University of California

Los Angeles

Register Allocation by Puzzle Solving

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Fernando Magno Quintão Pereira

2008

The dissertation of Fernando Magno Quintão Pereira is approved.

Vivek Sarkar

Todd Millstein

Sheila Greibach

Bruce Rothschild

Jens Palsberg, Committee Chair

University of California, Los Angeles

2008

to the Brazilian People

Table of Contents

1 Introduction 1

2 Background 5

2.1 Introduction 5

2.1.1 Irregular Architectures 8

2.1.2 Pre-coloring 8

2.1.3 Aliasing 9

2.1.4 Some Register Allocation Jargon 10

2.2 Different Register Allocation Approaches 12

2.2.1 Register Allocation via Graph Coloring 12

2.2.2 Linear Scan Register Allocation 18

2.2.3 Register allocation via Integer Linear Programming 21

2.2.4 Register allocation via Partitioned Quadratic Programming 22

2.2.5 Register allocation via Multi-Flow of Commodities 24

2.3 SSA based register allocation 25

2.3.1 The Advantages of SSA-Based Register Allocation 29

2.4 NP-Completeness Results 33

3 Puzzle Solving 37

3.1 Introduction 37

3.2 Puzzles 39

3.2.1 From Register File to Puzzle Board 40

3.2.2 From Program Variables to Puzzle Pieces 42

3.2.3 Register Allocation and Puzzle Solving are Equivalent . . . 45

3.3 Solving Type-1 Puzzles 46

3.3.1 A Visual Language of Puzzle Solving Programs 46

3.3.2 Our Puzzle Solving Program 50

3.4 Spilling and Coalescing 53

3.5 Implementation Details 57

3.6 Experimental Results 59

3.7 Related Work 68

3.8 Final Remarks 70

4 Local Coalescing 72

4.1 Introduction 72

4.2 Solving Uncolored Puzzles 73

4.3 Optimal Local Coalescing 75

4.3.1 On The Number of Pieces Displaced 87

4.4 Final Remarks 89

5 SSA Elimination after Register Allocation 90

5.1 Introduction 91

5.2 Example 94

5.3 Our SSA Elimination Framework 97

5.4 From windmills to cycles and paths 101

5.5 SSA elimination 102

5.5.1 SSA Elimination and Critical Edges 104

5.6 Optimizations 106

5.6.1 Store hoisting 107

5.6.2 Load Lowering 107

5.6.3 Memory coalescing 109

5.7 Experimental results 110

5.8 Final Remarks 114

6 Conclusion 115

6.1 Summary of Results 115

6.2 Limitations of our approach 118

6.3 Open Problems 118

A Proofs 119

A.1 Proof of Theorem 1 119

A.1.1 Definitions 119

A.1.2 Structure of the Proof 121

A.1.3 From register allocation to coloring 121

A.1.4 Elementary programs and graphs 122

A.1.5 An elementary graph is the interference graph of an elementary program 132

A.1.6 From Aligned 1-2-coloring to Puzzle Solving 134

A.2 Proof of Theorem 2 138

A.3 Proof of Theorem 4 141

A.4 Proof of Theorem 13 146

A.5 Proof of Theorem 14 147

References 152

List of Figures

2.1 Memory hierarchy in a typical computer architecture. 6

2.2 An example program. 7

2.3 General purpose registers from the x86 architecture. This Figure was taken from [76]. 10

2.4 (a) Example program. (b) Live ranges represented as intervals. (c) Program after live range of variable a is split. (d) New interval representation. 11

2.5 Interference graph for the Program in Figure 2.2. 12

2.6 Chaitin *et al.*'s iterative graph coloring based register allocator. This Figure was taken from [16]. 14

2.7 Chaitin *et al.*'s program to represent C_4. This example was taken from [74]. 15

2.8 Briggs *et al.* graph coloring based register allocator. 16

2.9 Iterated Register Coalescing. This Figure was taken from [4]. . . . 17

2.10 Linear Scan register allocation. 19

2.11 Some example cost matrices for the program in Figure 2.2. 24

2.12 (a) Traditional register allocation, (b) SSA-based register allocation. 26

2.13 (a) SSA-form program. (b) Register assignment. (c) colored-SSA-form program. 27

2.14 The ϕ-matrix. 28

2.15 (a) A chordal graph. (b-c) Two non-chordal graphs. 28

2.16 (a) SSA-form program taken from Pereira and Palsberg [73]. (b) Program after the SSA elimination phase. (c) Interference graph and sequence of register assignments. 30

2.17 (a) Dominator tree of the example program. (b) Interference graph of the SSA-form program, and assignment sequence. (c) Final program after SSA-based register allocation. 31

2.18 A graph coloring SSA-based register allocator. 33

3.1 Three types of puzzles. 39

3.2 Examples of register banks mapped into puzzle boards. 41

3.3 (a) Original program. (b) Elementary program. 44

3.4 Mapping program variables into puzzle pieces. 45

3.5 Padding: (a) puzzle board, (b) pieces before padding, (c) pieces after padding. The new pieces are marked with stripes. 47

3.6 A visual language for programming puzzle solvers. 47

3.7 Our puzzle solving program 50

3.8 (a) The puzzles produced for the program given in Figure 3.3 (b). (b) An example solution. (c) The final program. 51

3.9 Register allocation with spilling and local coalescing 55

3.10 Benchmark characteristics. `LoC`: number of lines of C code. `asm`: size of x86 assembly programs produced by LLVM with our algorithm (bytes). `btcode`: program size in LLVM's intermediate representation (bytes). 61

3.11 The distribution of the 1,486,301 puzzles generated from SPEC CPU2000. 62

3.12 Number of calls to the puzzle solver per nonempty puzzle. #puzzles: number of nonempty puzzles. `avg` and `max`: average and maximum number of times the puzzle solver was used per puzzle. `once`: number of puzzles for which the puzzle solver was used only once. 63

3.13 Number of copy and swap instructions inserted per puzzle. . . . 64

3.14 Static improvement due to First-Chance Coalescing. The bars are normalized to register allocation without global coalescing. . . . 64

