without parallel copy motion out of edges.

the end of the compilation process, we measured the number and weight of moves, split edges, branches, etc., using the basic block frequency estimations as provided by the compiler. The weight of moves is the number of moves times the basic block frequency. For basic blocks introduced by edge splitting, we also account for the branch instructions, because they are a consequence of the materialization of the moves. Except, of course, when the split edge does not generate a branch (we call such edges *false critical edges*). The frequency estimations come from static heuristics derived from [5] for the KERNELS and from edge profiling for Spec2000.

Second, we give the actual performance by comparing the cycle counts of benchmarks runs. As for the model figures introduced above, the performance measurements were run without profiling feedback on the KERNELS benchmarks and with profiling feedback on the Spec2000 benchmarks. For the latter, the performance were measured on the same data set as for the profiling feedback run as we want to illustrate the potential of the parallel copy motion technique on an accurate cost model.

7.4.1 The Impact of Copy Motion Out of Edges

7.4.1.1 Non-Splittable Edges

We found 36 critical non-splittable edges with implicit moves after SSA-based register allocation in Spec2000 as reported in Table 7.1. They come from 4 different applications: gcc, perlbmk, gap, and vortex. Given the coloring produced by the register allocator, the compilation of these 4 applications could not be completed without parallel copy motion. With *edge motion*, we were able to move all parallel copies out of these "abnormal" edges. Thus, this fairly simple strategy is sufficient to complete the compilation. In particular, this means that multiplexing region with non-splittable edges (such as in Figure 7.5) do not occur, at least in these benchmarks. The KERNELS do not exhibit such edges.

7.4.1.2 Number of Split Edges

We computed how many splits of critical edges were avoided by using the heuristics based on our cost model, i.e., for which it was preferable to move the parallel copies, according to the model. This shows, as one may expect, that the best insertion point for copies is not always on the edge. Table 7.2 presents the number and weight of critical edges that still carry moves at the end of the compilation process, and hence must be split, normalized to a strategy that always chooses to split. This table shows that, on the average, roughly half of the edges (0.48) are still split with *edge motion*. But in terms of weight, split edges are almost

Benchmark	Split everywhere		Edge motion	
	Number	Weighted	Number	Weighted
164.gzip	1	1	0.52	0.00
175.vpr	1	1	0.45	0.03
176.gcc	1	1	0.33	0.03
181.mcf	1	1	0.36	0.00
197.parser	1	1	0.4	0.00
253.perlbmk	1	1	0.69	0.02
254.gap	1	1	0.41	0.02
255.vortex	1	1	0.73	0.00
256.bzip2	1	1	0.48	0.02
300.twolf	1	1	0.56	0.01
G.Mean (10)	1	1	0.48	0.004

Table 7.2: Number of critical edges split after parallel copy motion, normalized to a split everywhere strategy.

Benchmark	Edge motion		All
	w/o bl motion	w/ bl motion	
164.gzip	+1%	+1%	+2%
175.vpr	+1%	+1%	+1%
181.mcf	+2%	+3%	+4%
197.parser	+3%	+4%	+4%
256.bzip2	+2%	+5%	+5%
300.twolf	0%	0%	0%
G.Mean (6)	+1%	+3%	+3%

Table 7.3: Execution speedup of the parallel copy motion heuristics compared to a split-everywhere strategy for the Spec2000 subset, obtained with profiling feedback activated. (Bigger is better)

never executed (0.004). This is because our model accounts for the additional branch inserted and for the low resource usage on multiple-issues architectures when an edge is split. In particular, it reflects the fact that a small sequence of operations, as generated by parallel copies, is more costly in a dedicated basic block than on a basic block where it may be scheduled with other operations.

7.4.1.3 Performance Impact

We evaluated the performance improvements of our method, for the insertion of parallel copies, when compiling at an aggressive optimization level in our static compiler tool-chain. The evaluation was done on the two sets of benchmarks previously presented, except for the four Spec2000 benchmarks that cannot be compiled with the split-everywhere strategy, due to non-splittable edges. For the Spec2000, the simple local heuristic *edge motion* leads to an average speedup of 1% with no loss (see Table 7.3, column edge motion w/o block motion). Three benchmarks (mfc, parser and bzip2) are improved by up to 2% with this simple heuristic. These performance results confirm that a split-everywhere strategy not only fails in the case of non-splittable edges, but is also inefficient compared to an heuristic based on a cost model to decide if edge splitting is profitable.

Looking at the KERNELS, we also got an average speedup of 2% and no loss (see Table 7.4, column edge motion w/o block motion). Over the 50 benchmarks, 26 are actually improved. Over these 26 benchmarks, 9 show a performance speedup of at least 5%. Note that for these tests, *we do not use profiling-feedback information*, thus even with frequencies estimation, we achieved good results, at least on computation-intensive benchmarks.

7.4.2 The Impact of Copy Motion in Basic Blocks

7.4.2.1 Weight of Moves

In order to evaluate the impact of parallel copy motion inside basic blocks, we compared the weight of move operations with *edge motion* and with *edge motion* followed by the heuristic for motion in basic blocks (*block motion*). Table 7.5 gives the results of these experiments on Spec2000. On average, *edge motion* followed by *block motion* divided the weight of move operations against *edge motion* by a factor 1.16 (0.86/0.74) and we observed no loss. For the bzip2 benchmark, it reduces this weight by a factor of 1.81. To be noted in the second column of the table, when compared to the split-everywhere heuristic where moves are inserted on critical edges, the weight of moves is always reduced by any of the copy motion heuristic. For the KERNELS, *block motion* has nearly no effects when we run the same experiment. At the basic block scope, there are fewer opportunities for reduction of the size of parallel copies in these benchmarks compared to Spec2000. Indeed, we observed that the length of the basic blocks is generally smaller in these benchmarks and that there are fewer call sites (a call site puts additional constraints on coloring and thus favors parallel copy motion).

7.4.2.2 Performance Impact

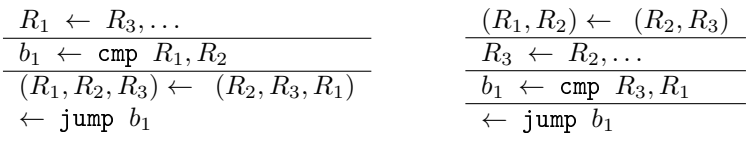
Finally, we measured the performance impact of motion in basic blocks in addition to the weight reduction of move operations. Table 7.3 (the two columns edge motion w/o and with block motion) shows the comparison in cycles on Spec2000 between the motion out of edges and the same heuristic followed by the motion in blocks. We see that this heuristic brings, on the average, an improvement of 2% of performance compared to *edge motion*. If we compare these results with the split-everywhere strategy, we get an average speedup of 3%, with an improvement of 5% on bzip2 and 4% on parser. Again, we observed no performance loss. Considering the KERNELS, the results are quite similar to *edge motion*. To be noted, we got a regression of 3% on the lsearch kernel, compared to *edge motion* alone. This regression is the result of a bad interaction between the motion in blocks and the compiler post-scheduling phase. This is a limitation of the cost model that does not account for the availability of resource slots. Thus, while in most cases the cost model is efficient, it may actually augment the schedule length, even when reducing the number of copies, due to a lack of resource at the point of insertion. Figure 7.7 shows such a case. The same observation applies for the euclid kernel. We also observed that *edge motion* alone reduces the weight of moves even if it was not our original motivation. This is because it removes a lot of edge splitting and thus the related branch penalty, which is counted in this weight.

Benchmark	Edge motion		All
	w/o bl motion	w/ bl motion	
BDTI.bkfir	+3%	+3%	+3%
BDTI.control	0%	0%	+3%
BDTI.ssfir	+3%	+3%	+3%
BDTI.vecprod	+5%	+5%	+5%
BDTI.vecsum	+6%	+6%	+6%
BDTI.viterbi	+3%	+3%	+3%
ITI.bitaccess	+2%	+2%	+2%
ITI.ctrlstruct	+2%	+2%	+2%
ITI.logop	+1%	+1%	+1%
ITI.recursive	+1%	+1%	+1%
KERN.bitonic	0%	0%	+7%
KERN.copya	+9%	+9%	+9%
KERN.dotprod	+6%	+6%	+6%
KERN.euclid	+5%	+4%	+3%
KERN.fir8	+1%	+1%	+1%
KERN.fircirc	+1%	+1%	+1%
KERN.latanal	+2%	+2%	+3%
KERN.lsearch	+6%	+3%	+3%
KERN.max	+7%	+7%	+7%
KERN.maxindex	+2%	+2%	+2%
KERN.mergesort	+2%	+2%	+3%
KERN.quicksort	+4%	+4%	+7%
KERN.strtrim	+17%	+17%	+17%
KERN.strwc	+23%	+23%	+23%
KERN.vadd	+4%	+4%	+4%
MUL.fir_int	+1%	+1%	+1%
MUL.jpeg	+1%	+1%	+1%
MUL.ucbqsort	+1%	+1%	+2%
STFD.stanford	0%	0%	+1%
(...plus 21 unchanged...)			
G.Mean (50)	+2%	+2%	+3%

Table 7.4: Benchmark execution speedup of the parallel copy motion heuristics compared to a split-everywhere heuristic for the KERNELS suite, obtained without profiling feedback. (Bigger is better)

Benchmark	Split	Edge motion		All
		w/o bl motion	w/ bl motion	
164.gzip	1	1	1	0.97
175.vpr	1	0.98	0.94	0.93
176.gcc	1	0.59	0.44	0.4
181.mcf	1	0.96	0.95	0.87
197.parser	1	0.59	0.47	0.45
253.perlbmk	1	0.96	0.9	0.85
254.gap	1	0.85	0.75	0.71
255.vortex	1	0.98	0.94	0.93
256.bzip2	1	0.94	0.52	0.52
300.twolf	1	0.93	0.84	0.82
G.Mean (10)	1	0.86	0.74	0.71

Table 7.5: Weighted moves normalized to a split everywhere strategy. (Lower is better)



(a) Before *block motion*: schedule length, 3 cycles

(b) After *block motion*: schedule length, 4 cycles

Figure 7.7: Even if *block motion* heuristic reduces the number of inserted moves, it may degrade the runtime performances. Horizontal bars are the frontiers of the bundles. Each bundle is proceeded in 1 cycle. Before *block motion*, (a), the region has 3 moves, depicted by the parallel copy just in front of the `jump`, but only 3 bundles. After *block motion*, (b), the region has only 2 moves, but they cannot be scheduled with any existing bundles, increasing the schedule length.

7.4.3 All Together

To take advantage of the recoloring ability of motion inside basic blocks, we mentioned in Section 7.3.2 that we can integrate, in the cost model of the local heuristic, the optimized cost of placing a copy, not only at the bottom or top of a block, but also inside it. The goal is to account for the effect of reducing the number of generated copies before making a choice. The benefit of this improved heuristic was illustrated in Figure 7.4d compared to the two-steps heuristic performing motion out of edge, then motion in block as shown in Figure 7.4c. In this section, we present the actual improvements of this cost model on the overall performance of the benchmarks.

Columns *All* in Table 7.3 and Table 7.4 report performances of respectively the Spec2000 and the KERNELS benchmarks. We have, on the average, 3% of improvements for both the benchmarks suites, with no loss compared to splitting the edges for inserting copies. We improve the performance of 5 over 6 benchmarks for Spec2000 and of 29 over 50 benchmarks for the KERNELS. We have 9 benchmarks with more than 5% of improvement in the KERNELS. In particular, 6 of these benchmarks are over 7% of improvement with greatest improvements for strtrim (17%) and strwc (23%).

7.5 Conclusion

We introduced a new technique that we called parallel copy motion, which can be seen as a formalized tool for moving copies around in a control flow graph after register allocation has been performed. The goal is to reduce the global cost that copies induce directly (additional instructions) or indirectly (edge splitting).

While our initial motivation was the motion of copies out of critical edges, this tool has been extended to handle recoloring arbitrary control flow regions containing operations with register constraints. Thanks to the expansion of parallel copies into permutation of colors, the simple and sound theory on permutation motion, and the simple constraints on region boundaries, it is now easy to formalize the parallel copy motion problem, including a cost model and a freedom of motion with different granularities: operation, basic block, and up to a complete region.

There are several possible applications to this technique. So far, we applied it to the problem of destruction of colored SSA, as provided by a decoupled register allocation algorithm over SSA. For this problem, we used the parallel copy motion technique to move copies introduced by ϕ-functions away from critical edges, when it is profitable, or simply when the edge cannot be split, as it is the case for some edges present in compiler code generators for C and C++. We have indicated that the permutation motion can be stuck in the presence of *multiplexing regions*, where all critical edges are non-splittable. In this case, we propose to use classical graph coloring techniques to recolor the multiplexing regions, possibly requiring additional spills. Nevertheless, in practice, the compiler hardly generates such regions (none showed up in our experiments), thus it does not appear to be an issue for performance.

In the context of this colored SSA destruction problem, and for the VLIW architecture for which we are compiling, we obtained performance improvements of 3% on average, for both the C integer subset of Spec2000 and our own benchmarks, compared to the edge-splitting approach generally used. More generally, we have shown not only that critical edge splitting can be completely avoided when necessary, but also that one can benefit from having a cost model to drive the edge splitting decision. In our context, we reduced the number of split critical edges by a factor of two when using a cost model, which demonstrates that edge splitting actually pays off only half of the time on average. Moreover, we got all these improvements with a very simple application of our model. We think that the approach is promising and that we can perform even better. In particular, we identified several items that could complement the current heuristic to achieve better performance: 1. a scheduling model, for instance to take into account empty slots for VLIW architecture or memory latency, 2. the possibility to decompose a permutation anywhere, for instance to go through register constraints or to fulfill empty slots of the scheduler, 3. the extension of the permutation when the liveness is growing: What is the best strategy to complete (expand) a parallel copy into a permutation?

We believe that discovering that parallel copies can be easily moved is a major breakthrough for out-of-SSA translations. Up to now, it was in general considered that placing copies on edges would require to split them, which is not necessarily the best approach. For this reason, people tried to introduce copies directly at the borders of basic blocks since the discovery of SSA, and the first algorithms [37] up to the out-of-SSA translation of Sreedhar et al. [100]

and Boissinot et al. [13]. Recently, the idea of doing register allocation while being still under SSA was developed. The goal is to use the nice properties of SSA for a longer time. However, the drawback is that going out of SSA introduces parallel copies on edges. A recoloring technique was proposed by Hack and Goos [57] to coalesce the copies on these edges, but splitting edges is still necessary whenever the coalescing fails. Last but not least, register allocators used for JIT compilation, mostly variants of linear scans, perform poor coalescing and could benefit from a fast parallel copy motion post-phase. As it is, our method can be applied in a JIT incrementally to improve coloring, since it performs local coloring that can be safely stopped at any time. For instance, one can start with most frequently executed edges and stop as soon as a time limit is reached.

Chapter 8

Elimination of Parallel Copies Using Copy Motion on Data Dependence Graphs

Rise of decoupled register allocators has increased the pressure on the elimination of copies. Indeed, decoupled register allocation is possible at the price of live-ranges splitting, thus more copy instructions. One possible form of live-range splitting for such allocators is given by the static single assignment (SSA) form.

Traditional copy elimination by coalescing using interference graphs (IGs) generally performs well in this setting [3, 32, 26, 51, 81]. However, recent SSA-based spilling and assignment heuristics [21, 22] avoid the costly construction of IGs, to cope with a potential use in a just-in-time (JIT) compiler. It is thus essential to develop new copy elimination techniques. One solution is to bias the register assignment to heuristically assign the same register to copy-related variables [28, 22]. Another possibility is to perform local recoloring [57] after the assignment phase.

A common limitation of existing approaches to copy elimination is that the ordering of the instructions in the program is *not* modified. We present an extension of the parallel copy motion technique, see Chapter 7, that operates on data dependence graphs (DDGs) and eliminates copies by performing local code motion. Our approach is based on parallel copies (see Section 2.1.1) that originate from mismatching register assignments at split points. The parallel copies are represented within a DDG, along with all other operations of a basic block. We then perform *upward* and *downward* code motion of instructions reading or defining a register of a parallel copy respectively. The goal is to render this particular register *dead* before or after the parallel copy, i.e. the register's value is no longer used. Once the register is dead, the parallel copy becomes (partially) useless and can be split or even completely eliminated. The goal is thus to eliminate as many copies in the dependence graph of each basic block as possible. Not all code motions are permissible. It has to be ensured that all data dependencies are preserved and no values are lost. In particular, we have to ensure that no cyclic dependencies are introduced in the DDG, which prevent an ordering of the instructions after copy elimination.

This chapter shows that the data dependence graph can be kept consistent using rather simple and elegant transformation rules that split and eliminate parallel copies until no further simplification is possible or a predefined time limit or threshold is reached.

The remainder of this chapter is organized as follows. We give some background on data dependence graphs in Section 8.1. Next, we will describe our approach to copy elimination on data dependence graphs in Section 8.2. Section 8.3 presents details on experiments conducted using the SPECINT 2000 benchmark suite. Related work is presented in Section 8.4 before concluding in Section 8.5.

8.1 Data Dependence Graphs

The presented algorithms operate on data dependence graphs to eliminate copies. We thus give a brief definition:

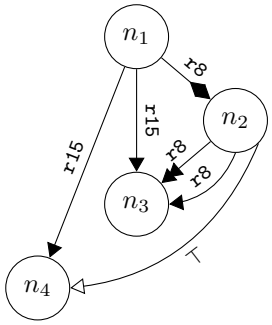
Definition 6. A data dependence graph (DDG) is an acyclic graph $G = (V, E, L)$, where nodes n in V represent instructions and labeled edges (u, v, l) in $E \subseteq V \times V \times L$ dependencies among instructions. We distinguish four kinds of labels in L: (1) true register dependencies $\overleftarrow{r}$, (2) register output dependencies $\overleftrightarrow{r}$, (3) register anti dependencies $\overrightarrow{r}$, where r is a register name, and, finally, (4) other dependencies $\top$.

The definition above follows the standard conventions for dependence graphs in instruction scheduling [101, Ch. 19] and the notion of dependencies used in computer architecture design [82, Ch. 3]. However, we focus on register dependencies only. A true register dependence (also known as, read-after-write dependence) appears in a program whenever one instruction writes to a register and another instruction subsequently reads from the very same register (without any other definition in between). When reordering the instructions of the program, the former instruction always has to appear before the latter in order to preserve the original data flow. These dependencies thus cannot be eliminated. An output register dependence (aka. write-after-write dependence) arises whenever two instructions subsequently write to the same register. Finally, an anti register dependence (aka. write-after-read dependence) arises when on instruction reads a register and a subsequent instruction writes to that register. In contrast to true dependencies, output and anti dependencies stem from *name conflicts*, i.e., the same register is reused to hold the result of otherwise independent computations. These dependencies can sometimes be eliminated by register renaming [82, Ch. 3]. Other forms of dependencies, such as memory dependencies between memory access instructions or control dependencies arising from branch and jump instructions, are not further distinguished, as this is not necessary for our purposes.

Data dependence graphs are best represented using a graphical notation as depicted by Figure 8.1, showing a linear code fragment and its DDG. Throughout this chapter we follow the convention that true dependencies are represented by an arrow with a filled triangle tip (→), output dependencies by an arrow with two overlapping triangles (↠), and anti dependencies by an arrow with a diamond tip (⇾). Other dependencies are represented by an open triangle (⇢).

```
1:  mov r15 = r8;
2:  ldw r8 = 44[r9];
3:  add r8 = r8, r15;
4:  stw 44[r9] = r15;
```

(a) Linear Assembly Code



(b) Data Dependence Graph

Figure 8.1: Assembly code (a) and the corresponding data dependence graph (b).

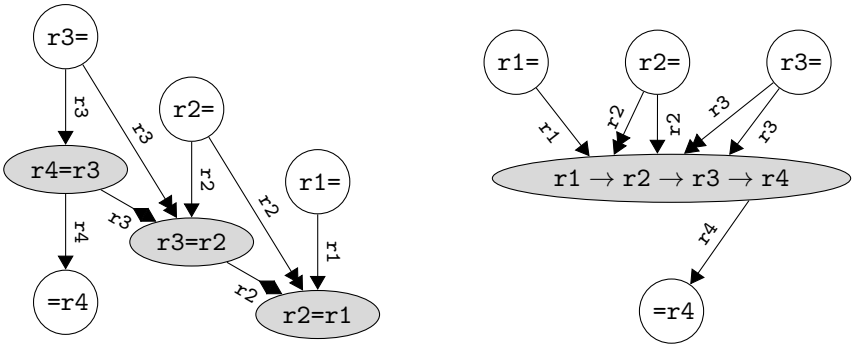
8.1.1 Parallel Copies

We rely on the notion of *parallel copies* to represent, for every split point in the program where live-ranges have been split, the set of mismatching register assignments after register allocation and a set of atomic copy operations to fix-up those mismatches. Under SSA form, for instance, split points are explicitly represented by ϕ-functions. Parallel copies arise where the register allocator was not able to assign the same register to some of the ϕ-functions source and destination operands [58].

At this stage of the compilation, a *Parallel Copy* is a set of register-to-register copies. The most general case of parallel copy is represented by graphs where the nodes have an in-degree of at most 1, i.e., each register is defined at most once. This form of parallel copies allows the *duplication* of register values. However, for the remainder of this work, we consider regular or cyclic copies only, see Section 2.1.1, more complex graphs are assumed to be decomposed beforehand [74, 13][15, Ch. 7.3]. The algorithms presented in the following are, in principle, applicable to more general graph structures under minor modifications.

When parallel copies are represented in a DDG, we implicitly assume a set of individual register-to-register copies that are merged into a single node. The dependencies between those moves and other instructions in the DDG are directed to the merged node accordingly. Representing parallel copies as a single node has the advantage that the resulting DDG is free of cycles, which would arise in the case of cyclic copies.[1] Figure 8.2 shows the two ways of representing a parallel copy in a DDG: once using explicit moves and once using a merged copy node. Note that all dependencies are reflected equally in both versions. in particular, the anti-dependencies among the individual copies are captured by the copy chain `r1` $\rightarrow$ `r2` $\rightarrow$ `r3` $\rightarrow$ `r4` within the parallel copy node.

[1] Cyclic parallel copies could also be represented as a sequence of swap instructions. This would avoid the cycles in the DDG, but would complicate the copy elimination presented later.



(a) DDG with a set of copy nodes (b) DDG with a merged parallel copy node

Figure 8.2: A parallel copy `r1` → `r2` → `r3` → `r4` represented as a set of register-to-register copies (a) and as a single parallel copy (b) in a DDG.

8.1.2 Parallel Copy Motion

Our approach is an extension of the *Parallel Copy Motion* technique presented in Chapter 7. In contrast to what the name suggests, parallel copy motion operates on register permutations covering all registers of the processor. The parallel copies are thus turned into permutations before the algorithm starts. The problem is then to find a good placement of these permutations within the program to minimize (1) the number of copies arising from projecting those permutations on the live registers at their final position and (2) the execution frequency of those copies, where the frequencies are either based on static estimates or profiling feedback.

Their algorithm proceeds by first treating permutations on critical edges within the control flow graph. This *Edge Motion* phase tries to use permutations on neighboring critical edges to cancel each other out and reduce the execution frequency of the resulting permutations. At the end, all permutations are either assigned to a basic block or otherwise the respective critical edge is split by forming a new basic block, the copy is then assigned to this block.

Next, during the *Block Motion* phase, the permutations are placed within basic blocks. The algorithm again tries to combine permutations to cancel each other out. The permutations are then placed within their basic blocks such that the number of copies, induced by projecting them to the set of live registers, is minimized using liveness information.

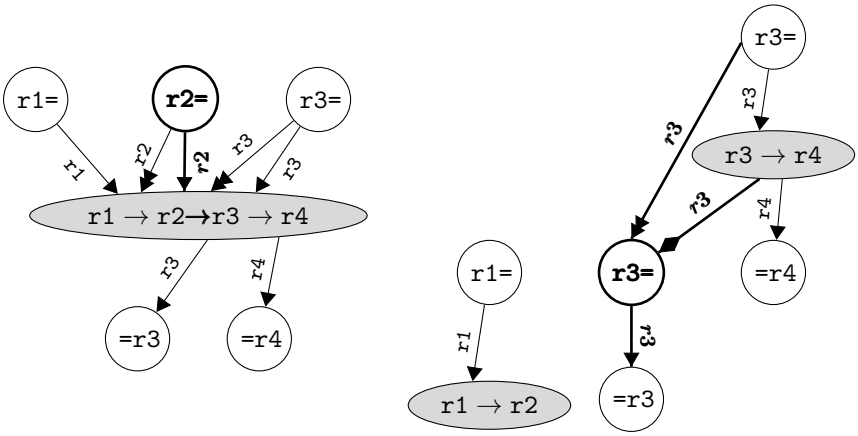
Basically, we reuse the *Edge Motion* phase without modification and replace the *Block Motion* phase by a more powerful technique based on data dependence graphs as explained in the next section.

8.2 Copy Elimination on Data Dependence Graphs

It is easy to see that it is possible to (partially) eliminate parallel copies by transforming the DDG and renaming register operands. We will present two different transformations that implicitly reorder the instructions of a basic block in order to eliminate parallel copies. The main idea is to *move* instructions

within the DDG *upward* or *downward* past the parallel copy, while at the same time renaming register operands. The involved parallel copy is split into smaller pieces as a side-effect of these transformations. This eliminates useless copies. In addition, it might enable other copy eliminations and break cyclic parallel copies, which usually have to be implemented using more costly swap operations.

The goal of these transformations is to eliminate as many copies within the DDG of each basic block as possible. The proposed transformations are thus repeatedly applied until no further simplifications of the DDG are possible, or a predefined time limit or threshold has been reached. Note that, since our algorithms operate on the DDGs of individual basic blocks, all copy operations incur equivalent runtime and code size costs, i.e., once a basic block is executed *all* the copy operations within that block are equally executed. Ignoring phase ordering issues, for instance with instruction scheduling, it is thus sufficient to reason only about the number of copies eliminated by the algorithms presented later on.



(a) Original DDG

(b) Transformed DDG

```
r2 = ...
r1 = ...
r3 = ...
r1 → r2 → r3 → r4
... = r4
... = r3
```

(c) Original Code

```
r1 = ...
r3 = ...
r1 → r2
r3 → r4
r3 = ...
... = r4
... = r3
```

(d) Transformed Code

Figure 8.3: A DDG before (a) and after (b) performing a downward motion of the definition of register r2.

8.2.1 Downward Motion of Definitions

The first form of transformation is to perform a *downward* motion of a *definition* of a register that is *used* by a parallel copy, i.e., the DDG contains a true dependence between the definition and the copy. It is then possible to move the definition down past the parallel copy while replacing the original register of the definition with the corresponding destination register of the parallel copy. The argument of the parallel copy becomes *dead* as a side-effect of this transformation, since the definition previously supplying the value now follows the parallel copy in any valid linear ordering of the DDG. The respective register-to-register copy thus becomes useless and can be eliminated, i.e., the parallel copy is split into two pieces. Due to the register renaming and splitting of the parallel copy, the DDG might need some additional updates in order to ensure correctness.

A more formal algorithm will be presented below, but we will first give a short example. Consider the original DDG from Figure 8.3a. The value calculated for register `r2`, the DDG node representing the definition is highlighted in bold, is immediately copied to register `r3`, without any other instruction touching the value. This copy operation can be avoided by register renaming and a minor update of the DDG. Figure 8.3b shows the final DDG after performing this transformation. Due to the renaming, some additional dependencies have to be added to the DDG as highlighted by bold lines. It is important to note that these dependencies can be easily derived from the dependencies of the original DDG node of the parallel copy. We also see that the copy has been split into two smaller pieces `r1` → `r2` and `r3` → `r4`, while the copy `r2` → `r3` was eliminated. This splitting is particularly interesting to break cyclic parallel copies because register swaps can be avoided.

8.2.1.1 Handling Regular Parallel Copies

Algorithm 15 Perform downward motion of a definition.

```
1:  procedure DefMotionDown(DDG G = (V,E), DDGNode def,
        Register r, DDGNode copy)
2:      // Ensure that no other uses exist besides copy
3:      uses ← {e | e = (def, u, r̄) ∈ E, u ∈ V}
4:      if uses ≠ {(def, copy, r̄)} then
5:          return
6:      // Check dependencies and transform the DDG
7:      if ¬ExistsPathFromDef(G, def, r, copy) then
8:          // Rename the destination register of def
9:          u ← ResultOfArgument(copy, r)
10:         RenameResult(def, r, u)
11:         // Split the parallel copy
12:         (lcopy, rcopy) ← SplitAtArgument(G, copy, r)
13:         // Update the data dependencies
14:         UpdateDefDependencies(G, def, r, u, lcopy, rcopy)
```

Algorithm 15 shows the main steps required to perform a downward motion of an instruction, denoted *def*, defining a register r with respect to a regular parallel copy operation *copy*. The algorithm first verifies that a linear ordering

can be derived from the DDG after performing the transformation, i.e., it verifies that no dependence cycles are introduced – see lines 2–7. The transformation is not performed if this check fails. The transformation itself consists of (1) renaming the destination register of the definition, (2) splitting the parallel copy, and (3) updating the DDG– see lines 8–14. It is important to note that only the destination register of the definition is renamed during this processing, all other registers, in particular, other register uses are *not* modified. Before we describe the phases in more detail, we define a few helper functions required during the processing:

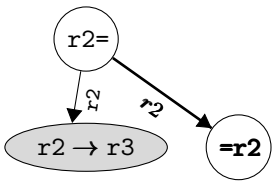
- ArgIndex and ResIndex take a parallel copy and a register as arguments and return the index within the argument list of the parallel copy. For instance, given the parallel copy $c = (d_1, \ldots, d_n) \leftarrow (a_1, \ldots, a_n)$, ArgIndex$(c, a_i)$ will return i, while ResIndex(c, d_j) will return j.
- ResultOfArgument takes a parallel copy and a register as an argument and returns the respective destination register of the argument register, i.e., for a copy $c = (d_1, \ldots, d_n) \leftarrow (a_1, \ldots, a_n)$ and register a_i, ResultOfArgument(c, a_i) returns d_i.
- The function RenameResult performs a simple renaming of the destination registers of an instruction represented by a DDG node.

8.2.1.2 Preventing Cyclic Dependencies

In order to perform a downward motion of a definition, it has to be assured that all data dependencies can be satisfied while a linear order of the DDG can still be derived after performing the transformation. Two situations have to be considered: (1) uses of the register defined by the instruction other than the parallel copy and (2) dependencies between the definition and the parallel copy.

In the former case, other uses than the parallel copy are simply rejected. This is foremost a simplification of the algorithm in order to avoid the need to track register uses and their relative position to the parallel copy in question. In particular, this avoids the need to track uses that are otherwise independent from the parallel copy, as shown by Figure 8.4. Note that we can always choose to rename those uses beforehand to allow the downward motion of the definition.

The second case arises foremost from data dependencies between the parallel copy and the definition, e.g., when the parallel copy uses or defines a register operand of the instruction *def*. In this case, after updating the DDG, we might

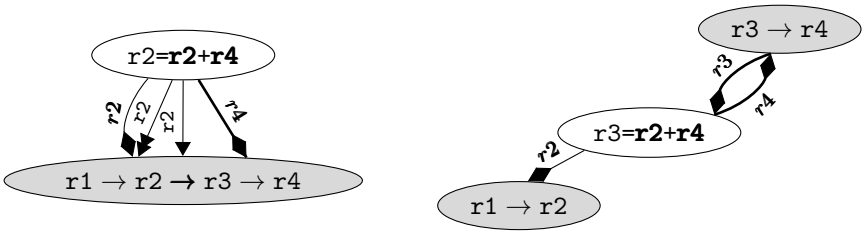


(a) DDG with extra use

```
r2  = ...
... = r2
r2 → r3
```

(b) Code with extra use

Figure 8.4: Care has to be taken that dependencies between a definition and its uses are not lost.



(a) Original DDG (b) Invalid transformed DDG

Figure 8.5: Performing a downward motion on the DDG in (a) results in cyclic data dependencies after the transformation (b).

find that the parallel copy has a (transitive) dependence leading to the definition, while at the same time a (transitive) dependence from the definition leads back to the parallel copy.

Figure 8.5 shows such an example. The bold anti dependence labeled with `r4` in Figure 8.5a leads to a cyclic dependence after the transformation. Note that it would be possible to resolve this issue by additional renaming if the respective values are still accessible through other registers. However, in the given example this is not possible since the value of `r4` is destroyed. We will come back to this issue in Section 8.2.3. The algorithm presented here does not consider the possibility of additional renaming as practical examples rarely require these transformations.

This is, in part, due to the fact that not all dependencies between the copy and the definition immediately lead to cyclic dependencies. Consider, for instance, the anti dependence labeled with `r2` in Figure 8.5b. This dependence remains in the final DDG even after the transformation is applied *without* causing any cycles. The reason for this is that the definition of register `r2` appears *before* the point where the parallel copy is split. The dependence thus cannot cause a cyclic dependence. This property applies to all register dependencies leading to regular parallel copies. We will make use of this property in the following algorithms.

As discussed later in more detail (see Lemma 2), the transformation may only lead to cyclic dependencies involving certain dependencies of one half of the split parallel copy. These dependencies are part of the original DDG before the transformation. Algorithm 16 thus explores all paths in the DDG leading from the definition *def* to the parallel copy *copy*. Whenever the path ends with a register dependence of an operand whose relative position is past the point where the parallel copy is split, the respective dependence will lead to a cycle during the transformation. All other kinds of dependencies are treated conservatively, i.e., are assumed to lead to cycles. These kinds of dependencies vary from compiler to compiler. As a general rule, memory dependencies are not problematic, since parallel copies do not access memory. Thus only control dependencies might pose problems. We assume DDGs of individual basic blocks only, which ensured that all control dependencies lead from some node in the DDG to the branch or call instruction terminating the basic block. Control

Algorithm 16 Check (transitive) dependencies between an instruction defining a register and a regular parallel copy using the same register.

```
1:  function ExistsPathFromDef(DDG G = (V,E), DDGNode def,
        Register r, DDGNode copy)
2:      for (n, copy, l) ∈ E where a path def →* n exists in G do
3:          if n = def and l ∈ {←r, →r, ↔r} then
4:              // Ignore all direct register dependencies

5:          else if ∃r_d : l ∈ {↔r_d, →r_d} then
6:              // Ignore anti and output dependencies to the parallel
7:              // copy if the involved operand appears before register r
8:              if ArgIndex(copy, r) ≤ ResIndex(copy, r_d) then
9:                  return TRUE

10:         else if ∃r_u : l = ←r_u then
11:             // Ignore true dependencies to the parallel copy if the
12:             // involved operand appears before register r
13:             if ArgIndex(copy, r) < ArgIndex(copy, r_u) then
14:                 return TRUE
15:         else
16:             return TRUE
17:     return FALSE
```

dependencies among other instructions cannot appear in this case. However, care has to be taken when other kind of dependencies are allowed or DDGs cover multiple basic blocks, e.g., for DDGs used by various forms of region scheduling [62, 44].

The usage of the relative position of arguments of the parallel copy is best understood when viewing the parallel copy as a chain of individual copy operations – as for example shown in Figure 8.2. The chain of copies is constructed from a regular parallel copy by processing the arguments of the copy in *reverse* order. Splitting the parallel copy then corresponds to splitting this chain of copy operations. Dependencies leading to the head of the chain, i.e., before the point where the copy is to be split, are problematic and will cause cyclic dependencies, while those leading to the tail of the chain, i.e., after the point where the copy is split, are safe.

Cyclic parallel copies, however, do not form a chain of copies but a cycle and thus need special treatment. We will return to this problem at the end of this section after discussing the final update procedure to keep the DDG in a consistent state after a downward motion of a definition.

8.2.1.3 Transforming the Dependence Graph

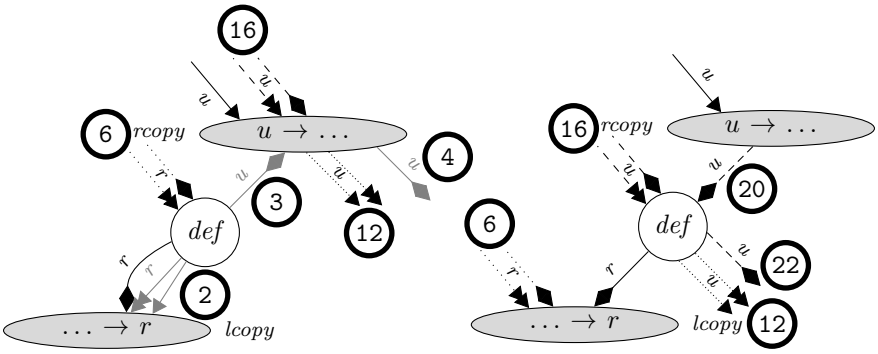
The transformation phase of downward code motion consists of three steps. First, the destination register of the definition is renamed – see Algorithm 15 line 10. Given that this step is straightforward, it is not further discussed here. We will instead focus on the splitting of the parallel copy (line 12) and the update of the DDG (line 14).

The parallel copy is split at the use point of the definition's register during

the transformation using SplitAtArgument, resulting in two independent parallel copies. Assume a parallel copy $c = (d_1, \ldots, d_{i-1}, d_i, d_{i+1}, \ldots, d_n) \leftarrow (a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n)$ and a register $r = a_i$. The splitting gives two copies $(d_1, \ldots, d_{i-1}) \leftarrow (a_1, \ldots, a_{i-1})$ and $(d_{i+1}, \ldots, d_n) \leftarrow (a_{i+1}, \ldots, a_n)$, where the first is referred to as *lcopy* and the second as *rcopy*. We, further, assume that, by splitting the copy, corresponding nodes are added to the DDG. The DDG nodes inherit the respective dependencies from the DDG node of the original parallel copy, which is discarded after the splitting. Note that both, *lcopy* and *rcopy*, may be empty at this point. For the sake of simplicity we assume that an empty copy will be inserted to the DDG in such a case – empty copies can easily be eliminated by a post-processing pass.

Algorithm 17 Update the dependencies of the DDG after a downward copy motion of a definition.

```
1:  procedure UpdateDefDependencies(DDG G = (V,E), DDGNode def,
    Register r, Register u, DDGNode lcopy, DDGNode rcopy)
2:      // Remove spurious dependencies
3:      remove (def, n, l) from E, for all n ∈ V, l ∈ {↔r, ←r}
4:      remove (def, rcopy, →u) from E
5:      remove (rcopy, n, →u) from E, for all n ∈ V

6:      // Transfer output and anti dependencies between def and lcopy
7:      if lcopy not is empty or ∃redef : (lcopy, n, →r) then
8:          let nextdef = { lcopy  , if lcopy is not empty
                          { redef  , otherwise
9:          for e = (n, def, l) ∈ E where l ∈ {↔r, →r} do
10:             remove e from E
11:             add (n, nextdef, l) to E

12:     // Transfer output and true dependencies between rcopy and def
13:     for e = (rcopy, n, l) ∈ E where l ∈ {↔u, ←u} do
14:         remove e from E
15:         add (def, n, l) to E

16:     // Transfer output and anti dependencies between rcopy and def
17:     for e = (n, rcopy, l) ∈ E where l ∈ {↔u, →u} do
18:         remove e from E
19:         add (n, def, l) to E

20:     // Account for redefinition of u by def
21:     add (rcopy, def, →u) to E

22:     // Create an anti dependence to the next definition of u
23:     if ∃(def, n, ↔u) ∈ E and def has a use of u then
24:         add (def, n, →u) to E
```



(a) Before transforming the DDG (b) After transforming the DDG

Figure 8.6: Illustration of the DDG update procedure after splitting the parallel copy, showing the DDG before (a) and after (b) the transformation. The bold circled numbers indicate the corresponding line of Algorithm 17.