3.15 In both charts, the bars are relative to the puzzle solver; shorter bars are better for the other algorithms. **Stack size:** Comparison of the maximum amount of bytes reserved on the stack. **Number of memory accesses:** Comparison of the total static number of load and store instructions inserted by each register allocator. . . 65

3.16 Comparison of the running time of the code produced with our algorithm and other allocators. The bars are relative to gcc -O3; shorter bars are better. 66

3.17 (left) Example program. (center) Puzzle pieces. (right) Register assignment. 67

3.18 Comparison between compilation time of the puzzle solver and extended linear scan (LLVM's default algorithm). The bars are relative to the puzzle solver; shorter bars are better for extended linear scan. 68

3.19 Interference graph of the program in Figure 3.3 (b). 69

3.20 Elementary graphs and other intersection graphs. RDV-graphs are intersection graphs of directed lines on a tree [63]. 69

3.21 Algorithms and hardness results for graph coloring. TD = this dissertation. 70

4.1 The different parts of the puzzle board. 73

4.2 Program P_u that solves puzzles containing no pre-coloring. . . . 74

4.3 The case $r = 7$ of Lemma 5. 75

4.4 The cases $r = 2$ (left) and $r = 3$ (right) of Lemma 5. 76

4.5 The cases $r = 4$ (left) and $r = 5$ (right) of Lemma 5. The case $r = 6$ is similar to $r = 5$ and was omitted. 76

4.6 Configurations of preferences in a board without pre-coloring. . . 77

4.7 (a) The puzzle formed for the instruction `E := c` in Figure 3.17. (b) Puzzle after one area has been filled. 78

4.8 Program P_c that maximizes local coalescing. 79

5.1 (a) Example program in high level language. (b) Control flow of program converted to SSA-form. (c) Program after constant propagation. 95

5.2 (a) A possible colored representation of the example program. (b) SSA elimination seen as the implementation of parallel copies. (c) SSA elimination with on-demand spilling. 96

5.3 SSA-based register allocation and spill-free SSA elimination. . . . 97

5.4 (a) Original, non-CSSA-form program. (b) Program converted into CSSA via Sreedhar's "Method I". (c) Redundant copies removed via Budimlic's fast copy coalescing. 99

5.5 A ϕ-matrix and its representation as three location transfer graphs. 101

5.6 The presence of critical edges leads to incorrect code. 105

5.7 (a) Example program. (b) Program in CSSA-form. (c) Program after critical edge is eliminated. The interference graph of each program is shown below its control flow graph. 106

5.8 (a) Example program (b) Program augmented with mock ϕ-functions. (c) SSA elimination without load-lowering. (d) Load-lowering in action. 108

5.9 #ltg: number of location transfer graphs (in thousands), %sp: percentage of LTG's that are potential spills, #edg: number of edges in all the LTG's (in thousands), %mt: percentage of the edges that are memory transfers. 111

5.10 Execution time of different compilation passes. 112

5.11 Impact of Load Lowering (LL) and Redundant Memory Transfer Elimination (RMTE) on the code produced after SSA-elimination. (Up) Code size. (Down) Run-time. 113

A.1 Example of a composition graph (taken from [42]). 120

A.2 Some special graphs. 123

A.3 (a) Directed tree T. (b) Paths on T. (c) Corresponding RDV graph.125

A.4 The grammar of elementary programs. 130

A.5 An elementary program representing a clique substitution of P_3. . 134

A.6 Example of padding. Square nodes represent vertices of weight two, and the other nodes represent vertices of weight one. 135

A.7 A chordal graph represented as a program. 145

A.8 The parallel copy $l_\phi : (v_1, v_2, \ldots, v_n) := (v_{i1}, v_{i2}, \ldots, v_{in})$. 148

Acknowledgments

There are a number of people whose help was essential for the success of this work. First, I must mention that four years of my stay in the United States were funded by CAPES, the Brazilian Agency of Education. Without their financial support, this research would be impossible. Equally important was the guidance of Jens Palsberg, my advisor and friend. He is one of the greatest examples of enthusiasm and competence in Computer Science that I have ever seen, and it was a great honor to work with him. I must thank also Jonathan Lee and Krishna Nandivada, with whom I could collaborate in different projects. I had the privilege to interact with amazing professors in UCLA. I must thank specially professor Todd Millstein for reviewing many drafts of my papers, and guiding me across the scary adventure of being a teaching assistant. While researching about register allocation I met with Fabrice Rastello, Florent Bouchez, Philip Brisk and Sebastian Hack. Our discussions helped me to better understand the benefits and trade-offs of SSA-based register allocation. I also met Lang Hames and Bernhard Scholz, who generously provided the implementation of PBQP and IRC used in my experiments. I would like to thank Daniel Berlin for hosting me at Google and teaching me a lot about compilers and their practical importance. I am also glad to have enjoyed the company of great friends in the UCLA Compiler's Lab. Our heated, but always civil, RISK matches are among the dearest memories that I bring from the doctorate. Finally, there exist some people in this world who have nothing to do with Computer Science, but have everything to do with me. They know who they are, and they know that they are much more important to me than any degree or research will ever be. Without them, everything that I do would be completely meaningless, and I thank them from the bottom of my heart for being part of my life.

Vita

1980 Born, Nova Era, Minas Gerais, Brazil.

1998–2001 B.S., Federal University of Minas Gerais, Brazil.

1999 Teaching Assistant, Computer Science Department, UFMG. Taught two semesters of Linear Algebra under direction of Professor Hamilton Prado Bueno.

2000–2001 Research Assistant, Computer Science Department, UFMG. Did research on partial evaluation and distributed systems under direction of Professor Roberto da Silva Bigonha.

2002–2003 M.S., Federal University of Minas Gerais, Brazil. Designed a major software system for Coordinating Communication Middleware. This system is still in use.

2004–present Ph.D program, Computer Science Department, UCLA. Did research on Compilers with emphasis on Register Allocation, under direction of Professor Jens Palsberg.

Fall 2007 Teaching Assistant, Computer Science Department, UCLA. taugh one section of CS131, Introduction to Programming Languages, under direction of Professor Todd Millstein.