After splitting the parallel copy the DDG has to be updated in order to account for new and/or eliminated data dependencies due to the register renaming, the copy elimination induced by the splitting of the parallel copy, or the code motion of the definition. Algorithm 17 shows the necessary steps to update the DDG given a definition *def* defining a register r, which is to be updated by register u, and two split pieces of the original parallel copy *lcopy* and *rcopy*. First, dependencies are removed that become superfluous, either due to the register renaming at the definition or the copy elimination. Then, anti and output dependencies leading to the definition and labeled with the definition's original register are *redirected* to the left half of the parallel copy. Note, true dependencies are not affected by the renaming and thus are left untouched. A similar transfer of dependencies is performed for output and true dependencies originating from the right piece of the parallel copy and labeled with the definition's new destination register u. Yet another transfer of dependencies is performed for dependencies leading to the right half of the parallel copy and labeled with the definition's new destination register. Finally, a new mandatory data dependence between the right piece of the parallel copy and the definition is added to the DDG labeled with u. Also, a potential anti dependence added to the DDG if the definition uses that register. A more detailed discussion of the algorithm is given below (see Lemma 1).

Figure 8.6 illustrates the various steps performed during the DDG update labeled with the corresponding line number of Algorithm 17. The lines of the affected data dependencies are, in addition, highlighted: dependencies that remain untouched are represented by solid black lines, those that are removed are gray, newly added dependencies are densely dashed, while other dependencies transferred between the parallel copies and the definition are represented by matching line styles at their original position before and final position after the transformation (dashed, dotted, densely dotted).

8.2.1.4 Correctness

In order to verify the correctness of our approach two properties have to be considered: (1) that data dependencies are neither falsely lost nor falsely introduced during the transformation and (2) that a linear ordering of the DDG nodes can be established, i.e., the DDG is free of cycles.

Lemma 1. *Given an acyclic DDG, a definition, and a regular parallel copy, Algorithm 17 correctly updates the DDG after splitting the parallel copy and renaming the definition. All data dependencies are preserved, i.e., dependencies are neither falsely lost not falsely introduced.*

Proof. The update procedure only operates on register dependencies carrying registers r and u, it is thus sufficient to consider the various cases of definitions and uses of these two registers before the transformation. The respective instructions are assumed to either (1) precede *def*, (2) precede the parallel copy, but not necessarily *def*, or (3) succeed the parallel copy in any valid linear ordering.
The three cases for another definition *dr* of r are as follows:

(1) The output dependence between *dr* and *def*, becomes obsolete due to the renaming. The dependence is redirected to the next redefinition of r succeeding *def*. Unless *lcopy* is empty it is the next redefinition succeeding *def* (see Algorithm 17, l. 7). If *lcopy* is empty, the dependence is redirected to the closest redefinition succeeding *lcopy* and thus *def*, if one exists. The dependence is thus correctly updated.

(2) This is impossible. *def* would not be a candidate for the transformation.

(3) This case is only relevant when *lcopy* is empty, since otherwise the output dependence between *lcopy* and *dr* is irrelevant for the DDG update. In case *lcopy* is empty, *dr* is a redefinition handled by case (1).

The cases for a use of r are analogous to the three cases above, with the exception that the dependencies in question are anti dependencies. True dependencies are not relevant for the transformation since they cannot refer to *def*. This is also true for case (3) when *lcopy* is non-empty.

Some dependencies carrying r are removed unconditionally (l. 3). The various cases discussed above reintroduce corresponding output and anti dependencies covering these dependencies when necessary. It is thus ensured that no dependencies are lost.
Considering the three cases for a definition *du* of register u:

(1) The output dependence between the parallel copy is obsolete due to the elimination of the copy $r \to u$. This dependence has to be redirected to *def*, since *def* defines u after renaming, (see l. 17). Anti dependencies leading to *du* are not relevant for the transformation.

(2) This is equivalent to case (1), unless a (transitive) dependence leads from *def* to *du* – and thus to the parallel copy. The transformation would then result in a cyclic dependence. This situation is covered by Algorithm 16.

(3) In this case, the output dependence between the parallel copy and *du* becomes obsolete due to the copy elimination. The dependence needs to

be transferred to *def*, since it defines u after renaming (see l. 13). In contrast to case (1), an anti dependence between the parallel copy and *du* can exist. This dependence becomes obsolete and has to be redirected to *def*, due to the register renaming. Furthermore, *def* may use register u. This requires an additional anti dependence between *def* and *du* (see l. 3, l. 21, and l. 23).

Cases (1) and (2) for uses of u are analogous to the respective cases above, with the exception that anti dependencies are considered. True dependencies are not relevant for both cases. The situation for case (3) is different. True dependencies between the parallel copy become obsolete due to the copy elimination the the register renaming. These dependencies have to be transferred from *lcopy* to *def*, since *def* replaces the previous definition of u by the copy (see l. 13). Anti dependencies originating from uses of u are not relevant.

Dependencies carrying u that are removed from the DDG (l. 3) are either replaced by corresponding anti dependencies or become simply obsolete. Thus no dependencies are lost.

Finally, other kinds of dependencies have to be considered. Since parallel copies never touch memory, corresponding dependencies are not an issue for the DDG update. In the case of simple basic blocks, the only control dependencies in the DDG lead from every node to the branch terminating the block. Non-register dependencies thus do not require additional handling during the DDG update. It follows that Algorithm 17 is correct. □

The previous proof already covered some cases potentially causing cyclic dependencies after the transformation. However, these cases do not cover all potential cycles. These are covered by the following lemma.

Lemma 2. *Given an acyclic DDG, a definition, and a regular parallel copy, Algorithm 16 ensures that the DDG remains acyclic after the transformation through Algorithm 17.*

Proof. The DDG is initially acyclic by definition. Furthermore, all cyclic dependencies have to include the anti dependence between *rcopy* and *def* introduced during the DDG update (see Algorithm 17, l. 21). All other dependencies cannot lead to cycles since they arise from simple transfers between nodes that originally were already ordered.

It thus remains to show that transitive dependence between *def* and *rcopy* cannot occur. Since *rcopy* inherits all its dependencies from the original parallel copy, any dependence between *def* and *rcopy* has to be present in the original DDG before the transformation. Note that only one dependence is added to the DDG involving *rcopy* (Algorithm 17, l. 21). It follows that cycles can be detected in the original DDG by examining dependencies between *def* and the original parallel copy. Note that only dependencies inherited by *rcopy* are relevant.
In the loop of Algorithm 16, the following four cases have to be considered:

(1) Direct output and true dependencies between *def* and the copy become obsolete and are thus removed. An anti dependence may remain in the DDG even after the transformation, but it simply refers to *lcopy*, the left half of the parallel copy. A cycle is thus impossible.

(2) Output and anti dependencies carrying other registers and/or originating from other nodes are *not* removed. It thus has to be ensured that these

dependencies refer to *lcopy* after the transformation. That is, the respective register needs to appear before or at the same position as r in the original parallel copy (also see the definition of SplitAtArgument). A cycle is thus impossible.

(3) Other kinds of true dependencies leading to the parallel copy similarly remain untouched during the transformation. Again, it has to be ensured that the dependence will refer to *lcopy* after the transformation. That is, the argument has to appear (strictly) before the position of r in the original parallel copy. Note that, in fact, the only true dependence leading to the position of r originates from *def* and thus is handled by (1). A cycle is thus impossible.

(4) This final case can never occur in DDGs of simple basic blocks. Memory dependencies cannot refer to parallel copies and control dependencies in DDGs of basic blocks can only refer to the branch terminating the basic block. This case is still included to highlight to potential of cyclic dependencies when other flavors of DDGs are used (e.g., in flavors used for region scheduling [62, 44]). Treating those dependencies conservatively renders cycles impossible.

Algorithm 16 thus ensures that the DDG remains acyclic after the the update procedure as given by Algorithm 17. □

The two lemmas ensure that no dependencies are falsely lost nor introduced when performing a downward motion of a definition. Furthermore, cyclic dependencies cannot occur during such a transformation.

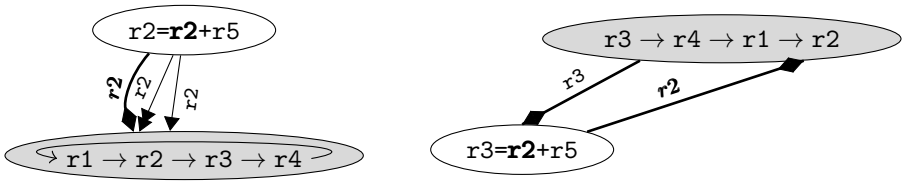
Theorem 5. *Algorithm 15 is correct for regular parallel copies.*

Proof. Since the renaming and splitting of the parallel copy are trivial, this follows from Lemma 1 and Lemma 2. □

8.2.1.5 Handling Cyclic Parallel Copies

Cyclic copies have to be treated conservatively. If a path exists in the DDG that leads from the definition to the parallel copy, a cyclic dependence will be introduced, unless the path is of length 1 consisting of a single true or output dependence labeled with the definition's register. These direct dependencies can safely be ignored since the parallel copy will be split and the respective dependencies will be removed during the transformation. Note that anti dependencies cannot be discarded at this point, unless the respective register use is renamed – which we do not considered here. Figure 8.7 shows an example of this situation.

The algorithms presented previously for the case of regular parallel copies need to be adapted in order to handle cyclic parallel copies correctly. Firstly, splitting a parallel copy always gives two non-empty regular parallel copies. The function SplitAtArgument is thus redefined in the case of a cyclic parallel copy as follows: given a cyclic parallel copy $c = (d_1, \ldots, d_{i-1}, d_i, d_{i+1}, \ldots, d_n) \leftarrow (a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n)$ and a register $r = a_i$. The function constructs a non-empty copy $(d_{i+1}, \ldots, d_n, d_1, \ldots, d_{i-1}) \leftarrow (a_{i+1}, \ldots, a_n, a_1, \ldots, a_{i-1})$. This copy is then returned as the left *and* the right piece of the split copy. In other words, *lcopy* and *rcopy* in the previous algorithms for regular parallel copies refer to the same copy.



(a) Original DDG (b) Invalid transformed DDG

Figure 8.7: Performing a downward motion on the DDG in (a) results in cyclic data dependencies after the transformation (b).

Algorithm 18 Check (transitive) dependencies between an instruction defining a register and a, potentially cyclic, parallel copy using the same register.

```
1:  function ExistsPathFromDef(DDG G = (V,E), DDGNode def,
        Register r, DDGNode copy)
2:      for (n, copy, l) ∈ E where a path def →* n exists in G do
3:          if cyclic then
4:              // Ignore direct true and output dependencies for cyclic parallel copies
5:              if n ≠ def or l ∉ {←r, ↔r} then
6:                  return TRUE
7:          else
8:              // Code of Algorithm 16
9:              ...
10:     return FALSE
```

A second issue stems from the fact that a cyclic parallel copy defines and uses every operand register once. Note, in particular, that the original destination register of the definition might be redefined by the parallel copy. This causes additional dependencies that have to be accounted for in order to prevent cyclic dependencies. Algorithm 18 shows an adapted variant of the ExistsPathFromDef function. The update procedure can be used as it is.

8.2.1.6 Correctness

The arguments proofing the correctness of the downward motion of definitions on regular parallel copies carry over to the specific case of cyclic parallel copies. The discussion is thus kept short:

Lemma 3. *Given an acyclic DDG, a definition, and a cyclic parallel copy, Algorithm 17 correctly updates the DDG after splitting the parallel copy and renaming the definition. All data dependencies are preserved, i.e., dependencies are neither falsely lost not falsely introduced.*

Proof. Analogous to Lemma 1. The only difference arises from the fact that *lcopy* and *rcopy* denote the same non-empty parallel copy. □

Lemma 4. *Given an acyclic DDG, a definition, and a cyclic parallel copy, Algorithm 18 ensures that the DDG remains acyclic after the transformation through Algorithm 17.*

Proof. Analogous to Lemma 2. The only difference arises from the fact that the parallel copy defines and uses all its register operands. Thus, all dependencies leading to the parallel copy may lead to cycles. □

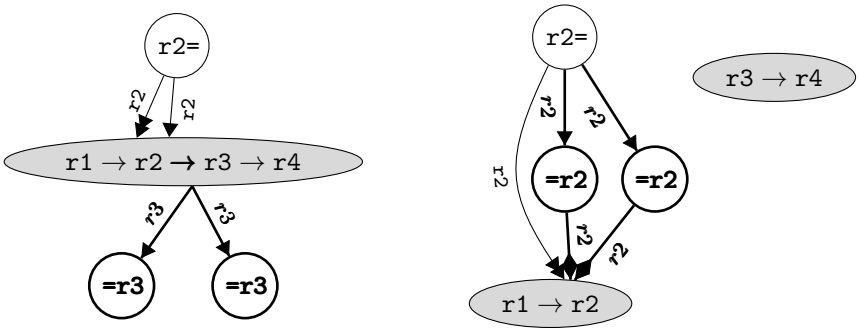
Theorem 6. *Algorithm 15 is correct for cyclic parallel copies.*

Proof. This follows from Lemma 3 and Lemma 4. □

8.2.2 Upward Motion of Uses

Another form of transformation is to perform an *upward* code motion of *all* uses of a register defined by a parallel copy while renaming the respective register uses. The result of this transformation, as before, is that the involved register becomes dead, the copy operation can thus be eliminated and the parallel copy split. As before, we start by giving an informal example of the transformation and then present detailed algorithms.

Consider the DDG from Figure 8.8a, where all uses of register `r3`, which is defined by the parallel copy, are highlighted in bold. The copy `r2` → `r3` can be eliminated if all these uses are renamed as shown by Figure 8.8b. As with the downward motion of definitions, some dependencies become useless due to this transformation, while at the same time new dependencies arise. For instance, the true dependence of the respective uses have to be updated to reflect the reordering and register renaming, as indicated by the bold true dependencies. In addition, new anti dependencies arise due to the definition of `r2` by the parallel copy (also highlighted in bold). Similar to before, the transformation may cause cyclic dependencies, which have to be avoided.



(a) Original DDG

(b) Transformed DDG

```
r2  = ...
r1 → r2 → r3 → r4
... = r3
... = r3
```

(c) Original Code

```
r3 → r4
r2  = ...
... = r2
... = r2
r1 → r2
```

(d) Transformed Code

Figure 8.8: A DDG before (a) and after (b) performing an upward motion of all uses of register r3.

Algorithm 19 Perform upward motion of a all uses.

```
1:  procedure UseMotionUp(DDG G = (V,E), DDGNodes uses, Register r,
        DDGNode copy)
2:      // Ensure that no other uses exist
3:      all_uses ← {u | (copy, u, r̄) ∈ E, u ∈ V}
4:      if all_uses ≠ uses then
5:          return

6:      // Check dependencies and transform the DDG
7:      if ¬ExistsPathToUses(G, uses, r, copy) then
8:          // Perform register renaming for every use
9:          u ← ArgumentOfResult(copy, r)
10:         RenameUses(G, uses, r, u)

11:         // Split the parallel copy
12:         (lcopy, rcopy) ← SplitAtResult(G, copy, r)

13:         // Update the data dependencies
14:         UpdateUseDependencies(G, uses, r, u, lcopy, rcopy)
```

8.2.2.1 Handling Regular Parallel Copies

In contrast to the code motion of definitions, *all* uses have to be considered during an upward motion. The following Algorithm 19 thus operates on the set of all uses, but otherwise follows the same principal phases as the downward motion of a definition. Given a set of DDG nodes *uses* reading register r and a parallel copy *copy* defining the same register, it is first verified that the upward motion is legal and does not cause any cyclic dependencies (lines 2–7). In the next step all register uses are renamed (line 10), before the parallel copy is split (line 12). Finally, the data dependence graph is updated for every use (line 14). The algorithms following hereafter make use of some helper functions, which are defined as follows:

- ArgIndex and ResIndex are defined as before – see Section 8.2.1.1.
- ArgumentOfResult takes a parallel copy and a register as an argument and returns the corresponding argument of the parallel copy for the matching destination register, i.e., for a copy $c = (d_1, \ldots, d_n) \leftarrow (a_1, \ldots, a_n)$, ArgumentOfResult$(c, d_i)$ returns a_i.
- The function RenameUses performs a simple renaming of the argument registers of the instructions represented by the uses in the DDG.

8.2.2.2 Preventing Cyclic Dependencies

In order to ensure that the DDG is in a valid state after an upward motion of the uses of a register defined by a parallel copy, it has to be guaranteed that (1) *all* uses are considered, and (2) no cyclic dependencies arise from the transformation.

The former case depends, to some extent, on the DDG representation, in particular, on how register uses outside of the scope of the currently considered DDG are represented. For instance, if the DDG covers basic blocks only, registers might be used by an instruction in a successor basic block, i.e., the register is live-out of the current basic block. The corresponding use is not amenable to code motion, since it is not covered by the DDG. For this work, we assume that artificial DDG nodes represent all *external* uses of registers live-out of the code region covered by the DDG. Since those artificial uses are not amenable to code motion they cannot appear in the set of *uses* in Algorithm 19, but may well appear in the set *all_uses* (line 3).

The second issue is similar to the problem of cyclic dependencies that appeared for the downward motion of definitions. Algorithm 20 shows the corresponding test that verifies that no cyclic data dependencies may arise from the transformation. The test is, in fact, very similar to that of Algorithm 16, except that the direction of the examined paths is inverted (line 3), i.e., DDG edges originating from the parallel copy are examined. Consequently, the relation between anti and true dependencies and the relative position of the respective operands within the parallel copy is inverted too (see line 10 and 5). The algorithm otherwise proceeds in the same manner as the corresponding version for the downward motion of definitions. Please refer to Section 8.2.1.1 for a more detailed discussion.

8.2.2.3 Transforming the Dependence Graph

The transformation phase of the upward code motion of uses consists of three principal steps. First, register renaming is performed using the ArgumentOfResult and RenameUses functions.

In the next step, the original parallel copy is split using the SplitAtResult function, which given a parallel copy $c = (d_1, \ldots, d_{i-1}, d_i, d_{i+1}, \ldots, d_n) \leftarrow (a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n)$ and a register $r = d_i$ returns two new parallel copies, $(d_1, \ldots, d_{i-1}) \leftarrow (a_1, \ldots, a_{i-1})$ and $(d_{i+1}, \ldots, d_n) \leftarrow (a_{i+1}, \ldots, a_n)$, which are denoted as *lcopy* and *rcopy* respectively. The two copies inherit the respective data dependencies from the original parallel copy – in particular, *rcopy* is assumed to inherit all dependencies carrying the original register and *lcopy* is assumed to inherit all dependencies carrying the new register used for renaming. The main difference to the corresponding function SplitAtArgument from Section 8.2.1.1 is that the registers defined by the parallel copy are examined instead of the registers used in order to determine the split point. An additional difference will become apparent when the handling of cyclic parallel copies is discussed later in the next section.

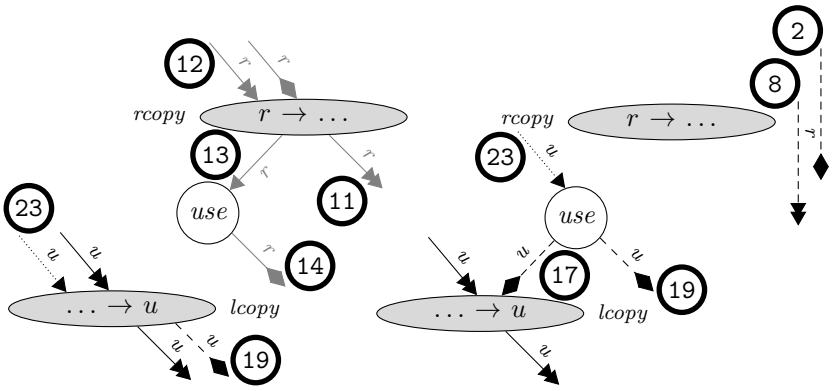
Finally, the data dependencies of the DDG, i.e., of the involved uses and pieces of the parallel copy, have to be updated. Since only register uses are touched the situation is simple, only true and anti dependencies originating from respectively leading to the use and the left half of the parallel copy as well as true and output dependencies to respectively from the right half of the copy have to be considered. The DDG update procedure, shown by Algorithm 21, takes a set of DDG nodes *uses*, a register r, which is read by the uses, a register

Algorithm 20 Check (transitive) dependencies between a regular parallel copy defining a register and all uses of that register.

```
1:  function ExistsPathToUses(DDG G = (V,E), DDGNodes uses, Register r,
        DDGNode copy)
2:      for (copy, n, l) ∈ E where a path n →* u exists in G, u ∈ uses do
3:          if n = u and l ∈ {←r, →r, ↔r} then
4:              // Ignore all direct register dependencies
5:          else if ∃r_d : l ∈ {↔r_d, ←r_d} then
6:              // Ignore true and output dependencies originating from the
7:              // parallel copy if the involved operand appears after register r
8:              if ResIndex(copy, r) > ResIndex(copy, r_d) then
9:                  return TRUE
10:         else if ∃r_u : l = →r_u then
11:             // Ignore anti dependencies originating from the parallel copy
12:             // if the involved operand appears after register r
13:             if ResIndex(copy, r) ≥ ArgIndex(copy, r_u) then
14:                 return TRUE
15:         else
16:             return TRUE
17:     return FALSE
```

Algorithm 21 Update the dependencies of the DDG after a upward copy motion of a set of register uses.

```
1:  procedure UpdateUseDependencies(DDG G = (V,E), DDGNodes uses,
    Register r, Register u, DDGNode lcopy, DDGNode rcopy)
2:      // Account for dependencies after removing the definition of r
3:      if ∃redef ∈ V : (rcopy, redef, r→) ∈ E then
4:          // Account for anti dependencies
5:          for n ∈ V : (n, cropy, r→) ∈ E do
6:              add (n, redef, r→) to E
7:              remove (n, rcopy, r↔) from E

8:          // Account for output dependencies
9:          if ∃n ∈ V : (n, rcopy, r↔) then
10:             add (n, redef, r↔) to E

11:     // Remove spurious dependencies
12:     remove (rcopy, n, r↔) from E, n ∈ V
13:     remove (n, rcopy, r↔) from E, n ∈ V
14:     remove (rcopy, n, r←) from E, ∀n ∈ uses
15:     remove (n1, n2, r→) from E, ∀n1 ∈ uses, n2 ∈ V

16:     // Transfer anti dependencies carrying the new register u
17:     if lcopy not is empty then
18:         add (n, lcopy, u→) to E, ∀n ∈ uses
19:         remove (lcopy, n, u→) from E, ∀n ∈ uses
20:     else if ∃n1 ∈ V : (lcopy, n1, u→) ∈ E then
21:         add (n2, n1, u→) to E, ∀n2 ∈ uses
22:         remove (lcopy, n1, u→) from E
23:     // Transfer true dependencies carrying the new register u
24:     if ∃n1 ∈ V : (n1, lcopy, u←) ∈ E then
25:         add (n1, n2, u←) to E, ∀n2 ∈ uses
26:         remove (n1, lcopy, u←) from E
```



(a) Before transforming the DDG (b) After transforming the DDG

Figure 8.9: Illustration of the DDG update procedure, showing the DDG before (a) and after (b) the transformation. The bold circled numbers indicate the corresponding line of Algorithm 21.

u to update those uses and two split pieces of the original parallel copy *lcopy* and *rcopy*. It proceeds in four main steps as follows.

First, output and anti dependencies carrying the original register r are handled. Here, special care has to be taken for the case then *rcopy* is empty. Some dependencies then have to be established between definitions and/or uses of r preceding and succeeding the parallel copy. Next, dependencies carrying the original register r involving any of the uses are removed. In comparison to the downward motion of definitions, the situation is simpler, since uses, as opposed to definitions, do not interfere with succeeding or preceding instructions using or defining that register. Subsequently, output dependencies carrying the register u are added to the DDG. Two cases need to be distinguished: If *lcopy* is non-empty, it redefines u and output dependence from all uses have to be inserted. In case the copy is empty, output dependencies are only appended if a redefinition of u succeeds the parallel copy. By now the DDG is almost complete, the only missing dependencies are true dependencies carrying the new register u. These can simply be derived from the left half of the parallel copy.

An illustration of the individual steps of the update procedure are shown by Figure 8.9. The bold circled numbers relate the respective dependencies with the corresponding code lines of Algorithm 21. The line style indicates untouched dependencies (solid), potentially removed dependencies (gray), newly added dependencies (densely dashed), and dependencies transferred between the parallel copy and the uses under consideration (dashed/dotted).

8.2.2.4 Correctness

As before, the correctness of the upward code motion of uses depends on (1) the fact that all data dependencies are correct after the transformation and (2) no cyclic dependencies arise.

Lemma 5. *Given an acyclic DDG, a set of uses, and a regular parallel copy, Algorithm 21 correctly updates the DDG after splitting the parallel copy and renaming the uses. All data dependencies are preserved, i.e., dependencies are neither falsely lost not falsely introduced.*

Proof. The algorithm operates on dependencies involving register dependencies carrying the registers r and u only. It is thus sufficient to consider definitions and uses of those registers surrounding the respective parallel copy. We will thus consider various cases involving such instructions whose relative position in any linear ordering before the transformation is as follows: the instruction (1) precedes the parallel copy, (2) succeeds the parallel copy (but not necessarily the uses involved in the transformation), or (3) succeeds some or all uses.

Considering a definition dr of register r, the following cases may occur:

(1) In case an output dependence leading from dr to the parallel copy exists, it becomes obsolete due to the splitting of the copy. The dependence has to be redirected to the next redefinition of the register, succeeding all uses (see Algorithm 21, l. 9 and 13). Any potential true dependencies leading to the parallel copy are not affected by the transformation and are inherited by *rcopy*.

(2) This case is impossible, since some or all of the uses are not candidates for the transformation. Otherwise, this case is equivalent to case (3) below.

(3) The only relevant scenario is that dr succeeds *all* uses. In case an output dependence between the copy and dr exists, it becomes obsolete. The situation is then handled by case (1) above. All anti dependencies between the uses of r and dr become obsolete due to the renaming of the uses (see l. 15). Furthermore, new anti dependencies may arise between uses of r preceding the parallel copy and dr due to the splitting of the copy (see l. 3). The anti dependence between the parallel copy and dr as well as any true dependencies originating from dr are not relevant to the transformation.

For a use ur of register r the following cases arise:

(1) The case corresponds to the case (1) for definition dr from above, with the exception that anti dependence instead of output dependencies have to be considered.

(2) The only interesting scenario here is that ur appears before the next redefinition of r, i.e., $ur \in uses$. The true dependencies between the copy and ur become obsolete due to the renaming of the use. The dependence needs to be redirected to the first definition preceding the parallel copy defining u (l. 24). Anti dependencies originating from ur similarly become obsolete. If *lcopy* is empty the anti dependence originating from it has to be duplicated at each use (l. 20). Otherwise, *lcopy* defines u and thus requires an anti dependence to all uses (l. 18).

(3) This case is covered by (2) above.

For a definition du of u the following cases have to be considered:

(1) True dependencies originating from *du* are covered by case (2) for a use *ur* above. If an output dependence between *du* and the parallel copy exists, it remains unchanged even after the splitting. Similarly, anti dependencies are not relevant to the transformation.

(2) This is impossible unless *du* succeeds all uses. Then case (3) applies.

(3) The only relevant scenario here is that *du* succeeds *all* uses. In case a output dependence between the parallel copy and the definition exists, it remains unchanged – as do true dependencies originating from *du*. Anti dependencies between uses and *du* become obsolete (this case is covered by case (2) for uses *ur* above. Other anti dependencies are not relevant for the transformation.

All register dependencies carrying register u and leading to/originating from any use are not relevant for the transformation, because all definitions of u remain untouched, i.e., no such definition is renamed or eliminated.

Other kinds of dependencies, not covered by the cases above, are also irrelevant to the transformation (considering DDGs on basic blocks). It follows the correctness of the algorithm. □

The preceding lemma shows that the DDG update procedure correctly preserves all dependencies. It remains to show that no cyclic dependencies arise from the transformation.

Lemma 6. *Given an acyclic DDG, a set of uses, and a regular parallel copy, Algorithm 20 ensures that the DDG remains acyclic after the transformation through Algorithm 21.*

Proof. The DDG is initially acyclic by definition. Furthermore, all cyclic dependencies have to include the anti dependence between some use and *lcopy* introduced during the DDG update (see Algorithm 21, l. 18 and 20). All other dependencies cannot lead to cycles since they arise from simple transfers between nodes that originally were already in an ordering relation.

First consider the case when *lcopy* is not empty, i.e., line 18 of Algorithm 21 is executed. Any cycle then has to include a path from *lcopy* to some use. Since no dependencies are added or removed originating from *lcopy*, this path also exists in the original DDG before the transformation (note line 20 of Algorithm 21 is never executed). It is thus sufficient to check the original DDG to recognize any potential cycles before the transformation.

Algorithm 20 examines all paths from the parallel copy to any use. Four different cases are distinguished:

(1) Direct register dependencies between the parallel copy and any use cannot result in a cycle. These dependencies become obsolete due to the register renaming.

(2) Output and true dependencies originating from a register operand appearing in the right half of the parallel copy cannot cause any cycles, since they will only appear in *rcopy*.

(3) Similarly, anti dependencies originating from the right half cannot lead to any cycles, as they will only appear in *rcopy*.

(4) Other kinds of dependencies are assumed to lead to cycles. As noted before, this can only appear in DDGs covering more than one basic block. The case is included for completeness and to highlight this issue.

Now consider the case when *lcopy* is empty, i.e., line 20 of Algorithm 21. Cycles may then arise from some definition *du* of u preceding some use, but succeeding the parallel copy in the original DDG. Since *du* defines u and the parallel copy uses u, a path from the copy to *du* exists (involving at least one anti dependence carrying u). It is thus sufficient to check the original DDG to recognize any potential cycles before the transformation as shown above.

Algorithm 20 thus ensures that the DDG remains acyclic after the update procedure as given by Algorithm 21. □

Theorem 7. *Algorithm 19 is correct for regular parallel copies.*

Proof. Since the renaming of the uses and the splitting of the parallel copy are trivial, this follows from Lemma 5 and Lemma 6. □

8.2.2.5 Handling Cyclic Parallel Copies

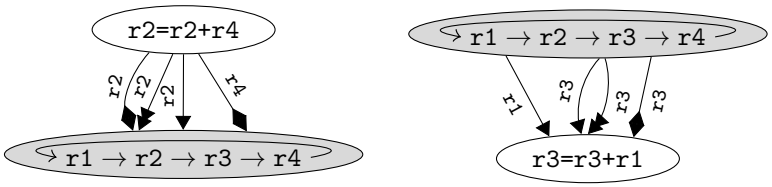
The upward motion of uses is affected by cyclic parallel copies to a lesser extent, since even regular copies might redefine both the used register before and after renaming. The DDG update procedure of Algorithm 21 can directly be applied to cyclic copies without modification. However, in order to prevent cyclic data dependencies Algorithm 20 has to be extended. The problem arises from the way cyclic copies are split. We will thus first discuss how the splitting is defined.

As noted before, the splitting of a cyclic parallel copy always yields one non-empty parallel copy. In Section 8.2.1.5, for the downward motion of definitions, the function SplitAtArgument was extended accordingly to correctly split cyclic copies. The function SplitAtResult is refined in exactly the same way, i.e., for the update procedure receives the non-empty result of SplitAtResult as both *lcopy* and *rcopy*.

Since the register of the respective uses after renaming is redefined by the parallel copy, it is now easy to see that any dependence between the copy and

Algorithm 22 Check (transitive) dependencies between a potentially cyclic parallel copy defining a register and all uses of that register.

```
1:  function ExistsPathToUses(DDG G = (V,E), DDGNodes uses, Register r,
        DDGNode copy)
2:      for (copy, n, l) ∈ E where a path n →* u exists in G, u ∈ uses do
3:          if cyclic then
4:              // Ignore true dependencies only
5:              if l ≠ ↼r then
6:                  return TRUE
7:          else
8:              // Code of Algorithm 20.
9:              ...
10:     return FALSE
```



(a) Original DDG (b) Transformed DDG

Figure 8.10: Downward and upward code motion is possible past cyclic parallel copies by renaming all operands of the respective instruction that also appear as operands of the copy.

any use immediately leads to a cycle in the DDG, unless the dependence is a true dependence carrying register r. Algorithm 22 ensures that, in the case of cyclic copies, all other forms of dependencies are rejected when paths between the copy and the respective uses are explored.

8.2.2.6 Correctness

Lemma 6 is easy to adapt to handle cyclic parallel copies, we thus do not further discuss it here. Lemma 5 applies to cyclic parallel copies without any modification. We thus conclude:

Theorem 8. *Algorithm 19 is correct for cyclic parallel copies.*

Proof. This follows from the lemmas given above (with minor adaption). □

8.2.3 Code Motion Past Cyclic Parallel Copies

In the previous sections the focus was on *eliminating* individual copies by moving individual instructions or sets of instructions downward or upward past a regular or cyclic parallel copy. The goal was to render a given register dead by renaming the corresponding definition or all uses respectively. In addition, an instruction can be moved past a cyclic parallel copy, both downward or upward, by renaming *all* register operands of the instruction that appear in the parallel copy. The only requirement is that no (transitive) non-register dependencies exist between the DDG node of the instruction and the parallel copy. An example of this transformation is shown in Figure 8.10.

Such a transformation is, by itself, not necessarily useful, ignoring indirect benefits that might arise in following optimization steps, e.g., instruction scheduling. However, the ability to move arbitrary instructions past a cyclic copy gives additional freedom and might be used to enable the elimination of a copy by one of the previously described techniques. Even more, it might sometimes be beneficial to turn a regular parallel copy into a cyclic one and move instructions past that copy to enable further possibilities for copy elimination.

8.2.4 Algorithm Complexity

The coalescing problem on global interference graphs is well studied in the literature, and complexity results showing that various variants of the problem

are NP-complete are available [17]. Our DDG-based copy elimination operates locally on basic blocks, which eventually might reduce the problem complexity. Unfortunately, this is not the case. Biró et al. [9] show that the coalescing problem is NP-complete even for block-local interference graphs, i.e., interval graphs, when pre-coloring constraints have to be respected. These constraints are required for our problem to ensure that the local coalescing respect the register assignments of predecessor and successor blocks. The general problem of finding an optimal local coalescing combined with instruction reordering is thus also NP-complete.

The complexity of our heuristic clearly depends on the size of the parallel copies of a basic block, i.e., the number of edges within all the parallel copies of that block. This in turn depends on how aggressive live-range splitting is performed. Considering SSA form, live-range splitting may only occur on join points in the control flow graph (CFG). Edge motion then assigns the parallel copies to surrounding basic blocks, inserting them either at the beginning or appending them at the end of a basic block. Since every processor register can only be assigned a new value *once* at the beginning and the end of a basic block, the total size of the parallel copies within a basic block is bounded by the constant $2k$, where k is the number of processor registers.

The algorithm complexity is thus dominated by the path searches performed for Algorithm 16 and 20. This path search can be performed using depth-first search in $O(|V| + |E|)$, where $|V|$ and $|E|$ correspond to the the number nodes and edges in the DDG respectively. The for-loop of both algorithms is bounded by $O(|V|)$, since all checks inside the loops can be performed in constant time – note again that the number of processor registers is constant.

The DDG update, Algorithm 17 and 21, operates locally on edges leading to/from a parallel copy to other nodes. The for-loops of the respective algorithms therefore execute in $O(|V| + |E|)$.

The handling of cyclic parallel copies neither increases the complexity of the path search nor the update procedure. The overall complexity of our algorithm, when live-range splitting is performed according to SSA form, is thus linear in the size of the DDG of a basic block $O(|V|+|E|)$. In the case of more aggressive live-range splitting after every instruction in the program the complexity is in $O(|V|^2+|V||E|)$. Even under such an aggressive live-range splitting, the number of program points where parallel copies remain is usually low.

8.2.5 Additional Remarks

The algorithms presented in the previous sections are invoked iteratively as long as parallel copies and candidate instructions, i.e., respective DDG nodes, exist that might be eligible for code motion, or a predefined threshold has been reached. So far, it was assumed that all instructions are amenable to renaming. However, in practice this is not always the case, in particular, when additional register constraints have to be accounted for. These constraints may arise from registers that are accessed by an instruction independent from its operands, i.e., *clobbered* registers, condition code registers, fixed operands. These registers can, of course, not be renamed. Another source of constraints arise from register usage conventions of the application binary interface (ABI), e.g., when function parameters are passed using registers on function calls. Renaming those registers is again not possible. These, and other forms of constraints, have not been

discussed to simplify the presentation, but are highly relevant in practice.

In addition, other forms of dependencies besides those discussed previously might appear in an actual DDG. The respective algorithms to detect cyclic dependencies and update the DDG have to be modified in order to preserve the program's original semantics throughout all code motion transformations. The algorithms presented in the previous sections mainly dealt with dependencies carrying registers in order to simplify the discussion.

Some processors provide instructions with multiple result registers, e.g., instructions yielding the quotient and the remainder of a division. The presented algorithms do not consider this case, but can easily be extended to handle instructions with multiple results. The only difference is that additional dependencies may arise between a parallel copy and the respective register use or definition.

8.3 Experiments

We evaluate our approach for the ST2xx architecture within the production compiler (version 6.5.0) of STMicroelectronics, which is based on the Open64 optimizers[2] and the LAO [38] backend extension. The ST2xx architecture is a 4-way parallel VLIW architecture offering a single load/store unit, i.e., only a single memory access can be performed per cycle. The architecture defines 64 32-bit general purpose registers and 8 single-bit predicate registers, which can be used to control a conditional branch or a `select` operation that conditionally copies one if its two input operands to its output operand.

We applied our DDG-based copy elimination to the integer benchmarks of the SPEC2000 suite. The `252.eon` benchmark is omitted due to lacking C++ support. Also the `253.perlbmk` is not considered since certain system calls required by this benchmark are no longer supported by the ST2xx platform.

The code generator of the ST2xx compiler performs instruction selection, followed by pre-pass instruction scheduling, register allocation, and post-pass instruction scheduling. Pre-pass scheduling conservatively tries to reorder instructions to minimize execution time, while avoiding an increase of register pressure. Since register allocation introduces additional operations, e.g., memory accesses and register copies, a second, more aggressive, instruction scheduling pass produces the final instruction sequence.

Register allocation is performed under SSA by a generic graph coloring register allocator featuring iterated coalescing [51], many copies are thus already eliminated. The remaining copies are induced by ϕ-functions introduced by the conversion to SSA form. The respective copies are converted to regular and cyclic parallel copies, see Section 2.1.1, which are preliminarily placed *on* the CFG edges before the corresponding ϕ-functions. The parallel copies are then assigned to basic blocks using the *Edge Motion* technique of Chapter 7. Finally, our DDG-based copy elimination is performed for each basic block in the program separately. The copy elimination thus effectively executes after register allocation and before post-pass scheduling. A potentially suboptimal ordering of instructions produced by our copy elimination thus is automatically taken care of by post-pass scheduling.