Summer 2008 Summer Internship, Google, Washington DC. Did research on pointer analysis under direction of software developer Daniel Berlin.

Publications

Fernando Magno Quintão Pereira and Jens Palsberg. *Register allocation by puzzle solving.* PLDI - SIGPLAN Conference on Programming Language Design and Implementation. 216-226. 2008

Venkata K. Nandivada,Fernando Magno Quintão Pereira andJens Palsberg. *A Framework for End-to-End Verification and Evaluation of Register Allocators.* SAS - 14th International Static Analysis Symposium. 153 - 169. 2007

Jonathan K. Lee, Jens Palsberg and Fernando Magno Quintão Pereira. *Alias Register Allocation for Straight-line Programs is NP-complete.* ICALP - 34th International Colloquium on Automata, Languages and Programming. 680-691. 2007

Fernando Magno Quintão Pereira, Marco Túlio de Oliveira Valente, Roberto da Silva Bigonha and Mariza Andrade da Silva Bigonha. *Arcademis: a Framework for Object Oriented Communication Middleware Development.* Software: Practice and Experience. 36(5) 495 - 512. 2006.

Fernando Magno Quintão Pereira and Jens Palsberg. *Register Allocation After Classical SSA Elimination is NP-Complete.* FOSSACS - Foundations of Software Science and Computation Structures. 79 - 93. 2006.

Fernando Magno Quintão Pereira and Jens Palsberg. *Register Allocation via*

Coloring of Chordal Graphs. APLAS - 3rd Asian Symposium on Programming Languages and Systems. 315 - 329. 2005.

Fernando Magno Quintão Pereira, Wagner Salazar Pires, Marco Túlio de Oliveira Valente, Roberto da Silva Bigonha and Mariza Andrade da Silva Bigonha. *Tactics for Remote Method Invocation.* SBLP - 8th Brazilian Symposium on Programming Languages. 102 - 115. 2004.

Fernando Magno Quintão Pereira, Marco Túlio de Oliveira Valente, Roberto da Silva Bigonha and Mariza Andrade da Silva Bigonha. *Arcademis: a Java Based Framework for Middleware Development.* SBRC - 22nd Brazilian Symposium on Computer Networks. 539 - 552. 2004.

Fernando Magno Quintão Pereira, Marco Túlio de Oliveira Valente, Roberto da Silva Bigonha and Mariza Andrade da Silva Bigonha. *Chamada Remota de Métodos na Plataforma J2ME/CLDC.* Revista do Instituto Nacional de Telecomunicaćões. Inatel. 7(1) 21 - 31. 2004.

Fernando Magno Quintão Pereira, Marco Túlio de Oliveira Valente, Roberto da Silva Bigonha and Mariza Andrade da Silva Bigonha. *Tactics for Remote Method Invocation.* Journal of Universal Computer Science (J.UCS). 10(7) 824 - 842. 2004.

Fernando Magno Quintão Pereira, Marco Túlio de Oliveira Valente, Roberto da Silva Bigonha and Mariza Andrade da Silva Bigonha. *Chamada Remota de Métodos na Plataforma J2ME/CLDC.* WCSF - 5th Brazilian Workshop on Wireless Communication and Mobile Computation. 157 - 168. 2003.

Fernando Magno Quintão Pereira, Leonardo Trivelato Rolla, Cristiano Gato de Rezende and Rodrigo Lima Carceroni. *The Language LinF for Fractal Specification.* SIBGRAPI - 16th Brazilian Symposium on Computer Graphics. 67 - 74. 2003.

Marco Túlio de Oliveira Valente, Fernando Magno Quintão Pereira, Roberto da Silva Bigonha and Mariza Andrade da Silva Bigonha. *A Coordination Model for Ad Hoc Mobile Systems.* EURO-PAR - 9th International Conference on Distributed Computing. 1075 - 1081. 2003.

Fernando Magno Quintão Pereira, Marco Túlio de Oliveira Valente, Roberto da Silva Bigonha and Mariza Andrade da Silva Bigonha. *A Java-based Simulator for Ad Hoc Mobile Distributed Systems.* FIDJI - International Workshop on Scientific Engineering of Distributed java Applications. Springer. 2002.

Marco Túlio de Oliveira Valente, Roberto da Silva Bigonha, Mariza Andrade da Silva Bigonha and Fernando Magno Quintão Pereira. *A Coordination Model for Ad Hoc Mobile Systems and its Formal Semantics.* WCSF - 4th Brazilian Workshop on Wireless Communication and Mobile Computation. 58 - 67. 2002.

Fernando Magno Quintão Pereira, Marco Túlio de Oliveira Valente, Roberto da Silva Bigonha and Mariza Andrade da Silva Bigonha. *Uma Linguagem para Coordenação de Aplicações em Redes Móveis Ad-hoc.* SBLP - 6th Brazilian Symposium on Programming Languages. 152 - 165. 2002.

Fernando Magno Quintão Pereira, Roberto da Silva Bigonha, Mariza Andrade da Silva Bigonha and Vladimir Oliveira de Iorio. *Aplicações de Avaliação Parcial de Programas.* 55th Reunião Nacional da Sociedade Brasileira pelo Progresso da Ciência. 2002.

Fernando Magno Quintão Pereira, Roberto da Silva Bigonha, Mariza Andrade da Silva Bigonha and Vladimir Oliveira de Iorio. *Avaliação Parcial de Programas usando CMIX/II.* SBLP - 5th Brazilian Symposium on Programming Languages. 32 - 47. 2001.

Abstract of the Dissertation

Register Allocation by Puzzle Solving

by

Fernando Magno Quintão Pereira

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2008

Professor Jens Palsberg, Chair

Register allocation is the problem that compilers face when assigning storage locations to the values used in a program. Even though this is an old problem, compilers still use heuristics to solve register allocation, or use exact algorithms that have exponential complexity. We present an optimal, polynomial time, register assignment algorithm. Our key insight is to show that register assignment is analogous to solving a collection of puzzles. We model the register file as a puzzle board and the program variables as puzzle pieces; pre-coloring and register aliasing fit in naturally. Our puzzles are solvable in linear time for a vast number of architectures, including x86, PowerPC and StrongARM. We have augmented our optimal register assignment algorithm with simple, yet powerful, heuristics for spilling and coalescing. Our implementation is as fast as the extended version of linear scan used by LLVM, the JIT compiler used in the openGL stack of Mac OS 10.5. Our implementation produces x86 code of similar quality to the code produced by the slower, state-of-the-art iterated register coalescing algorithm of George and Appel with extensions proposed by Smith, Ramsey, and Holloway. Relying on extensive experiments and theoretical insights, we show that register allocation by puzzle solving is feasible and useful in practice.

CHAPTER 1

Introduction

Programs in a high level language are written under the assumption that an unbounded number of variables can be created to store values. Once these programs are translated into machine code, some of these variables will be stored in registers, while others will have to be stored in memory. It is advantageous to maximize the number of variables stored in registers, because reading or writing registers is faster than accessing memory. However, computer architectures only provide a small number of registers, and normally different variables cannot be simultaneously stored in registers. If the number of registers is not large enough to accommodate all the variables, some of them must be sent to memory. These are called *spilled values*. The proposed dissertation is mostly concerned with the so-called Spill Free Register Allocation Problem, which can be defined as follows:

> Spill-free Register Allocation Problem
> **Instance**: a program P and a number K of available registers.
> **Problem**: can each of the variables of P be mapped to one of the K registers such that variables simultaneously alive are assigned to different registers?

A Bit of Recent History. The research that constitutes this dissertation started in the Winter of 2005, with the observation that the *interference graph* of many real world programs are chordal. Register allocation has a very intimate relation with graph coloring, and chordal graphs can be colored in time linear on the number of edges. This observation motivated our first paper on this field: *Register Allocation via Coloring of Chordal Graphs*, published in the Third Asian Symposium on Programming Languages and Systems (APLAS'05). This paper has been received relatively well by the research community: it is cited in at least 20 other publications, and it has been used in the Introductory Compiler Course taught in Carnegie Mellon by Frank Pfenning. One year after this first publication we released our second paper on this subject: *Register Allocation After Classical SSA Elimination is NP-Complete*, which we presented in the 2006 edition of the Conference on Foundations of Software Science and Computation Structures. In that work we proved that, although chordal graphs can be colored in polynomial time, the translation of programs into assembly code was still using algorithms that are NP-complete. This somehow negative result was later mitigated by the design of new algorithms that translate a program into assembly code without demanding NP-complete transformations. Finally, on the Spring of 2008 we released our most important result: *Register Allocation by Puzzle Solving*, published in The Conference on Programming Language Design and Implementation. The algorithms introduced in this last work have also been presented in talks at Intel, Google, Apple and many universities, and they are the basis of this dissertation.

In addition of these three papers, the research on this dissertation has produced direct contributions on two other publications. While implementing our register allocators, we have developed a technique to perform the translation validation of register allocation outputs. This technique is the core of the paper *A Framework for End-to-End Verification and Evaluation of Register Allocators*, presented in the 14th International Static Analysis Symposium in 2007. Our debugging method was paramount in the implementation of our register allocators, and it has found users among members of the open software compiler community. In the paper *Aliased Register Allocation for Straight-Line Programs Is NP-Complete*, published in the 34-th International Colloquium on Automata, Languages and Programming, we helped Jonathan Lee and Jens Palsberg to prove that aliasing, a common trait of architectures such as x86 and Ultra Sparc, causes register assignment to be NP-complete. This result is surprising, given how easy it is to perform register assignment in architectures without aliasing. Register allocation with aliasing is similar to the Shipbuilding Problem [42]. Larry Stockmeyer had proved that the latter problem is NP-complete, while he was working at IBM on the Seventies; however, the late Stockmeyer had never published his proof. Jonathan Lee has been able to reduce the Shipbuilding Problem to the Alias Register Allocation problem; thus, we believe that our reduction, to appear in Theoretical Computer Science, will be the first public proof of Stockmeyer's theorem.

The Thesis Statement. The objective of this research is to develop optimal, polynomial time, register assignment algorithms that can be effectively used in computer architectures such as x86, ARM, SPARC and PowerPC. In order to achieve this objective we will create a new model for register allocation, which we will call the puzzle-based paradigm. Thus, my thesis can be stated as follows:

> Register allocation by puzzle solving is feasible and effective for most computer architectures currently in use.

The Structure of this Dissertation. This dissertation contains the following contributions:

- we introduce the concept of register allocation by puzzle solving. The abstraction of puzzles simplifies register allocation because it naturally subsumes constraints of computer architectures such as aliasing and pre-colored registers. We show that register allocation puzzles have polynomial time solution. Furthermore, we can easily augment our puzzle solving technique with heuristics for spilling and coalescing (Chapter 3);

- we describe a new class of graphs: the *elementary graphs.* Many problems that are NP-complete for general graphs, such as graph coloring and maximal clique identification, have polynomial time solution when restricted to the family of elementary graphs (Chapter 3);

- we give optimal algorithms for the *local coalescing* problem. This problem consists in maximizing the number of variables assigned to the same registers across successive program instructions (Chapter 4);

- we introduce a new *SSA-elimination* algorithm. SSA-elimination is the process of converting a program from an intermediate representation called Static Single Assignment (SSA-form) into assembly code (Chapter 5). SSA-form is the intermediate representation used in many among the most popular compilers, such as Gcc 4.0 [44], Sun's HotSpot JVM [85], IBM's Jikes [86] and LLVM [58];

The remaining of this dissertation is organized as follows: Chapter 2 contains a short review on register allocation. The main concepts related to register allocation by puzzle solving are given in Chapter 3. Chapter 4 discuss optimal local coalescing. Chapter 5 presents our SSA-elimination algorithm and Chapter 6 summarizes our contributions and concludes this work.