[2] http://www.open64.net/

In our experiments we evaluate the effectiveness of our DDG-based approach with respect to the unmodified *Block Motion* technique of Chapter 7. The experiments focus on the total count of copies after register allocation in comparison to the number of copies remaining after applying either standard *Block Motion* or our approach. In addition, we also compare the reduction in copy costs, by assigning each copy in the program a weight corresponding to the execution frequency of the copy. The weight is computed using standard formulas of the Open64 compiler as $\frac{1}{4}f_{BB}$, where f_{BB} denotes the execution frequency of basic block `BB`. The factor $\frac{1}{4}$ accounts for the potential parallel execution of up to 4 copy operations in a single VLIW. Execution frequencies are estimated using the standard approach of Ball and Larus [5]. The results are compared for each of the 4811 function of the benchmark programs individually. To make the figures easier to read, only functions where either of the two copy elimination strategies was able to eliminate some copy are shown. Furthermore, aggregated results over whole benchmarks are discussed and the runtime behavior of the various benchmarks are compared.

8.3.1 Copy Elimination after Full Coalescing

Our first setup shows the potential for our technique ($\texttt{DDG}_\phi$) after iterated coalescing, where coalescing of ϕ-related variables is enabled. The result is compared with a configuration ($\texttt{BASE}_\phi$), where only *Edge Motion* is performed, and a configuration ($\texttt{BM}_\phi$), where *Block Motion* is performed in addition. Figure 8.11 compares the number of remaining copies, sorted ascending according to the $\texttt{DDG}_\phi$ configuration. The plot shows a data point for every function, where either $\texttt{BM}_\phi$ or $\texttt{DDG}_\phi$ eliminate some copies. In the best case, only 25% of the original copies remain for $\texttt{DDG}_\phi$ (`197.parser`), compared to 48% for $\texttt{BM}_\phi$ (`300.twolf`). On average, depicted by the horizontal lines, over the 461 functions, only 90% of the copies remain for $\texttt{DDG}_\phi$, whereas for $\texttt{BM}_\phi$ 94% remain. Considering that iterated coalescing already delivers much better results than other heuristic coalescing techniques (about 20% in comparison to Brigg's conservative coalescing [51]), these results are surprisingly good. Due to the conversion between parallel copies and permutations, $\texttt{BM}_\phi$ may, in some cases, increase the number of copies. In the worst case, this increase amounts to 49% (`197.parser`), which is explained by an adverse interaction between *Block Motion* and inter-procedural register allocation. The problem here is that *Block Motion* touches caller-saved registers when constructing permutations. This impacts the register assignment at the call sites of the respective function and increases the number of copies.

Figure 8.12 shows the relative costs induced by register-to-register copies, i.e., the sum of the estimated execution frequencies of the copies per function, sorted ascending with respect to the $\texttt{DDG}_\phi$ configuration, which again delivers considerably better results. In the best case, only 0.8%(!) (`254.gap`) of the copy costs remain for $\texttt{DDG}_\phi$, whereas 7% (`176.gcc`) of the costs remain for $\texttt{BM}_\phi$. On average over the 461 functions, only 91% of the costs remain for our new approach, while for $\texttt{BM}_\phi$ 94% of the costs remain.

The number of remaining copies and their total costs over all 4811 functions is shown by Table 8.1. The trend observed previously is again reflected by these numbers, albeit at a reduced magnitude. On average, $\texttt{DDG}_\phi$ eliminates more than 3% of all the copies that could not be eliminated by iterated coalescing, whereas *Block Motion* eliminates just about 2%. For `300.twolf` both approaches per-

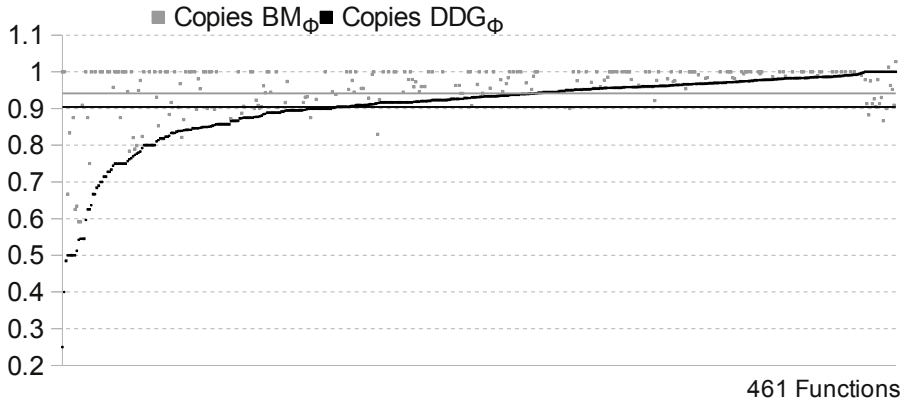


Figure 8.11: Remaining copies relative to the $\texttt{BASE}_\phi$ configuration, per function, after full coalescing. (Lower is better)

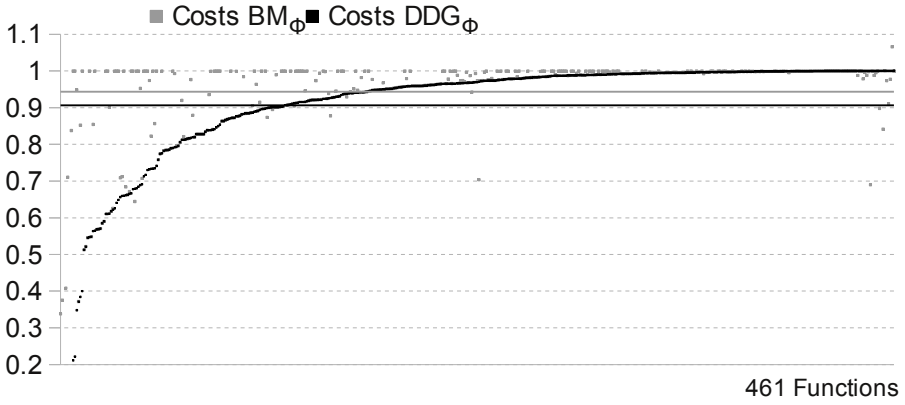


Figure 8.12: Remaining costs relative to the $\texttt{BASE}_\phi$ configuration, per function, after full coalescing. (Lower is better)

	Number of Copies			Total Costs		
Benchmark	$\texttt{BASE}_\phi$	$\texttt{BM}_\phi$	$\texttt{DDG}_\phi$	$\texttt{BASE}_\phi$	$\texttt{BM}_\phi$	$\texttt{DDG}_\phi$
164.gzip	825	797	796	4212710	4211230	4211560
175.vpr	5437	5334	5312	3215486	3215023	3214308
176.gcc	40055	39075	38745	487753	475099	466556
181.mcf	228	227	223	1113	1106	1079
186.crafty	2149	2128	2122	247843	247839	247838
197.parser	3426	3352	3321	482189	363355	362490
254.gap	14446	14304	14129	16154931	16148202	16139236
255.vortex	22796	22722	22708	10922	10909	10898
256.bzip2	673	662	661	4362319	4361304	4361293
300.twolf	5201	4828	4752	1390838	1389218	1389016

Table 8.1: Total number of copies and total copy costs remaining for each benchmark after register allocation with full coalescing. (Lower is better)

form best, eliminating 9% ($\texttt{DDG}_\phi$) respectively 7% ($\texttt{BM}_\phi$) of the copies. In terms of copy costs, `197.parser` shows the best reductions amounting to about 25% for both techniques.

8.3.2 Copy Elimination after *Decoupled* Register Allocation

In our second setup, we *simulate* decoupled register allocation by deactivating the coalescing of ϕ-related variables, i.e., more parallel copies appear. The configurations (names without the ϕ subscript) remain unchanged otherwise. Figure 8.13 shows the remaining copies relative to the base configuration `BASE`. `DDG` performs considerably better, only 73% of the copies remain on average over the 2296 functions, where either `DDG` or `BM` are able to eliminate some copy. For the `BM` configuration, on the other hand, 82% of the copies remain. The medians for the DDG-based and block-motion-based configurations lie at 76% and 83% respectively. In the best case, `DDG` eliminates all copies (`181.mcf`, `186.crafty`, `197.parser`), while for `BM` 11% of the copies remain in the best case (`176.gcc`). As before, *Block Motion* increases the number of copies for certain functions, which amounts to 9% in the worst case (`176.gcc`).

The corresponding reductions in copy costs are shown by Figure 8.14. For `DDG` only 71% of the initial copy costs remain, while 81% of the costs remain for `BM`. Naturally, the copy costs for the functions where `DDG` is able to eliminate all copies become zero as well. The *Block Motion* algorithm is only able to reduce the remaining copy costs to less than 1% for a single function (`186.crafty`). The medians lie at 80% and 88% respectively. For 6 functions of `176.gcc` *Block Motion* increases the copy costs between 2% and 22%.

A comparison of the total number of copies and their respective costs over all 4811 functions for the three configurations is given by Table 8.2. The DDG-based approach, on average, over all benchmarks eliminates 32% of all copies with respect to the `BASE` configuration. The best result is achieved for `164.gzip`, where 45% of the copies are eliminated. The approach performs even better with regard to copy costs, where the reduction amounts to 37% on average. The `254.gap` benchmark here shows the best result, 55% of the copy costs are eliminated. The configuration based on *Block Motion* shows better results than for our first setup. However, it cannot reach the `DDG` configuration. On average over all benchmarks, 21% of the copies and 23% of the copy costs are eliminated. The best results are achieved for `300.twolf` and `197.parser` with reductions of 28% in the number of copies and 42% in the total copy costs respectively. Both configurations appear to have difficulties with the `255.vortex` benchmark, where `DDG` is able to eliminate only 13% and `BM` only 10% of the copies.

Note, however, that the total number of copies and their costs summarized by the table favors benchmarks with larger functions having more copies, which are often easy to eliminate. These large functions may dominate the overall numbers, as depicted by Figure 8.15. The figure shows the accumulated and normalized number of copies eliminated by `DDG` in comparison to the accumulated and normalized number of initial copies from `BASE`. For `BASE`, 414, i.e., less than 10%, out of the 4811 functions contain 50% of the total number of copies. During copy elimination this is even further amplified. For `DDG`, only 214 functions account for 50% of the eliminated copies. For the `BASE` configuration, these functions contain about 37% of the total number of copies of all

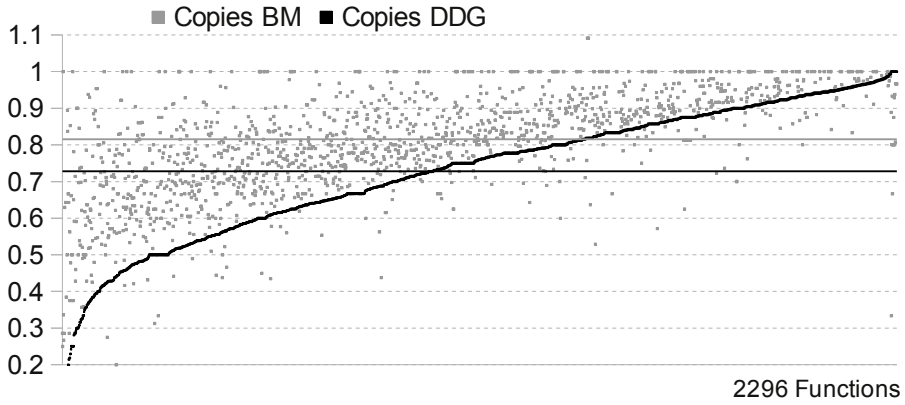


Figure 8.13: Remaining copies relative to the `BASE` configuration, per function, after *decoupled* register allocation. (Lower is better)

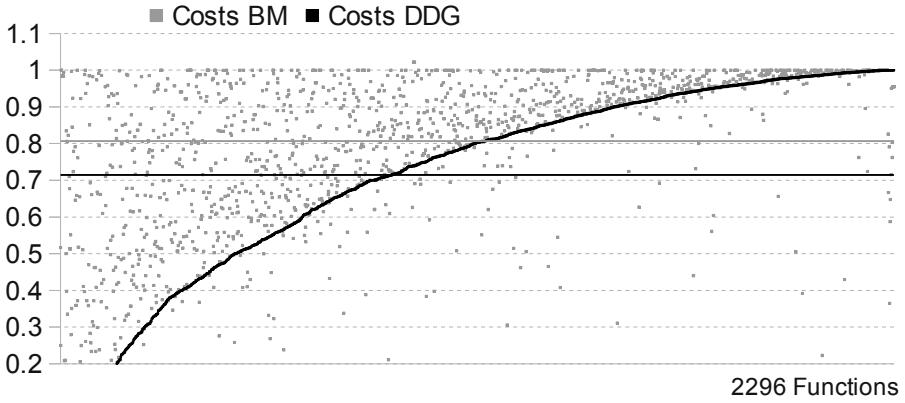


Figure 8.14: Remaining costs relative to the `BASE` configuration, per function, after *decoupled* register allocation. (Lower is better)

	Number of Copies			Total Costs		
Benchmark	`BASE`	`BM`	`DDG`	`BASE`	`BM`	`DDG`
164.gzip	3205	2392	1754	5357735	4290840	4255540
175.vpr	8614	7254	6570	7627027	6163193	5170911
176.gcc	64359	52862	47567	13382491	9736197	7300438
181.mcf	527	404	342	7212	5924	3549
186.crafty	5912	4306	3453	2693734	2134310	1284878
197.parser	6508	5353	4677	1039805	604656	578991
254.gap	30039	24932	20679	67426060	56894894	30334354
255.vortex	27881	25008	24175	15696	13514	12803
256.bzip2	1780	1347	1153	6683255	5120760	5075978
300.twolf	14640	10579	8812	15582390	11381541	9985798

Table 8.2: Total number of copies and total costs remaining for each benchmark after *decoupled* register allocation. (Lower is better)

functions.

8.3.3 Coalescing versus DDG-Based Copy Elimination

Due to the conservative nature of iterated register coalescing we can compare the results of the two experimental setups presented in the previous sub-sections. Figure 8.16 shows the number of remaining copies per function for the `BASE` and `DDG` configurations (after decoupled register allocation) in relation to the $\texttt{BASE}_\phi$ configuration, which performs full coalescing. Disabling the coalescing of ϕ-related variables (`BASE`) leads to a dramatic increase in the number of copies by 87% on average, per function. While 1812 (38%) of the functions do not show any significant increase, 293 (18%) show an increase by a factor of two or more – up to a factor of 49 in the worst case.

Given the local scope of our DDG-based copy elimination, it turns out to be

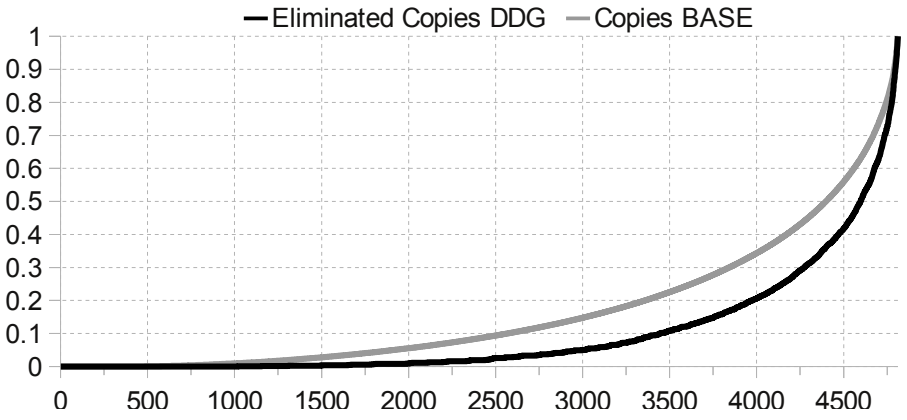


Figure 8.15: Accumulated and normalized number of copies eliminated by `DDG` in comparison to the accumulated and normalized number of copies for the `BASE` configuration, per function, after *decoupled* register allocation.

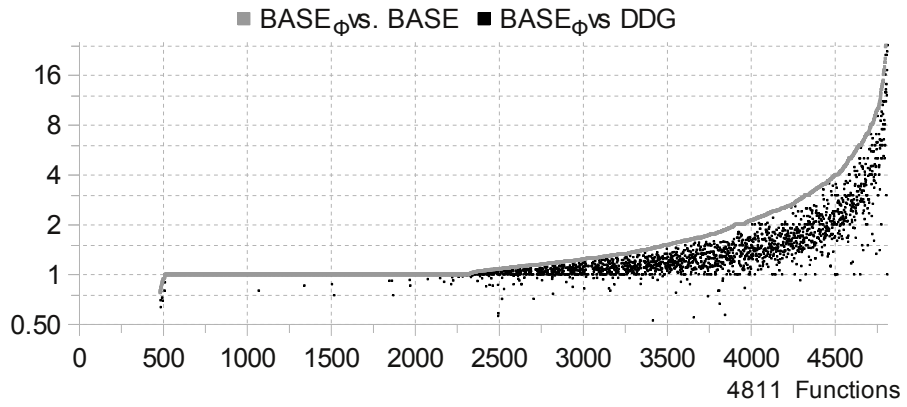


Figure 8.16: Increase in the number of copies relative to the base configuration $\texttt{BASE}_\phi$, per function. (Lower is better)

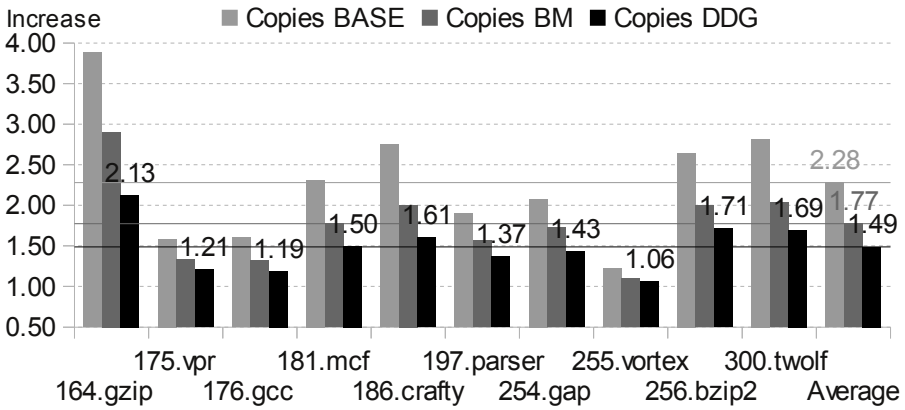


Figure 8.17: Increase in the number of copies relative to the base configuration $\texttt{BASE}_\phi$. (Lower is better)

surprisingly effective. On average, the increase for `DDG` in comparison to $\texttt{BASE}_\phi$ amounts to only 37% per function, i.e., our technique is able to make up for more than half of the losses. Almost half of the functions, 2154 (42%) contain the same amount or less copies than with full coalescing enabled, and only 357 (7%) show increases of a factor of two or more. The worst case increase is also reduced down to a (still high) factor of 24 – only 5 out of all functions show an increase larger than 16 and 27 functions an increase larger than 8.

The results per benchmark follow this trend, as depicted by Figure 8.17. The `255.vortex` benchmark is the least impacted by the disabled coalescing of ϕ-related variables, showing increases of 10% and 6% for `BASE` and `DDG` respectively. The worst results are observed for `164.gzip`, with an increase of a factor of 4 and 2.13 for these two configurations respectively. On average, we thus observe an increase of a factor of 2.28 for the base configuration, compared to an increase of 49% for our DDG-based technique. The standard *Block Motion* approach consistently gives results worse than our new method. The increases in copy costs follow similar trends as the total number of copies.