CHAPTER 2

Background

2.1 Introduction

Computer architectures rely on a memory hierarchy to store the data that is manipulated by programs. Figure 2.1 shows the storage pyramid that is typically observed on a ordinary computer. At the very top of the pyramid we have registers, which provide to the CPU the fastest access to data. Operations of reading and writing to registers in a modern computer take no more than one cycle of the CPU clock. All this velocity comes with a cost: the register file is very small. For instance, the 32-bit x86 chip contains only eight general purpose registers, the 16-bit x86 chip contains 16 and the ARM and the PowerPC chips contain only 32 integer registers. In comparison, its is fairly common to find 200G hard disks in current computers - a difference of almost 33 orders of magnitude!

Because registers are so fast and so few, one of the greatest challenges of compiler writers is to design algorithms that keep the most used program variables in registers, while relegating the least used variables to memory if necessary. Register allocation is thus the problem of mapping variables to registers or memory. We will be using the program in Figure 2.2 to illustrate the main concepts related to register allocation.

A program can be described by its *control flow graph*. The control flow graph is formed by *basic blocks*. Each basic block has a unique label, and a list of

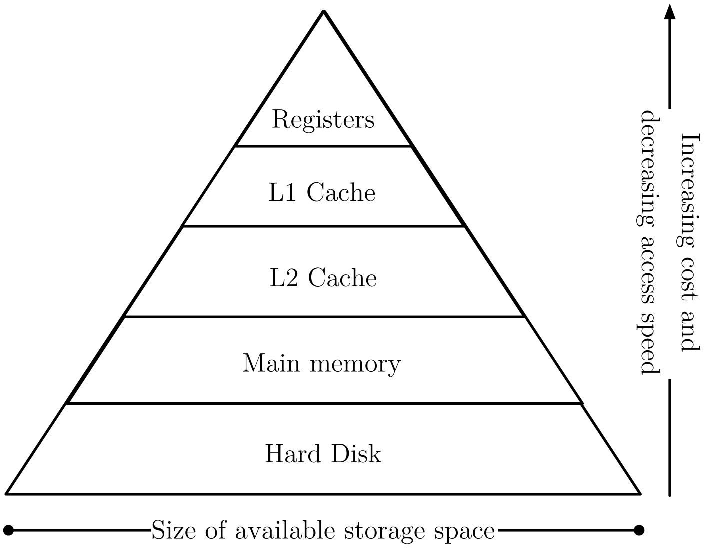

Registers
L1 Cache
L2 Cache
Main memory
Hard Disk
Increasing cost and decreasing access speed
Size of available storage space

Figure 2.1: Memory hierarchy in a typical computer architecture.

instructions. Instructions are the primary constituents of programs. Each instruction may apply an operation on some variables, which are called the *used variables.* An instruction may define one or more variables; these are called *defined variables.* Different computer architectures, e.g x86, PowerPC, ARM, etc, provide different sets of instructions, but all these sets are Turing Complete. The program in Figure 2.2 contains four basic blocks and 14 instructions. For simplicity we will be dealing with only four types of instructions: assignments, branches, jumps and joins. The first instruction of basic block L_2 is an assignment that uses the variable a and defines the variable c. We use assignments to abstract instructions that use or define variables. The actual operation that the instruction applies on its operands is not important for our purposes, and if the instruction has no operand on either the left or right side, we will represent this with a $\bullet$ symbol. The other kinds of instructions model the shape of the control

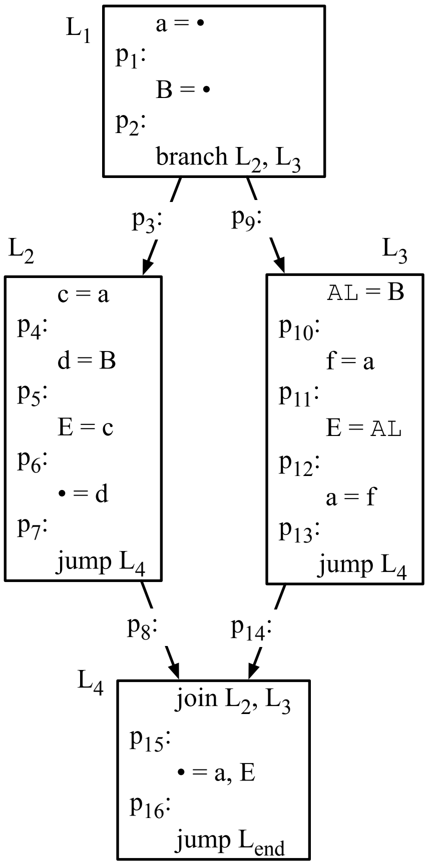


Figure 2.2: An example program.

flow graph. Branches finish basic blocks with multiple successors. Jumps finish basic blocks with a single successor and joins start basic blocks with multiple predecessors. The only operands of these instructions are basic block labels.

We call a *program point* the point between two consecutive instructions. The program in Figure 2.2 contains 16 program points named p_1 to p_{16}. We say that a variable v is *alive* at a program point p if there is a path from p to an instruction that uses v where v is not re-defined by any instruction. The collection of program points where a variable is alive is called its *live range*. For instance, the live range of variable B is $\{p_2, p_3, p_4, p_9\}$, whereas the live range of variable a includes all the program points but p_{11}, p_{12} and p_{16}. Notice that a is not alive at program points p_{11} and p_{12} because this variable is redefined by the instruction $a = f$, and its old value is not necessary after the instruction $f = a$. A simple algorithm to compute liveness information is given by Appel and Palsberg [4, p.206].

We say that two variables *interfere* if the intersection between their live ranges is non-empty. In this case, we also say that their live ranges overlap. For instance, variables c and d interfere, because their live ranges overlap at program point p_5; however, variables c and E do not intefere. This concept is very important for register allocation, because two variables that do not interfere can be stored in the same register.

2.1.1 Irregular Architectures

Modern computer architectures present *irregular* register banks. The two most common sources of irregularities are *pre-colored* registers and *aliasing* [57, 82, 83].

2.1.2 Pre-coloring

Pre-coloring is a very common phenomenon that forces some variables to be assigned to particular machine registers. A typical example is parameter passing in function calls. Architectures such as PowerPC and StrongARM use registers to pass arguments to functions. For instance, a two argument function call in PowerPC is written in assembly in a way similar to the code strip below:

```
r0 = arg0 ; the first argument must be passed in r0
r1 = arg1 ; the second argument must be passed in r1
bl foo ; branch and link, e.g, calls the function foo
```

The variables `arg0` and `arg1` can be stored in any register, but the variables r0 and r1 cannot: they are already allocated to registers r0 and r1. Many other common examples of pre-coloring are found in x86. For instance, in that architecture, the results of a division operation must be placed in the registers edx and eax. As a convention, we name a pre-colored variable with the name of its pre-coloring register. For instance, the variable AL in block L_3 of Figure 2.2 is pre-colored with register AL.

2.1.3 Aliasing

We say that an architecture contains aliasing when an assignment to one register name can affect the value of another [83]. For example, Figure 2.3 shows the set of general purpose registers used in the x86 architecture. The x86 architecture has four general purpose 16-bit registers that can also be used as eight 8-bit registers. Each 8-bit register is called a *low* address or a *high* address. The initial bits of a 16-bit register must be *aligned* with the initial bits of a *low*-address 8-bit register. The x86 architecture allows the combination of two 8-bit registers into one 16 bit register. Another example of aliased registers is the combination of two aligned single precision floating-point registers to form one double-precision register. Examples of architectures with such aliased registers include early versions of HP PA-RISC, the Sun SPARC, and the ARM processors. For a different kind of architecture, Scholz and Eckstein [82] describe experiments with the Carmel 20xxDSP Core, which has six 40 bit accumulators that can also be used as six 32-bit registers or as twelve 16-bit aligned registers. As a convention, along this dissertation we will use lower case names to denote values that fit in one single register, and upper case names to denote values that must fit in one register pair. Thus, in Figure 2.2 variables a, c and d fit in one register, whereas variables B and E fit in a register pair.

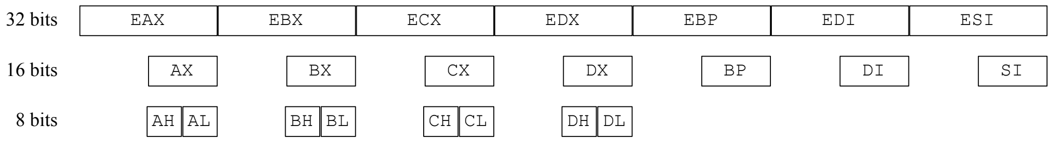


Figure 2.3: General purpose registers from the x86 architecture. This Figure was taken from [76].

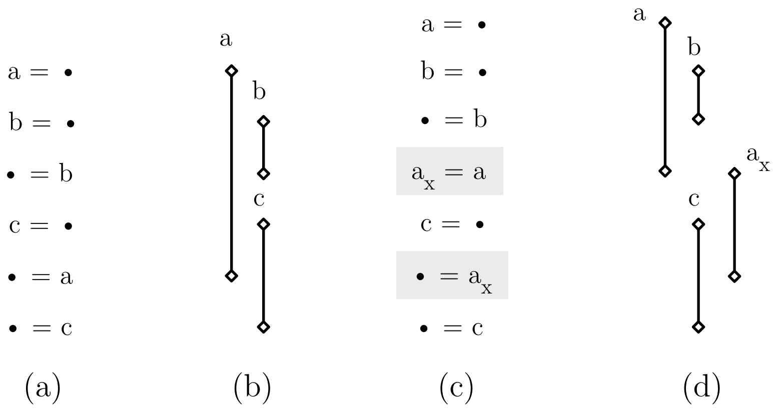
2.1.4 Some Register Allocation Jargon

Spilling Because computers have limited number of registers, they may not be enough to store all the variables in the source program. If that is the case, then some variables must be mapped to memory. The act of storing a variable into memory is called *spilling*. Spill is normally undesirable because it forces the register allocator to insert special instructions, that we will call spill code, in the target program to access values stored in memory. An instruction that copies a value from a register to a memory address is called a store. The opposite instruction, which copies a value from memory into a register, is called a load. These instructions tend to be slow compared to operations that do not access memory; thus, one of the objectives of a register allocator is to avoid inserting such instructions in the code that it produces.

Coalescing If two variables v_1 and v_2 do not interfere, and they are related by a copy instruction, that is, the source program contains an instruction such as $v_1 = v_2$, then it is desirable that these variables be allocated into the same register r. In this case, we will have the copy instruction $r = r$, which is redundant and can safely be removed from the target program. *Coalescing* is the act of mapping two non-interfering variables that are related by a copy instruction to the same register. For instance, in the program in Figure 2.2, the instructions $f = a$ and

$a = f$ can be eliminated from the program if variables a and f are assigned to the same register. Therefore, a good register allocator should not only assign different registers to interfering variables, but also try to assign the same register to variables related by copies.

Live Range Splitting This concept is the inverse of coalescing. Whereas coalescing join the live ranges of variables by removing copies from the source program, live range splitting divides the live range of variables by adding copies to the program and renaming variables. The splitting of live ranges tends to reduce the interferences between variables; thus, it might minimize the number of registers required by programs. Figure 2.4 shows an example of live range splitting.



```
a = •
b = •
• = b
c = •
• = a
• = c
```

(a)

(b)

```
a = •
b = •
• = b
a_x = a
c = •
• = a_x
• = c
```

(c)

(d)

Figure 2.4: (a) Example program. (b) Live ranges represented as intervals. (c) Program after live range of variable a is split. (d) New interval representation.

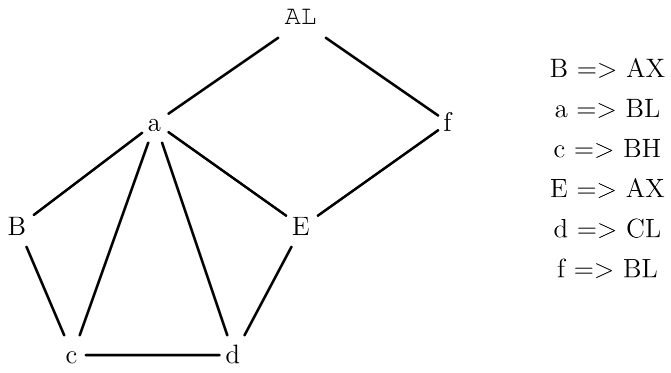


Figure 2.5: Interference graph for the Program in Figure 2.2.