8.3.4 Runtime Behavior

In a final experiment we further investigate the impact of the DDG-based copy elimination on the runtime behavior of the benchmark programs. Note that for these measurements the `176.gcc` as well as the `253.perlbmk` benchmarks are omitted, because they cannot be executed on the ST2xx platform. The execution data is collected using the cycle-accurate ST2xx simulator. Due to the large size of the benchmark programs and their data sets, we observe that the cache behavior is dominating the execution time by far. The actual gains in execution time are thus negligible. This is not surprising as the number of copy operations in comparison to other code is relatively small. Furthermore, memory accesses and, in particular, cache misses take orders of magnitude long than simple arithmetic and copy operations. In one case the benchmark even spends about 65% of its execution time on servicing instruction and data cache misses. We thus report measurements based on the total number of instructions

executed by the processor normalized to a base configuration – see Table 8.3.

In comparison to the base configuration after *decoupled* register allocation (`BASE`), the DDG-based copy elimination (`DDG`) reduces the total amount of executed instructions by 4% for `256.bzip2` and 3% for `164.gzip`. *Block Motion* (`BM`) gives slightly inferior reductions of up to 2% for `164.gzip` and `254.gap`. On average the two copy elimination techniques reduce the total number of executed instructions by about 1% and 2% respectively. *Block Motion* leads to slightly better improvements in comparison to our technique in only two cases (`181.mcf` and `254.gap`). When full coalescing is performed during register allocation the gains of copy elimination naturally diminish. We thus do not see any relevant improvements for the $\texttt{DDG}_\phi$ and $\texttt{BM}_\phi$ configurations with respect to $\texttt{BASE}_\phi$. The largest improvement is in the order of 0.1% for `175.vpr` and `186.craft`, slightly favoring the DDG-based copy elimination.

It is interesting to note that both copy elimination techniques increase the number of executed `nop` operations, i.e., instructions without side effect that are sometimes required to ensure latency and alignment constraints. In the worst case (`181.mcf`) this increase amounts to a factor of 2.69(!) for both, the `DDG` and `BM` configurations, while on average the increase amounts to 34% and 22% respectively. One might suspect an adverse interaction with instruction scheduling, which has fewer copy operations available to fill holes in the schedule. Or in other words, one might suspect that copy operations are simply replaced by `nops` during scheduling. However, we could not confirm this intuition when examining the final code produced after post-pass scheduling. Here, we observe on average a minimal increase in the number of `nop` operations for `BM` (1.0003), while `DDG` even yields a reduction by 0.5%. This is also reflected by corresponding code size reductions. Overall we conclude that our DDG-based approach leads to slightly denser instruction schedules, while at the same time the total number of executed instructions is reduced.

8.4 Related Work

A standard approach to copy elimination during register allocation is register coalescing [32]. The problem, however, is that coalescing might increase

Benchmark	`BASE`	`BM`	`DDG`	$\texttt{BASE}_\phi$	$\texttt{BM}_\phi$	$\texttt{DDG}_\phi$
164.gzip	1.00	0.98	**0.97**	1.00	1.00	1.00
175.vpr	1.00	1.00	1.00	1.00	1.00	1.00
181.mcf	1.00	**0.98**	0.99	1.00	1.00	1.00
186.crafty	1.00	0.99	0.99	1.00	1.00	1.00
197.parser	1.00	0.99	**0.98**	1.00	1.00	1.00
254.gap	1.00	**0.98**	0.99	1.00	1.00	1.00
255.vortex	1.00	0.99	0.99	1.00	1.00	1.00
256.bzip2	1.00	0.99	**0.96**	1.00	1.00	1.00
300.twolf	1.00	0.99	0.99	1.00	1.00	1.00
Average	1.00	0.99	**0.98**	1.00	1.00	1.00

Table 8.3: Total number of instructions executed by a ST2xx processor for each benchmark relative to the base configuration without *Block Motion* and DDG-based copy elimination. (Lower is better)

the register pressure and may lead to additional spilling. Heuristics thus try to find a good balance [26, 51, 81] between overly aggressive coalescing and spilling. Bouchez et al. showed that various variants of the coalescing problem are NP-complete [17]. Grund and Hack [53] proposed an optimal solution to the coalescing problem using linear programming. Our approach does not require an interference graph and is thus better suited for decoupled approaches.

Similar to Brisk et al. [28], Braun et al. proposed techniques to eliminate copies during register assignment [22] in decoupled register allocation. The basic idea is to bias the assignment such that copy-related variables are assigned the same register. In another approach, proposed by Buchwald et al. [29], the register assignment is modeled as a non-linear optimization problem (partitioned Boolean quadratic programming [40]), that can be solved heuristically or optimally using branch-and-bound.

Linear scan register allocators [89], similarly try to avoid the explicit construction of an interference graph. Wimmer and Mössenböck proposed *register hints* [105] to propagate information on copy-related variables during register allocation and, if possible, assign them to the same register. This is similar to biased register assignment techniques.

Pereira and Palsberg proposed punctual coalescing [87] to model the coalescing and assignment problem in a puzzle-based register allocator [85]. Given a valid puzzle assignment for an instruction, it seeks a valid assignment for a succeeding instruction, while avoiding useless copy operations. Since punctual coalescing operates locally between two adjacent instructions, its scope is even more restricted than our basic-block-local copy elimination.

These approaches are complementary to our work, it thus might prove interesting to combine their respective strengths: the register assignment optimizes *global*, ϕ-related live-ranges spanning multiple basic blocks, while the DDG-based copy elimination handles basic-block-local assignment mismatches.

As proposed in this work, local recoloring after the assignment can be performed. Hack and Goos proposed a recoloring technique on interference graphs [57] that tries to fix-up mismatches locally, which are then propagated throughout the entire IG. Parallel Copy Motion, briefly introduced in Section 8.1, aims at finding a good placement of copies in the control flow graph and within basic blocks. The main advantage of this technique, similar to our technique, is that the construction of an IG is avoided. It is thus best suited for dynamic code generators, such as JIT compilers.

In contrast to this work, none of the previous approaches exploits the reordering of instructions to eliminate copies.

8.5 Conclusion

This work presents a new algorithm to eliminate register-to-register copies after register allocation based on the idea of local recoloring. Our approach operates on data dependence graphs and thus has the unique ability to reorder instructions, if this appears to be profitable.

Our experiments show that even after traditional copy elimination, using a state-of-the-art coalescing algorithm, our approach is able to eliminate additional copies. The approach also proves very powerful as an alternative to coalescing in the context of decoupled register allocation. In both settings, our

DDG-based algorithm offers considerable improvements with respect to Parallel Copy Motion.

A limitation of our approach, in comparison to traditional coalescing, is the local scope, i.e., we currently limit our approach to basic blocks. It should be fairly easy to extend the presented algorithms to operate on data dependence graphs of extended basic blocks, super blocks, or even traces [62, 44]. The increased scope of the optimization might then improve its effectiveness – in particular with respect to copy costs. It might also be interesting to investigate a closer intertwining between *Edge Motion* and our technique. For instance, it might be profitable to process the basic blocks in post-order using our technique, while propagating the remaining copies to neighboring basic blocks along control-flow edges.

We could, furthermore, exploit copies more aggressively after copy elimination in the final instruction scheduling pass. Uses of a register can be scheduled freely before or after a related copy involving the same register, if no dependencies exist between the use and the copy. The scheduler might even introduce copies to avoid costly stalls. It might thus be interesting to extend our approach to operate in concert with instruction scheduling.

Part V

Conclusion

Chapter 9

Conclusion

In this thesis, we demonstrated, based on a deep empirical evaluation, that complex algorithms are not required to cope with aliasing, application binary interface (ABI), or encoding constraints in decoupled register allocators. We showed that handling such constraints is possible with another notion of register pressure, with post-processing phases, or with better-placed split points. All these contributions use the elegant framework of decoupled register allocation.

9.1 Contributions

9.1.1 Spilling

Chapter 3 introduced a new integer linear programming (ILP) formulation of the spilling problem. This formulation is general enough to emulate, to our knowledge, all existing exact approaches. In other words, it subsumes them. Using this formulation, we studied optimal spilling, with respect to static spill cost, the state-of-the-art spilling metric, in the context of static single assignment (SSA) form. We showed that it is difficult – although possible – to capture the effects of ϕ-functions in the formulation, thus, a fortiori, in a heuristic. The ϕ-functions are virtual split points and they should be handled as such, otherwise the static spill cost may be degraded by up to 40% and the runtime by up to 10% for the best model with and without SSA. In particular, an optimal spiller should be able to transfer values through memory copies, i.e., without requiring any registers, to match the performance of the same program without SSA. On a reduced instruction set computing (RISC) architecture, this is possible if and only if both values share the same memory location. Therefore, when SSA comes in play, the spiller should be able to solve a coalescing problem for the memory slots and it must solve it optimally to reach the optimal spilling solution.

As coalescing is a hard problem, we did not want to include it in the spilling problem, this would have blown up the number of ILP variables and constraints. Instead, we proposed two approaches to approximate the memory coalescing problem. The first approach, optimistic, considers that all move-related memory locations, i.e., all memory slots of variables connected by a (parallel) copy or a ϕ-function in the original program, can be coalesced, thus memory-to-memory copy is free. The second approach, pessimistic, considers that all move-related variables that do not interfere in the original program have their memory slots coalesced, otherwise their memory slots interfere. Both approaches proved to

compete with their non-SSA counterpart, which offers a simple model to overcome the difficulties implied by SSA for spilling. Moreover, we showed that, when rematerialization is enabled, configurations using SSA achieved better performances as this form gives more information on the variables. Similar results may be reachable for other configurations but with more complex analysis.

Finally, in this chapter, we observed that the static spill-cost model does not capture the important feature of modern architectures, in particular the fact that the latency of `loads` depends on the distance from the next use, thus producing mitigate results for runtime performance.

Following this observation, we reviewed, in Chapter 4, the existing spilling criteria and the related spillers. We identified the limitations of these criteria and their misuses in the spillers. We proposed several extensions to increase their scope. We defined a better profitability metric for further-first-based heuristics and a latency-based model for cost-based approaches. Moreover, using our ILP formulation, we validated empirically some simplifying assumptions that may help to derive good spilling heuristics. In particular, we validated that, in a first approximation, `store` instructions can be blocked at definition points and be considered as free. However, they are still minimized, i.e., no useless `store` should be inserted. This way, spillers can focus on improving the cost of `loads`. For `loads`, we proposed to force them to be placed just before the related uses but not necessarily before all uses. Of course, this can lead to some bad cases but interestingly, on average, this approach, coupled with the `store` simplification, does not impact the runtime when followed by a latency optimization as we proposed. Thus, a very simple model, `store` at definitions and free, `load` at uses and costly, yields runtime performances comparable to optimal, with respect to spillers based on a static spill cost. This model may be of interest for a just-in-time (JIT) compiler.

9.1.2 Coloring

In Chapter 5, we presented an extension of the interference graph (IG) model to deal with encoding and ABI constraints. This extension features *antipathies*. Antipathies look like the affinities used in the coalescing optimization but, unlike affinities, they represent a dislike between two variables. These entities provide a convenient way to model weak interferences, i.e., interferences between variables but that may be broken at some cost. In our case, the cost represents the weight of `move` instructions to be inserted to *repair* this interference.

This extension has two main advantages. First, it does not require any additional split points than the ones provided by SSA, but still it produces an IG with the nice properties of SSA. This way, it eliminates the pre-processing phases needed to perform additional live-range splitting, e.g., liveness analysis, split points insertion, SSA reconstruction. Second, it is compatible with existing graph-coloring-based approaches, as antipathies are modeled with affinities of negative weight. We proposed three strategies to take advantage of this model in existing approaches. These strategies achieve different degrees of quality of the generated code, depending on the implementation effort made to handle this extension. In all cases, this effort is kept light. These strategies are:

Freeze Ignore the antipathies during the simplification process and take them into account when choosing the color.

Dummy Node After graph building and prior to coloring, replace each antipathy by an affinity and an interference using an additional dummy node.

Conservative Insert a new "coalescing" rule for antipathies and replace them by interferences when it is conservative to do so.

We demonstrated that this model and its repairing counterpart produce code whose quality is comparable to the approaches with extensive live-ranges splitting, but without the need for this splitting. We then showed how a similar method, based on repairing, can be applied on scan-based approaches with our fast register allocator *tree-scan* . Using this allocator, we evaluated the impact of different bias techniques on the code quality, the compile time, the memory footprint, and the runtime. Tree-scan proved to be a very efficient allocator whose produced code quality, memory footprint, and compile time can be tuned with the different bias techniques according to a time budget. In particular, it produces code whose runtime is within 2% of a decoupled iterated register coalescer (IRC) with extensive split points, in 12x less compile time and almost 30x smaller memory footprint.

In Chapter 6, we presented a new spilling criterion to handle aliasing and still be able to decouple the register allocation. We defined a new form of live-ranges splitting, the *semi-elementary* form. This form provides the first light program representation for decoupled register allocation when aliasing is involved while previous approaches rely on split-everywhere strategies (i.e., possibly a split on each program point). We demonstrated the interest of such a form with existing graph-coloring-based allocation. The size of this representation yields smaller graphs, which have a smaller memory footprint and are solved faster, and produces better results as heuristics have difficulties to cope with large graphs. Although graph coloring is not intended to be used in JIT compilers, these results are interesting for regular compilers, in particular those featuring aggressive approaches and for which the problem was previously intractable because of the size of the representation. We believe that this improvement will allow to further investigate this kind of approach, leading to a better problem understanding and possibly heuristics suitable for JIT compilation.

9.1.3 Post Phases

In Chapter 7, we proposed an elegant formalism, called *parallel copy motion*, that works on register-allocated codes. Parallel copy motion is useful to perform region recoloring (a re-allocation of variables by register permutations). We demonstrated its interest with two different applications. First, allocated codes, in particular those produced by decoupled or fast allocators, often contain (parallel) copies that are, from the semantics of SSA, to be placed on the control flow graph (CFG) edges. Prior to our work, to be materialized in the code, these instructions required edge splitting, i.e., the addition of a basic block on the edge. We showed how parallel copy motion can be used to move these instructions from the CFG edges, possibly making this block empty and thus useless. This proved to be beneficial both for the code quality and the runtime performances. Second, parallel copy motion turned out to be useful also to remove some (parallel) copy instructions that can remain anywhere in the code at the end of the allocation process, not necessarily on CFG edges.

In Chapter 8, we further extended our parallel copy motion framework on data dependence graphs (DDGs). Working directly with this information on

data dependences enables to reschedule the code as well as performing region recoloring to eliminate copy instructions.

Both methods are not needed to produce a correct allocated code. However, they improve its quality. Moreover, each method is composed by a sequence of transformations (moving up, moving down), whose order can be tweaked via heuristics. Each of these transformations produces a valid output. Therefore, we believe that these methods offer the flexibility required to be used in a JIT compiler. Indeed, they can be used to increase the performance until a time budget or a threshold is reached between each transformation.

9.2 Perspectives

9.2.1 Spilling

We believe that our ILP formulation provides a good framework to capture the spilling problem with fixed scheduling. Using this framework, it is easy to explore different spill costs and restrictions. From our point of view, it will not pay off to invest in a more sophisticated formulation, unless new results help to model the memory coalescing problem.

We are currently working on a fast spilling heuristic that will use either our latency model or the proposed simplifying assumptions (or both). Using these simplifications, we believe that with a proper live-range splitting, as introduced in Chapter 4, a spill-everywhere, thus simple, approach can be used. Moreover, we want to continue our research on defining a good spilling criterion. Indeed, we gave some evidence that helping the scheduler is a good idea for runtime performances, but we do not know yet how to model that. This problem is even more complex in the case of very-long instruction word (VLIW) machines, like our supplied target, as bundles may have a huge impact on the runtime performances. In particular, a well-placed `load` instruction, i.e., whose latency is completely hidden and that fills a hole in an existing bundle, can be cheaper (actually even free!) than a badly-placed `move` instruction. Indeed, if all bundles are full, a new one will be created. In other words, on these architectures, the assumptions that `load`s are, in first approximation, more expensive than `move`s may not be true. We want to investigate further the implications of this fact.

In a longer-term perpective, we want to explore the coupling of a scheduler and a spiller with a spill-aware scheduling or a scheduling-aware spilling, first in a static compiler, then in a JIT compiler.

9.2.2 Coloring

In this thesis, we provided a complete framework to deal with encoding and ABI constraints in a wide range of approaches. In particular, without aliasing, we believe that our coloring approach, tree-scan, will be difficult to overcome on both the compile-time and the memory footprint. Moreover, the code quality of its generated code, when combined with bias technique, competes with the best known coloring approaches. This makes this approach appealing even for static compilers.

However, our approach of aliasing constraints remained focused on graph-coloring-based approaches. We are working on a tree-scan allocator that would

handle both kind of constraints. Our approach is a generalization of the puzzle solver approach of Pereira and Palsberg [85] and yields better spilling decisions.

We think that our antipathy model has an applicability potential that has not been exploited yet and that should be investigated. For instance, when working on our post latency optimization for the spilling problem, we saw that it would have been possible to bias the coloring using this model, so that the resulting allocated code would have been easier to schedule with the regular scheduler while hidding the latency.

9.2.3 Post Phases

Our parallel copy motion framework nicely solves the problem of the copies on the edges. Moreover, as a by-product, our study has empirically proved that it is generally a bad idea to leave these copies on the CFG edges. In terms of correctness and cleanless of the approach, we believe it is not worth trying to do better. However, our proposed methods can be tweaked to produce even better code quality in terms of runtime. In particular, we presented a DDG-based copy motion that works at the basic-block level. The objective function was to minimize the number of `move`s, since all places in a basic block have the exact same cost. We want to extend this approach in two orthogonal directions:

- The scope of the considered DDGs.
- Scheduling-aware elimination of `move`s.

For the first point, we have to refine the cost model and we may have add some compensation code on edges. For the second point, we would look for a better scheduling and not just a feasible one. Indeed, removing an instruction can actually increase the schedule length on VLIW architectures.

List of Publications

[a] Florent Bouchez, Quentin Colombet, Alain Darte, Christophe Guillon, and Fabrice Rastello. Parallel copy motion. In *13th International Workshop on Software & Compilers for Embedded Systems (SCOPES'10)*, pages 1–10, St. Goar, Germany, June 2010. ACM Press.

[b] Florian Brandner and Quentin Colombet. Copy elimination on data dependence graphs. In *Symposium on Applied Computing (SAC'12)*, Trento, Italy, March 2012. ACM Press.

[c] Quentin Colombet, Benoit Boissinot, Philip Brisk, Sebastian Hack, and Fabrice Rastello. Graph coloring and treescan register allocation using repairing. In *International Conference on Compilers, Architectures, and Synthesis of Embedded Systems (CASES'11)*, Taipei, Taiwan, October 2011. IEEE Computer Society.

[d] Quentin Colombet, Florian Brandner, and Alain Darte. Studying optimal spilling in the light of ssa. In *International Conference on Compilers, Architectures, and Synthesis of Embedded Systems (CASES'11)*, Taipei, Taiwan, October 2011. IEEE Computer Society.

[e] André Tavares, Quentin Colombet, Mariza Bigonha, Christophe Guillon, Fernando Pereira, and Fabrice Rastello. Decoupled graph-coloring register allocation with hierarchical aliasing. In *14th International Workshop on Software & Compilers for Embedded Systems (SCOPES'11)*, pages 1–10, St. Goar, Germany, June 2011. ACM Press.

Bibliography

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11, New York, NY, USA, 1988. ACM Press.

[2] C.S. Ananian. The static single information form. *Technical Report MIT-LCS-TR-801*, 1999.

[3] A. W. Appel and L. George. Optimal spilling for CISC machines with few registers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*, pages 243–253, Snowbird, USA, June 2001. ACM Press.

[4] A. W. Appel and J. Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition, 2002.

[5] T. Ball and J. R. Larus. Branch prediction for free. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'93)*, pages 300–313, Albuquerque, USA, June 1993. ACM.

[6] R. Barik. *Efficient optimization of memory accesses in parallel programs*. PhD thesis, Rice University, 2009.