2.2 Different Register Allocation Approaches

Register allocation is possibly the compilation problem with the greatest number of different algorithms already described in the literature. In the remainder of this section we will be describing several approaches to register allocation, using the program in Figure 2.2 as a running example.

2.2.1 Register Allocation via Graph Coloring

Graph coloring is the most used approach to solve register allocation. The *Interference Graph* of a program is the intersection graph of the live ranges of the variables in the program. That is, given a program P, its interference graph $G = (V, E)$ contains a vertex for each variable v of P. An edge (u, v) is in E if the intersection of the live ranges of variables v and u is non-empty. Figure 2.5 shows the interference graph for the Program in Figure 2.2, as well as a valid register assignment using three x86 registers: `AX`, `BX` and `CX`.

The problem of assigning registers to variables can thus be approximated by coloring the interference graph of the source program. Each color corresponds to a register, and interfering variables will be given different colors, given that they are adjacent on the interference graph. One of the first and most celebrated graph coloring based register allocators was described by Chaitin *et al.* [23, 24]. The algorithm described in [23] laid the foundations of practically all the graph coloring based register allocators that were introduced later. The core of Chaitin's algorithm is Kempe's coloring scheme [54]. Basically, a node v in the interference graph G can be colored if it has less than K neighbors, where K is the number of colors available. In this case, the node v can be safely removed from G, and placed on a stack of nodes that are guaranteed to be colorable. This process, called simplification, iterates until there is no more nodes to remove from G.

Two aspects of the register allocation problem complicate this technique: *spilling* and *coalescing*. Spilling is the act of mapping a variable to memory because there is no more registers available to hold its value. Coalescing is the act of mapping two non-interfering variables that are related by a copy instruction to the same register. For instance, in the program in Figure 2.2, the instructions $f = a$ and $a = f$ can be eliminated from the program if variables a and f are assigned to the same register. Therefore, a good graph coloring based register allocator should not only assign different colors to interfering variables, but also try to assign the same color to variables related by copies. Due to spilling and coalescing, Chaitin *et al.* proposed an iterative algorithm, illustrated in Figure 2.6. This algorithm has the following phases:

1. **Renumber**: discover live range information in the source program.
2. **Build**: build the interference graph.
3. **Coalesce**: merge the live ranges of non-interfering variables related by copy

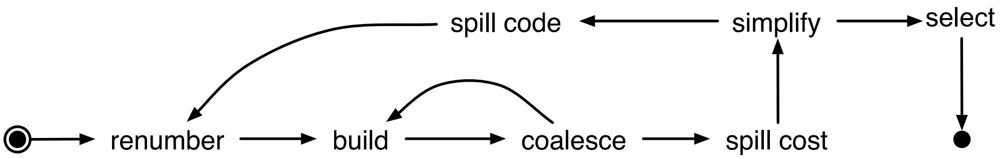


Figure 2.6: Chaitin *et al.*'s iterative graph coloring based register allocator. This Figure was taken from [16].

instructions.

4. **Spill cost**: compute the spill cost of each variable. That is a measure of the impact of mapping a variable to memory on the speed of the final program.
5. **Simplify**: Kempe's coloring method.
6. **Spill Code**: insert spill instructions, i.e loads and stores to commute values between registers and memory.
7. **Select**: assign a register to each variable.

One of the early achievements of Chaitin *et al.* [24] was to show that spill free register allocation is a NP-complete problem. Basically, Chaitin *et al.* proved that any graph is the interference graph of some program. For instance, to represent C_4, the cycle with four nodes, the NP-completeness proof would produce the program in Figure 2.7. The minimal coloring of such graph can be trivially mapped to a minimal coloring of C_4, by simply deleting node x. The NP-completeness result comes from the fact that finding a minimal coloring to a graph is NP-complete, as shown by Richard Karp in its seminal work [53].

Chaitin's algorithm had a few deficiencies that were improved by later works. One of the problems of that allocator was the aggressive coalescing policy. Merg-

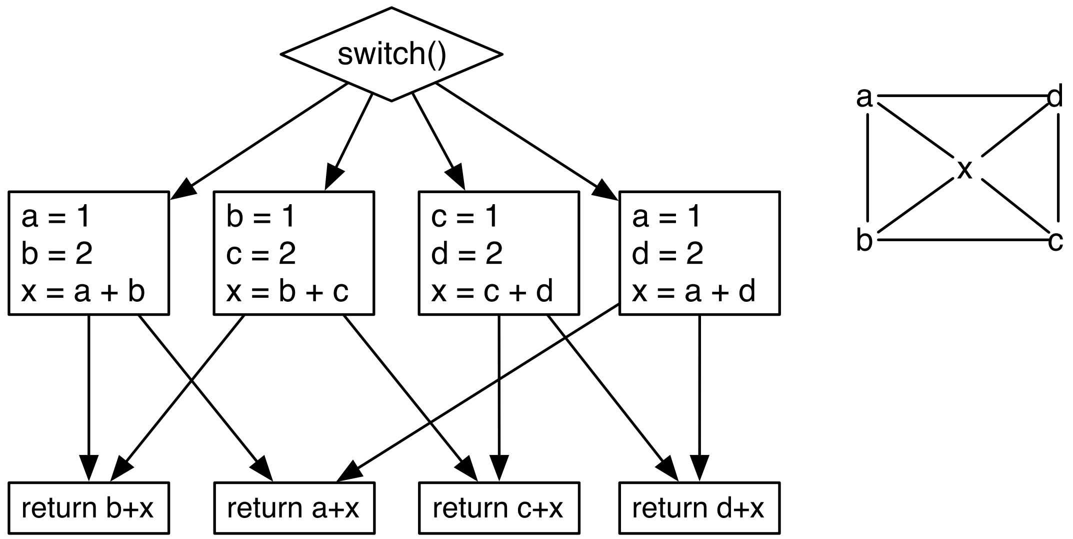


Figure 2.7: Chaitin *et al.*'s program to represent C_4. This example was taken from [74].