[7] L. A. Belady. A study of replacement algorithms for a virtual storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[8] D. Bernstein, M. Golumbic, Y. Mansour, R. Pinter, D. Goldin, H. Krawczyk, and I. Nahshon. Spill code minimization techniques for optimizing compliers. In *Proceedings of the SIGPLAN '89 Conference on Programming language design and implementation*, pages 258–263. ACM Press, 1989.

[9] Miklós Biró, Mihály Hujter, and Zsolt Tuza. Precoloring extension. I. interval graphs. *Discrete Mathematics*, 100(1âĂŞ3):267–279, 1992.

[10] R. Bodík, R. Gupta, and M. L. Soffa. Load-reuse analysis: design and evaluation. *SIGPLAN Not.*, 34(5):64–76, May 1999.

[11] B. Boissinot. *Towards an SSA based compiler back-end: some interesting properties of SSA and its extensions*. PhD thesis, École normale supérieure de Lyon, 2011.

[12] B. Boissinot, F. Brandner, A. Darte, B. de Dinechin, and F. Rastello. A non-iterative data-flow algorithm for computing liveness sets in strict SSA programs. *International Symposium on Programming Languages and Systems*, pages 137–154, 2011.

[13] B. Boissinot, A. Darte, B. Dupont de Dinechin, C. Guillon, and F. Rastello. Revisiting out-of-SSA translation for correctness, code quality, and efficiency. In *International Symposium on Code Generation and Optimization (CGO'09)*. IEEE Computer Society Press, 2009.

[14] B. Boissinot, S. Hack, D. Grund, B. Dupont de Dinechin, and F. Rastello. Fast liveness checking for SSA-form programs. In *CGO'08: proceedings of the sixth annual ieee/acm international symposium on code generation and optimization*, pages 35–44, New York, NY, USA, 2008. ACM.

[15] F. Bouchez. *A Study of Spilling and Coalescing in Register Allocation as Two Separate Phases*. PhD thesis, École normale supérieure de Lyon, April 2009.

[16] F. Bouchez, A. Darte, C. Guillon, and F. Rastello. Register allocation and spill complexity under SSA. Technical Report RR2005-33, LIP, ENS-Lyon, France, August 2005.

[17] F. Bouchez, A. Darte, and F. Rastello. On the complexity of register coalescing. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 102–114, Washington, DC, USA, mar 2007. IEEE Computer Society Press. Best paper award.

[18] F. Bouchez, A. Darte, and F. Rastello. On the complexity of spill everywhere under SSA form. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'07)*, pages 103 – 112, San Diego, 2007.

[19] F. Bouchez, A. Darte, and F. Rastello. Advanced conservative and optimistic register coalescing. In *CASES'08: Proceedings of the 2008 international conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 147–156, New York, NY, USA, 2008. ACM.

[20] 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 2006, Fifth Annual Workshop on Duplicating, Deconstructing, and Debunking, part of ISCA-33*, Boston, MA, June 2006.

[21] M. Braun and S. Hack. Register spilling and live-range splitting for SSA-form programs. In *Compiler Construction 2009*, volume 5501 of *LNCS*, pages 174–189. Springer-Verlag, 2009.

[22] M. Braun, C. Mallon, and S. Hack. Preference-Guided Register Assignment. In *Compiler Construction 2010*, volume 6011 of *Lecture Notes In Computer Science*, pages 205–223. Springer, 2010.

[23] P. Briggs. *Register allocation via graph coloring*. PhD thesis, Rice university, April 1992.

[24] P. Briggs, K. D. Cooper, T. J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software: Practice and Experience*, 28(8):859–881, 1998.

[25] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proceedings of the conference on Programming language design and implementation*, pages 275–284. ACM Press, 1989.

[26] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[27] P. Brisk, F. Dabiri, J. Macbeth, and M. Sarrafzadeh. Polynomial time graph coloring register allocation. In *14th International Workshop on Logic and Synthesis*, June 2005.

[28] P. Brisk, A. K. Verma, and P. Ienne. An optimistic and conservative register assignment heuristic for chordal graphs. In *Proc. of the Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '07, pages 209–217. ACM, 2007.

[29] S. Buchwald, A. Zwinkau, and T. Bersch. SSA-based register allocation with PBQP. In *Proc. of the Conf. on Compiler Construction*, pages 42–61. Springer, 2010.

[30] Z. Budimlić, K. D. Cooper, T. Harvey, K. Kennedy, T. Oberg, and S. Reeves. Fast copy coalescing and live range identification. In *Proceedings of the ACM Sigplan Conference on Programming Language Design and Implementation (PLDI'02)*, pages 25–32, Berlin, Germany, June 2002. ACM Press.

[31] D. Callahan and B. Koblenz. Register allocation via hierarchical graph coloring. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 192–203, New York, NY, USA, 1991. ACM.

[32] G. J. Chaitin. Register allocation & spilling via graph coloring. In *ACM SIGPLAN Symposium on Compiler Construction (CC'82)*, volume 17(6) of *SIGPLAN Notices*, pages 98–105, 1982.

[33] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, January 1981.

[34] F. C. Chow and J. L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(4):501–536, Oct. 1990.

[35] K. D. Cooper and L. Taylor Simpson. Live range splitting in a graph coloring register allocator. In *Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*, pages 174–187. Springer Verlag, 1998.

[36] K. D. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2004.

[37] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[38] B. Dupont de Dinechin, F. de Ferrière, C. Guillon, and A. Stoutchinin. Code generator optimizations for the ST120 DSP-MCU core. In *Proc. of the Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '00, pages 93–102. ACM, 2000.

[39] D. Ebner, B. Scholz, and A. Krall. Progressive spill code placement. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'09)*, pages 77–86. ACM Press, 2009.

[40] E. Eckstein. *Code Optimization for Digital Signal Processors*. PhD thesis, Institut für Computersprachen, Technische Universität Wien, November 2003.

[41] J. Fabri. Automatic storage optimization. In *Proceedings of the SIGPLAN symposium on Compiler construction*, pages 83–91, 1979.

[42] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. Lx: A technology platform for customizable VLIW embedded processing. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 203–213. ACM, June 2000.

[43] M. Farach-Colton and V. Liberatore. On local register allocation. In *SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 564–573, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.

[44] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, 1981.

[45] *SC140 DSP Core Reference Manual*. Freescale Semiconductor, Inc., 2005.

[46] R. A. Freiburghouse. Register allocation via usage counts. *Commun. ACM*, 17(11):638–642, November 1974.

[47] S. M. Freudenberger, T. R. Gross, and P. G. Lowney. Avoidance and suppression of compensation code in a trace scheduling compiler. *ACM Transactions on Programming Languages and Systems*, 16(4):1156–1214, 1994.

[48] C. Fu and K. Wilken. A faster optimal register allocator. In *ACM/IEEE International Symposium on Microarchitecture (MICRO'35)*, pages 245–256, Istanbul, Turkey, 2002. IEEE Computer.

[49] G. Gao, J. Amaral, J. Dehnert, and R. Towle. The SGI Pro64 compiler infrastructure. In *Tutorial, International Conference on Parallel Architectures and Compilation Techniques*, 2000.

[50] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[51] L. George and A. W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.

[52] D.W. Goodwin and K.D. Wilken. Optimal and near-optimal global register allocation using 0-1 integer programming. *Software: Practice and Experience*, 26(8):929 – 965, 1996.

[53] D. Grund and S. Hack. A fast cutting-plane algorithm for optimal coalescing. In *Proc. of the Conf. on Compiler Construction*, CC'07, pages 111–125. Springer, 2007.

[54] J. Guo, M. Jesús Garzarán, and D. Padua. The power of belady's algorithm in register allocation for long basic blocks. In *Languages and Compilers for Parallel Computing*, volume 2958/2004 of *Lecture Notes in Computer Science*, pages 374–390. Springer Berlin / Heidelberg, 2004.

[55] S. Hack. *Register Allocation for Programs in SSA Form.* PhD thesis, Univ. Karlsruhe, October 2007.

[56] S. Hack and G. Goos. Optimal register allocation for SSA-form programs in polynomial time. *Information Processing Letters*, 98(4):150–155, May 2006.

[57] S. Hack and G. Goos. Copy coalescing by graph recoloring. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 227–237, New York, NY, USA, 2008. ACM.

[58] S. Hack, D. Grund, and G. Goos. Register allocation for programs in SSA form. In *International Conference on Compiler Construction (CC'06)*, volume 3923 of *LNCS*. Springer, 2006.

[59] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, third edition, 2003.

[60] U. Hirnschrott, A. Krall, and B. Scholz. Graph coloring vs. optimal register allocation for optimizing compilers. In *JMLC*, pages 202–213, 2003.

[61] L. P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd. Index register allocation. *J. ACM*, 13(1):43–61, 1966.

[62] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The Superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1-2):229–248, 1993.

[63] A. B. Kempe. On the Geographical Problem of the Four Colours. *American Journal of Mathematics*, 2(3):193–200, September 1879.

[64] K. Knobe and K. Zadeck. Register allocation using control trees. Technical Report No. CS-92-13, Brown University, 1992.

[65] D. R. Koes and S. C. Goldstein. A global progressive register allocator. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*, pages 204–215, San Diego, June 2006.

[66] D. R. Koes and S. C. Goldstein. Register allocation deconstructed. In *12th International Workshop on Software & Compilers for Embedded Systems (SCOPES'09)*, pages 21–30. ACM Press, 2009.

[67] T. Kong and K. D. Wilken. Precise register allocation for irregular architectures. In *MICRO*, pages 297–307. IEEE, 1998.

[68] C. Lattner and V. S. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE, 2004.

[69] J. K. Lee, J. Palsberg, and F. M. Q. Pereira. Aliased register allocation. In *ICALP*, 2007.

[70] A. Leung and L. George. Static single assignment form for machine code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*, pages 204–214. ACM Press, 1999.

[71] V. Liberatore, M. Farach-Colton, and U. Kremer. Evaluation of algorithms for local register allocation. In *8th International Conference on Compiler Construction (CC'99), held as part of ETAPS'99*, volume 1575 of *Lecture Notes in Computer Science*, pages 137–152, Amsterdam, The Netherlands, March 1999. Springer Verlag.

[72] F. Lu, L. Wang, X. Feng, Z. Li, and Z. Zhang. Exploiting idle register classes for fast spill destination. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 319–326, New York, NY, USA, 2008. ACM.

[73] G. Lueh, T. Gross, and A. Adl-Tabatabai. Fusion-based register allocation. *ACM Transactions on Programming Languages and Systems*, 22(3):431–470, 2000.

[74] C. May. The parallel assignment problem redefined. *IEEE Transactions on Software Engineering*, 15(6):821–824, 1989.

[75] R. Morgan. *Building an Optimizing Compiler*. Elsevier Science, 1998.

[76] H. Mössenböck and M. Pfeiffer. Linear scan register allocation in the context of SSA form and register constraints. In *International Conference on Compiler Construction (CC'02)*, volume 2304 of *LNCS*, pages 229–246. Springer, 2002.

[77] V. K. Nandivada, F. Pereira, and J. Palsberg. A framework for end-to-end verification and evaluation of register allocators. In *SAS*, pages 153–169. Springer, Kongens Lyngby, Denmark, August 2007.

[78] C. Norris and L. L. Pollock. Register allocation over the program dependence graph. *SIGPLAN Notices*, 29(6):266–277, 1994.

[79] R. Odaira, T. Nakaike, T. Inagaki, H. Komatsu, and T. Nakatani. Coloring-based coalescing for graph coloring register allocation. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 160–169, New York, NY, USA, 2010. ACM.

[80] J. Park and S. Moon. Optimistic register coalescing. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT'98)*, pages 196–204. IEEE Press, 1998.

[81] J. Park and S. Moon. Optimistic register coalescing. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, 26(4), 2004.

[82] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design - The Hardware/Software Interface*. Morgan Kaufmann, 5th edition, 2012.

[83] F. Pereira. *Register Allocation by Puzzle Solving*. PhD thesis, UCLA, University of California, Los Angeles, 2008.

[84] F. M. Q. Pereira and J. Palsberg. Register allocation via coloring of chordal graphs. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS'05)*, pages 315–329, Tsukuba, Japan, November 2005.

[85] F. M. Q. Pereira and J. Palsberg. Register allocation by puzzle solving. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 216–226, New York, NY, USA, 2008. ACM.

[86] F. M. Q. Pereira and J. Palsberg. SSA elimination after register allocation. In *18th International Conference on Compiler Construction (CC'09)*, volume 5501 of *LNCS*, pages 158–173, York, UK, March 2009. Springer.

[87] F. M. Q. Pereira and J. Palsberg. Punctual coalescing. In *Proceedings of the International Conference on Compiler Construction*, CC'10, pages 165–184. Springer, 2010.

[88] M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: a system for fast, flexible, and high-level dynamic code generation. *SIGPLAN Notices*, 32(5):109–121, 1997.

[89] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.

[90] G. Ramalingam. On loops, dominators, and dominance frontiers. *ACM Transactions on Programming Languages and Systems*, 24(5):455–490, September 2002.

[91] Fabrice Rastello, Francois de Ferrière, and Christophe Guillon. Optimizing translation out of SSA using renaming constraints. In *International Symposium on Code Generation and Optimization (CGO'04)*, pages 265–276. IEEE Computer Society Press, March 2004.

[92] L. Rideau, B. P. Serpette, and X. Leroy. Tilting at windmills with Coq: Formal verification of a compilation algorithm for parallel moves. *Journal of Automated Reasoning*, 40(4):307–326, 2008.

[93] H. Rong. Tree Register Allocation. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 67–77. ACM, 2009.

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

[95] V. Sarkar and R. Barik. Extended linear scan: An alternate foundation for global register allocation. In *International Conference on Compiler Construction (CC'07)*, volume 4420 of *LNCS*, pages 141–155. Springer, 2007.

[96] B. Scholz and E. Eckstein. Register allocation for irregular architectures. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems: software and compilers for embedded systems*, LCTES/SCOPES '02, pages 139–148, New York, NY, USA, 2002. ACM.

[97] R. Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *J. ACM*, 17(4):715–728, October 1970.

[98] J. Singer. Static program analysis based on virtual register renaming. *Technical Report UCAM-CL-TR-660*, February 2006.

[99] M. D. Smith, N. Ramsey, and G. Holloway. A generalized algorithm for graph-coloring register allocation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 277–288, New York, NY, USA, 2004. ACM.

[100] V. C. Sreedhar, R. D.-C. Ju, D. M. Gillies, and V. Santhanam. Translating out of static single assignment form. In *6th International Symposium on Static Analysis (SAS'99)*, volume 1694 of *LNCS*, pages 194–210. Springer-Verlag, 1999.

[101] Y. N. Srikant and P. Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2nd edition, 2007.

[102] *TMS320C5x User's Guide*. Texas Instrument, 2006.

[103] O. Traub, G. Holloway, and M. D. Smith. Quality and speed in linear-scan register allocation. *SIGPLAN Not.*, 33(5):142–151, 1998.

[104] C. Wimmer and M. Franz. Linear scan register allocation on ssa form. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 170–179. ACM, 2010.

[105] C. Wimmer and H. Mössenböck. Optimized interval splitting in a linear scan register allocator. In *1st ACM/USENIX International Conference on Virtual Execution Environments (VEE)*, pages 132–141, 2005.

[106] B. Yang, S. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcioglu, and E. Altman. Latte: A java vm just-in-time compiler with fast and efficient register allocation. *Parallel Architectures and Compilation Techniques, International Conference on*, 0:128, 1999.

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

Appendix A

Appendix

A.1 Coloring with Encoding Constraints

This section reports the detailed numbers of Chapter 5.

Table A.1: Execution time of the generated code: 32 registers

	IRC	IRC Split	Dummy Nodes	Freeze	Conser.	Pref.	H	HC	HR	HCR	HW	HA	HAC	HAR	HAM	HARC	HARCS	HACM	HARCM	HARM
gzip	1	0.91	0.9	0.91	0.91	0.95	0.92	0.94	0.91	0.93	0.89	0.89	0.9	0.87	0.89	0.86	0.92	0.9	0.88	0.87
vpr	1	0.99	0.99	1	0.99	1.02	1.03	1.03	1.03	1.03	1.02	1	1	1.01	1.01	0.99	1.04	1	0.99	1.01
mcf	1	0.95	0.95	0.95	0.95	0.96	0.99	0.98	0.95	0.94	0.95	0.95	0.95	0.92	0.92	0.94	0.97	0.91	0.9	0.91
crafty	1	0.95	0.95	0.97	0.95	0.98	0.98	0.98	0.98	0.97	0.96	0.96	0.95	0.95	0.96	0.94	0.99	0.96	0.95	0.95
parser	1	0.94	0.93	0.94	0.93	0.96	0.98	0.99	0.95	0.96	0.95	0.94	0.94	0.93	0.94	0.93	0.94	0.93	0.92	0.93
perlbmk	1	0.95	0.95	0.95	0.94	0.97	0.98	0.98	0.95	0.96	0.96	0.96	0.95	0.93	0.95	0.93	0.94	0.95	0.93	0.93
vortex	1	0.97	0.97	0.98	0.97	0.99	0.99	0.98	0.96	0.95	0.98	0.98	0.97	0.95	0.98	0.94	0.97	0.97	0.95	0.95
gap	1	0.9	0.9	0.9	0.9	0.93	0.93	0.94	0.89	0.89	0.91	0.91	0.91	0.86	0.87	0.85	0.91	0.87	0.86	0.87
bzip2	1	0.95	0.95	0.95	0.95	0.96	0.96	0.96	0.97	0.97	0.95	0.95	0.95	0.94	0.95	0.94	0.95	0.95	0.94	0.94
twolf	1	0.99	0.98	0.99	0.99	1	1	1	1	1	1	0.99	0.99	0.98	0.99	0.98	0.98	0.99	0.98	0.99
G.Mean	1	0.95	0.95	0.95	0.95	0.97	0.97	0.98	0.96	0.96	0.96	0.95	0.95	0.93	0.94	0.93	0.96	0.94	0.93	0.93

Table A.2: Execution time of the generated code: 16 registers

	IRC	IRC Split	Dummy Nodes	Freeze	Conser.	Pref.	H	HC	HR	HCR	HW	HA	HAC	HAR	HAM	HARC	HARCS	HACM	HARCM	HARM
gzip	1.06	0.96	0.95	0.95	0.95	0.99	0.97	1.04	0.95	1	0.95	0.95	0.97	0.94	0.96	0.94	0.95	0.97	0.96	0.94
vpr	1.03	1.01	1.01	1.01	1.01	1.02	1.03	1.02	1.02	1.03	1.03	1.02	1.02	1.01	1.01	1.02	1	1.02	1.02	1.01
mcf	1.01	0.95	0.96	0.96	0.96	0.97	0.99	0.99	0.99	0.96	0.97	0.97	0.96	0.95	0.96	0.94	0.97	0.96	0.96	0.96
crafty	1.12	1.09	1.09	1.1	1.09	1.09	1.1	1.1	1.1	1.11	1.08	1.08	1.07	1.06	1.09	1.06	1.06	1.08	1.06	1.07
parser	1.01	0.95	0.94	0.95	0.94	0.97	0.99	0.99	0.98	0.96	0.95	0.95	0.94	0.94	0.95	0.93	0.93	0.94	0.93	0.94
perlbmk	1.02	0.96	0.95	0.95	0.94	0.97	0.98	0.99	0.96	0.97	0.96	0.96	0.96	0.94	0.96	0.94	0.95	0.96	0.94	0.95
vortex	1	0.97	0.97	0.98	0.97	0.99	0.98	0.99	0.97	0.97	0.98	0.98	0.97	0.96	0.98	0.95	0.95	0.97	0.95	0.96
gap	0.97	0.92	0.92	0.92	0.92	0.95	0.95	0.95	0.9	0.91	0.93	0.93	0.93	0.86	0.9	0.86	0.87	0.89	0.88	0.88
bzip2	1	0.96	0.97	0.97	0.97	0.98	0.98	0.99	0.96	0.97	0.96	0.97	0.97	0.96	0.97	0.96	0.97	0.97	0.96	0.96
twolf	1.02	1.01	1	1.01	1	1.01	1.02	1.02	1.01	1.01	1.01	1.01	1	1.01	1.01	1	1	1	1	1.01
G.Mean	1.02	0.98	0.97	0.98	0.97	0.99	1	1.01	0.98	0.99	0.98	0.98	0.98	0.96	0.98	0.96	0.96	0.98	0.97	0.97

Table A.3: Execution time of the generated code: 8 registers

	IRC	IRC Split	Dummy Nodes	Freeze	Conser.	Pref.	H	HC	HR	HCR	HW	HA	HAC	HAR	HAM	HARC	HARCS	HACM	HARCM	HARM
gzip	1.13	1.06	1.06	1.06	1.06	1.08	1.11	1.11	1.05	1.09	1.07	1.07	1.07	1.02	1.1	1.03	1.03	1.1	1.05	1.04
vpr	1.14	1.12	1.11	1.15	1.14	1.12	1.12	1.12	1.12	1.14	1.12	1.12	1.12	1.08	1.13	1.13	1.13	1.13	1.13	1.11
mcf	1.12	1.07	1.07	1.08	1.07	1.09	1.1	1.1	1.13	1.13	1.1	1.1	1.09	1.09	1.1	1.08	1.08	1.1	1.1	1.08
crafty	1.43	1.45	1.45	1.45	1.45	1.37	1.38	1.39	1.39	1.42	1.38	1.38	1.38	1.36	1.38	1.36	1.36	1.37	1.36	1.36
parser	1.03	1.01	1.01	1.01	1.01	1.01	1.04	1.04	1.07	1.05	1.02	1.02	1.02	1	1.02	1	1	1.01	1	1.01
perlbmk	1.05	1.03	1.03	1.02	1.02	1.04	1.04	1.06	1.02	1.04	1.04	1.04	1.04	0.99	1.04	1	1	1.04	1.01	1.02
vortex	1.06	1.03	1.03	1.03	1.03	1.04	1.05	0.97	1.02	1.02	1.04	1.04	1.03	1.01	1.04	1.01	1.01	1.03	1.01	1.01
gap	1.03	1	1	1.01	1	1.01	1.02	1.01	0.96	0.97	1.01	1.01	1.01	0.92	0.97	0.93	0.93	0.95	0.94	0.95
bzip2	1.06	1.05	1.05	1.06	1.05	1.05	1.06	1.06	1.04	1.05	1.05	1.05	1.04	1.05	1.05	1.03	1.03	1.05	1.04	1.05
twolf	1.07	1.06	1.06	1.06	1.06	1.05	1.06	1.06	1.06	1.06	1.05	1.05	1.05	1.05	1.05	1.05	1.05	1.05	1.05	1.05
G.Mean	1.11	1.08	1.08	1.09	1.08	1.08	1.09	1.09	1.08	1.09	1.08	1.08	1.08	1.05	1.08	1.06	1.06	1.08	1.06	1.06

Table A.4: Dynamic Moves: 32 registers

	IRC	IRC Split	Dummy Nodes	Freeze	Conser.	Pref.	H	HC	HR	HCR	HW	HA	HAC	HAR	HAM	HARC	HARCS	HACM	HARCM	HARM
gzip	1	0.94	0.01	28.17	0.01	1.9	2.86	2.87	4.32	2.89	0.97	0.96	0.02	1.45	3.79	0.02	11.36	0.02	0.02	4.73
vpr	1	0.71	0.74	7.78	0.76	1.15	2.04	1.81	3.08	1.82	1.91	1.56	0.98	2.41	2.29	0.96	3.26	0.95	0.95	2.52
mcf	1	0.36	0.4	1.36	0.39	0.77	1.55	1.4	2.19	1.78	1.45	1.05	0.6	1.1	1.13	0.52	2.35	0.49	0.5	1.14
crafty	1	0.12	0.12	1.05	0.11	0.35	0.63	0.54	2.1	1.42	0.77	0.44	0.18	0.96	1	0.21	2.13	0.2	0.19	1.13
parser	1	0.28	0.32	1.51	0.46	0.58	1.15	1.29	1.97	1.36	1.14	0.77	0.73	1.31	1.57	0.7	2.16	0.69	0.68	1.61
perlbmk	1	0.3	0.38	2.81	0.57	1.35	1.85	1.77	3.55	2.44	1.56	1.43	1.09	2.3	2.11	1.05	1.05	1.01	1.02	2.24
vortex	1	0.12	0.11	1.18	0.14	0.19	0.38	0.28	1.01	0.67	0.38	0.31	0.16	0.46	0.35	0.15	1.07	0.14	0.14	0.43
gap	1	0.67	1.05	2.89	1.06	1.99	2.8	3.19	3.54	3.39	1.66	1.44	1.2	1.79	1.8	1.11	4.9	1.08	1.08	1.89
bzip2	1	0.56	0.05	23.13	0.02	2.2	4.02	2.83	7.61	3.19	2.45	2.34	0.14	5.82	7.36	0.14	0.27	0.14	0.14	8.55
twolf	1	0.96	1.18	8.24	1.18	1.29	1.72	1.83	2.59	1.84	1.72	1.5	1.07	1.5	2.04	1.07	1.39	1.07	1.07	2.36
G.Mean	1	0.40	0.22	3.92	0.22	0.93	1.57	1.45	2.81	1.89	1.26	1.02	0.37	1.55	1.77	0.36	1.96	0.35	0.35	1.98

Table A.5: Dynamic Moves: 16 registers

	IRC	IRC Split	Dummy Nodes	Freeze	Conser.	Pref.	H	HC	HR	HCR	HW	HA	HAC	HAR	HAM	HARC	HARCS	HACM	HARCM	HARM
gzip	1.02	0.94	0.001	28.15	0.01	0.02	0.99	0.99	2.89	0.99	0.96	0.96	0.02	1.91	2.86	0.02	0.02	0.02	0.02	2.86
vpr	0.65	0.31	0.31	4.73	0.52	0.54	1.1	0.85	1.6	0.85	1.07	0.99	0.42	1.39	1.05	0.41	0.48	0.37	0.37	1.45
mcf	1.42	0.21	0.19	0.96	0.29	0.48	0.88	1.09	1.51	1.09	0.75	0.67	0.4	0.83	0.99	0.36	0.71	0.3	0.38	1.02
crafty	0.88	0.08	0.07	0.91	0.27	0.17	0.34	0.33	1.51	0.8	0.43	0.26	0.15	0.46	0.95	0.12	0.1	0.12	0.1	0.97
parser	0.88	0.24	0.34	0.86	0.39	0.34	0.71	0.93	1.43	1.02	0.56	0.48	0.47	0.98	0.99	0.51	0.53	0.53	0.53	0.99
perlbmk	1.1	0.25	0.01	0.05	0.01	1.03	1.44	1.43	2.58	1.45	1.28	1.2	0.93	1.82	1.9	1.04	1.05	0.98	0.97	1.98
vortex	1.58	0.05	0.06	0.67	0.26	0.14	0.3	0.75	0.66	0.73	0.25	0.25	0.16	0.28	0.33	0.07	0.06	0.08	0.08	0.31
gap	0.88	0.25	0.39	1.88	0.4	0.78	1.04	1.27	1.77	1.46	0.66	0.61	0.36	0.87	1.1	0.39	0.57	0.3	0.3	1.09
bzip2	0.87	0.57	0.02	21.85	0.05	0.75	1.62	1.41	6.52	1.46	1.2	1.28	0.07	4.52	5.15	0.07	0.17	0.37	0.37	5.15
twolf	0.57	0.53	1.6	0.98	0.96	0.65	0.97	0.97	1.42	0.97	1.18	0.97	0.64	1.29	1.5	0.64	0.75	0.75	0.75	1.5
G.Mean	0.94	0.25	0.08	1.63	0.15	0.33	0.83	0.94	1.83	1.05	0.75	0.67	0.24	1.10	1.31	0.22	0.26	0.25	0.25	1.36

Table A.6: Dynamic Moves: 8 registers

	IRC	IRC Split	Dummy Nodes	Freeze	Conser.	Pref.	H	HC	HR	HCR	HW	HA	HAC	HAR	HAM	HARC	HARCS	HACM	HARCM	HARM
gzip	0.51	0.001	0.001	5.63	0.001	0.01	0.03	0.51	0.99	0.98	0.01	0.01	0.01	0.02	0.95	0.01	0.01	0.01	0.01	0.95
vpr	0.19	0.02	0.02	0.86	0.03	0.03	0.1	0.23	0.41	0.29	0.05	0.05	0.04	0.22	0.18	0.04	0.04	0.04	0.15	0.3
mcf	1.03	0.05	0.05	0.24	0.05	0.27	0.51	0.62	1.03	0.92	0.33	0.32	0.36	0.21	0.33	0.29	0.29	0.32	0.32	0.32
crafty	0.76	0.01	0.01	0.18	0.01	0.16	0.46	0.42	0.85	0.92	0.1	0.1	0.1	0.12	0.16	0.06	0.06	0.02	0.02	0.18
parser	0.64	0.05	0.06	0.18	0.11	0.08	0.27	0.57	0.47	0.61	0.08	0.08	0.08	0.23	0.21	0.08	0.07	0.08	0.07	0.22
perlbmk	1.01	0.16	0.16	0.4	0.16	0.77	0.91	0.98	1.65	1.3	0.8	0.77	0.78	1.01	0.93	0.76	0.75	0.72	0.73	0.93
vortex	1.15	0.02	0.02	0.14	0.01	0.16	0.2	0.7	0.36	0.5	0.07	0.07	0.07	0.08	0.09	0.06	0.06	0.06	0.06	0.14
gap	0.54	0.01	0.03	0.52	0.07	0.07	0.22	0.69	0.9	1.11	0.1	0.1	0.07	0.24	0.23	0.07	0.06	0.04	0.04	0.23
bzip2	0.53	0.01	0.01	4.38	0.03	0.18	0.07	0.44	1.17	0.77	0.03	0.1	0.02	0.77	1.69	0.03	0.03	0.33	0.34	1.68
twolf	0.03	0.001	0.001	1.28	0.001	0.001	0.01	0.02	0.23	0.02	0.001	0.001	0.001	0.22	0.22	0.001	0.001	0.001	0.001	0.11
G.Mean	0.47	0.01	0.01	0.60	0.02	0.07	0.15	0.39	0.69	0.53	0.05	0.06	0.05	0.20	0.33	0.05	0.05	0.05	0.06	0.34

Table A.7: Alloction time: 32 registers

	IRC	IRC Split	Preference	None	Hints	Round	Caller	Web	Web filter	Aggressive	Move related	Split
gzip	7.56	10.37	5.32	1	0.99	1.08	1.5	1.11	1.12	1.31	1.39	1.64
vpr	9.06	23.39	4.26	1	1.01	1.09	1.53	1.12	1.13	1.24	1.33	1.77
mcf	7.41	9.74	4.9	1	1	1.08	1.52	1.12	1.13	1.24	1.32	1.57
crafty	10.99	11.55	4.47	1	1.01	1.1	1.51	1.14	1.15	1.31	1.41	1.68
parser	7.25	11.5	4.46	1	1	1.13	1.48	1.12	1.13	1.23	1.36	1.78
perlbmk	10.6	12.45	4.95	1	1	1.11	1.44	1.13	1.13	1.4	1.51	1.77
vortex	9.15	9.99	4.79	1	0.99	1.1	1.49	1.11	1.12	1.26	1.36	1.59
gap	8.59	12.52	4.35	1	1.02	1.14	1.5	1.13	1.13	1.22	1.36	1.93
bzip2	7.81	10.71	4.86	1	0.99	1.11	1.53	1.12	1.13	1.24	1.35	1.68
twolf	10.74	12.75	4.95	1	1	1.09	1.52	1.13	1.14	1.3	1.39	1.71
G.Mean	8.81	12.09	4.72	1	1	1.1	1.5	1.12	1.13	1.27	1.38	1.71

Table A.8: Allocation time: 16 registers

	IRC	IRC Split	Preference	None	Hints	Round	Caller	Web	Web filter	Aggressive	Move related	Split
gzip	6.44	8.86	3.72	0.98	0.97	1.05	1.47	1.1	1.12	1.4	1.47	1.58
vpr	7.04	9.32	3.47	1	1.01	1.08	1.55	1.21	1.22	1.9	1.98	1.7
mcf	6.67	7.63	3.63	1.01	1.01	1.1	1.52	1.15	1.16	1.41	1.5	1.5
crafty	8.21	9.16	3.4	1.02	1.02	1.1	1.55	1.21	1.22	1.64	1.72	1.61
parser	6.42	9.87	3.48	0.99	0.99	1.08	1.49	1.16	1.17	1.61	1.7	1.72
perlbmk	10.26	11.53	3.61	1	0.99	1.07	1.46	1.17	1.19	1.83	1.9	1.73
vortex	8.07	8.84	3.63	1	0.99	1.09	1.49	1.14	1.15	1.5	1.59	1.56
gap	6.69	10.16	3.44	0.99	0.98	1.06	1.47	1.16	1.18	1.63	1.7	1.75
bzip2	6.43	8.6	3.47	0.99	0.98	1.09	1.52	1.14	1.16	1.41	1.51	1.61
twolf	7.7	9.29	3.38	0.99	0.98	1.06	1.51	1.22	1.23	2.29	2.36	1.6
G.Mean	7.31	9.28	3.52	1	0.99	1.08	1.5	1.17	1.18	1.64	1.73	1.63

Table A.9: Allocation time: 8 registers

	IRC	IRC Split	Preference	None	Hints	Round	Caller	Web	Web filter	Aggressive	Move related	Split
gzip	4.6	5.82	2.75	0.93	0.91	0.99	1.39	1.05	1.06	1.3	1.36	1.35
vpr	4.86	6.49	2.83	0.93	0.92	0.97	1.46	1.12	1.13	1.54	1.58	1.53
mcf	4.79	5.66	2.98	0.99	0.99	1.06	1.48	1.13	1.14	1.37	1.44	1.38
crafty	6.03	7.02	2.93	1.01	1.01	1.06	1.54	1.21	1.22	1.54	1.59	1.52
parser	5.21	7.01	2.84	0.94	0.93	1	1.43	1.09	1.1	1.45	1.51	1.55
perlbmk	5.4	7.15	2.69	0.92	0.92	0.97	1.37	1.09	1.1	1.48	1.53	1.47
vortex	5.27	6.33	2.85	0.97	0.94	1.03	1.44	1.1	1.11	1.4	1.46	1.41
gap	4.44	6.26	2.8	0.92	0.91	0.96	1.41	1.1	1.11	1.5	1.54	1.52
bzip2	5.27	6.12	2.83	0.97	0.95	1.02	1.48	1.11	1.12	1.34	1.39	1.46
twolf	5.65	6.49	2.71	0.95	0.94	1	1.44	1.17	1.18	1.79	1.84	1.48
G.Mean	5.13	6.42	2.82	0.95	0.94	1.01	1.44	1.12	1.13	1.47	1.52	1.47

Table A.10: Memory footprint: 32 registers

	IRC	IRC Split	Preference	None	Hints	Round	Caller	Web	Web filter	Aggressive	Move related	Split
gzip	14.63	20.71	6.22	1	0.98	1.04	3.43	1.51	1.51	1.56	1.38	2.46
vpr	22.82	31.51	5.48	1	1	1.05	3.58	1.54	1.54	2.03	1.39	3.28
mcf	10.45	13.67	5.79	1	0.99	1.04	3.37	1.6	1.6	1.6	1.33	2.21
crafty	37.31	47.7	5.91	1	0.99	1.06	3.84	1.6	1.6	1.8	1.47	3.36
parser	16.81	25.04	5.11	1	0.99	1.04	3.11	1.53	1.53	1.74	1.35	2.61
perlbmk	33.07	45.27	5.19	1	0.99	1.03	3.15	1.53	1.53	1.72	1.37	2.89
vortex	19.33	25.36	5.62	1	0.97	1.05	3.37	1.52	1.52	1.59	1.4	2.62
gap	20.28	30.5	5.11	1	1	1.02	3.4	1.56	1.56	1.79	1.38	3.39
bzip2	15.1	21.4	5.61	1	0.99	1.05	3.29	1.57	1.57	1.6	1.38	2.61
twolf	43.73	57.84	5.99	1	0.98	1.06	3.61	1.52	1.52	2.09	1.44	4.04
G.Mean	21.24	29.27	5.59	1	0.99	1.04	3.41	1.55	1.55	1.74	1.39	2.9

Table A.11: Memory footprint: 16 registers

	IRC	IRC Split	Preference	None	Hints	Round	Caller	Web	Web filter	Aggressive	Move related	Split
gzip	14.07	19.66	4.31	0.93	0.9	0.97	3.26	1.43	1.43	1.48	1.29	2.24
vpr	20.82	29	4.06	0.93	0.92	0.96	3.41	1.45	1.45	1.93	1.29	2.65
mcf	10.19	12.49	4.08	0.94	0.93	0.97	3.18	1.53	1.53	1.53	1.27	1.99
crafty	38.27	46.46	4.22	0.93	0.91	0.97	3.69	1.53	1.53	1.72	1.38	2.78
parser	16.05	23.54	3.83	0.93	0.92	0.97	3	1.46	1.46	1.67	1.27	2.28
perlbmk	32.08	43.83	3.82	0.93	0.92	0.96	3.05	1.46	1.46	1.65	1.31	2.5
vortex	18.73	24.45	4.07	0.93	0.9	0.98	3.24	1.45	1.45	1.52	1.31	2.35
gap	17.87	27.68	3.87	0.93	0.92	0.95	3.25	1.47	1.47	1.7	1.29	2.42
bzip2	14.52	20.02	3.97	0.93	0.92	0.98	3.12	1.5	1.5	1.53	1.29	2.28
twolf	40.79	53.82	3.98	0.93	0.91	0.98	3.3	1.44	1.44	1.98	1.35	2.98
G.Mean	20.26	27.52	4.02	0.93	0.91	0.97	3.24	1.47	1.47	1.66	1.3	2.43

Table A.12: Memory footprint: 8 registers

	IRC	IRC Split	Preference	None	Hints	Round	Caller	Web	Web filter	Aggressive	Move related	Split
gzip	13.78	16.86	3.56	0.89	0.86	0.92	3.17	1.39	1.39	1.4	1.23	2.07
vpr	19.11	25.18	3.53	0.9	0.88	0.91	3.28	1.4	1.4	1.51	1.22	2.62
mcf	9.86	11.56	3.55	0.91	0.9	0.94	3.12	1.5	1.5	1.48	1.22	1.87
crafty	43.01	48.02	3.69	0.9	0.88	0.92	3.65	1.51	1.51	1.52	1.32	2.73
parser	15.5	20.7	3.37	0.9	0.88	0.92	2.92	1.42	1.42	1.52	1.2	2.2
perlbmk	30.24	38.17	3.26	0.89	0.88	0.91	2.94	1.41	1.41	1.51	1.22	2.33
vortex	18.7	22.49	3.47	0.9	0.87	0.93	3.17	1.42	1.42	1.43	1.25	2.19
gap	16.67	21.54	3.4	0.88	0.88	0.9	3.17	1.42	1.42	1.56	1.2	2.44
bzip2	14.92	18.45	3.43	0.91	0.89	0.94	3.05	1.47	1.47	1.47	1.24	2.18
twolf	41.34	49.46	3.39	0.9	0.88	0.93	3.22	1.4	1.4	1.55	1.29	3.03
G.Mean	19.97	24.63	3.46	0.9	0.88	0.92	3.16	1.43	1.43	1.49	1.24	2.34