ing live ranges of variables has the undesirable effect of increasing the degree of vertices in the interference graph, and thus it might cause spilling. In order to solve this omission, Briggs *et al.* [16] introduced the concept of *conservative coalescing*. This is an extra criterion to decide when two live ranges can be merged. Thus, in addition of the non-interfering requirement, two variables can only be coalesced if their merging will not cause further spilling in the interference graph. Another improvement brought up by Briggs *et al.* was *biased coloring*: the select phase tries to assign the same color to variables that are copy related. Briggs *et al.* point that the combination of conservative coalescing and biased coloring could remove most of the copy instructions in the original program before register allocation. Finally, Briggs *et al.* introduced the concept of *optimistic coloring*: instead of spilling away variables that could not be simplified using Kempe's technique, Briggs *et al.* defer this decision to the simplification phase. Many

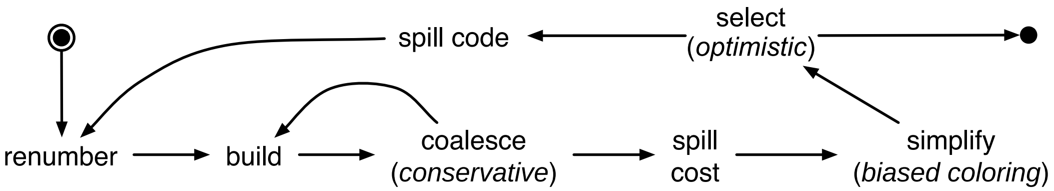


Figure 2.8: Briggs *et al.* graph coloring based register allocator.

times two or more of the neighbors of a vertex v will be assigned the same color, and v will be colorable even if it has a number of neighbors that is larger than the number of available colors. The modified version of Chaitin's algorithm, as described by Briggs *et al.* [16], is illustrated in Figure 2.8.

A criticism of Briggs allocator is that it is too conservative with regard to the coalescing policy, and thus it misses some copy instructions that could be removed. The coalescing criterion used by Briggs *et al.* [16] is described as follows: nodes a and b can be coalesced if the node that results from their merging has less than K neighbors of *significant* degree, where a node has significant degree if it has K or more neighbors. Subsequently, George and Appel [40] showed that this criterion could be relaxed to allow more aggressive coalescing without introducing extra spilling. They proposed Iterated Register Coalescing, a graph coloring based register allocator [40] that remains today, almost 15 years after its first release, the base algorithm taught in many compiler courses [4] and used as baseline in many research projects. Figure 2.9 illustrates the several phases of this algorithm.

An important addition of Iterated Register Coalescing over the previous allocators was a *Freeze* phase. If neither simplify nor coalesce could remove any node from the interference graph, the freeze step would mark a copy related node, so

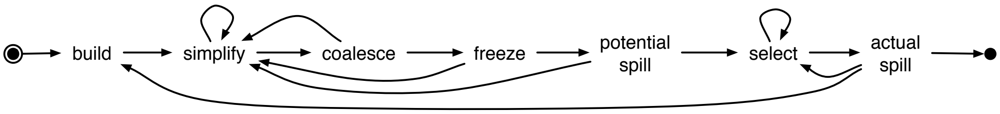


Figure 2.9: Iterated Register Coalescing. This Figure was taken from [4].

that it would no longer be considered for coalescing.

2.2.1.1 A Taxonomy of Coalescing Approaches

One quarter century after Chaitin's seminal paper, coalescing has been one of the main forces pushing new variations in graph coloring register allocation. Bouchez *et al.* [12] summarizes some of the best known approaches for performing register coalescing:

- *Aggressive Coalescing* [24, 23]: merge move-related vertices, regardless of the colorability of the interference graph after the merging.
- *Conservative Coalescing* [16]: merge moves if, and only if, it does not compromise the colorability of the interference graph.
- *Optimistic Coalescing* [68, 69]: coalesce moves aggressively, and if it compromises the colorability of the graph, then give up as few moves as possible.
- *Incremental Conservative Coalescing* [40]: remove one particular move instruction, while keeping the colorability of the graph.

Bouchez *et al.* [10, 12] have shown, by means of an ingenious sequence of reductions, that all these different realizations of the register coalescing problem are NP-complete for general interference graphs.

2.2.2 Linear Scan Register Allocation

Graph coloring is the most popular approach for register allocation, but it is not the only one, and it has been losing ground to a different, simpler approach called *Linear Scan*. Variations of this technique are adopted in many modern compilers, including LLVM [34] and the Java HotSpot client compiler [90]. In this section we describe Linear Scan and some of its more important variations.

Linear scan is a greedy algorithm introduced by Poletto and Sarkar on the late nineties [77]. It simplifies register allocation by reducing it to the problem of assigning colors to an ordered sequence of intervals. It is well know that ordered intervals can be colored optimally by a simple greedy algorithm [37]. However, linear scan is not optimal: it uses the optimal algorithm as an approximation to solve the register allocation problem. The algorithm starts by *linearizing* the basic blocks of the source program, that is, arranging these blocks in a linear sequence; the exact ordering chosen is not important and will not compromise the correctness of the results. Given this linearization, linear scan replaces the live range of each variable with a contiguous interval, called the variable *lifeline*, and then proceeds to color these intervals. Figure 2.10 illustrates this algorithm being applied on the program in Figure 2.2. In this case, register allocation amounts to coloring the seven intervals defined by variables a, B, c, d, E, f and `AL`.

The main appeal of linear scan is the allocation speed. This fact makes linear scan an attractive option to Just in Time (JIT) compilers, like Java HotSpot and LLVM. Timing comparisons between graph coloring and linear scan span a wide spectrum [76]. The original linear scan paper [77] suggests that graph coloring is about twice as slow as linear scan. These numbers are corroborated by Sagonas and Stenman [80]. Traub *et al.* [87] gives an slowdown of up to 3.5x for large

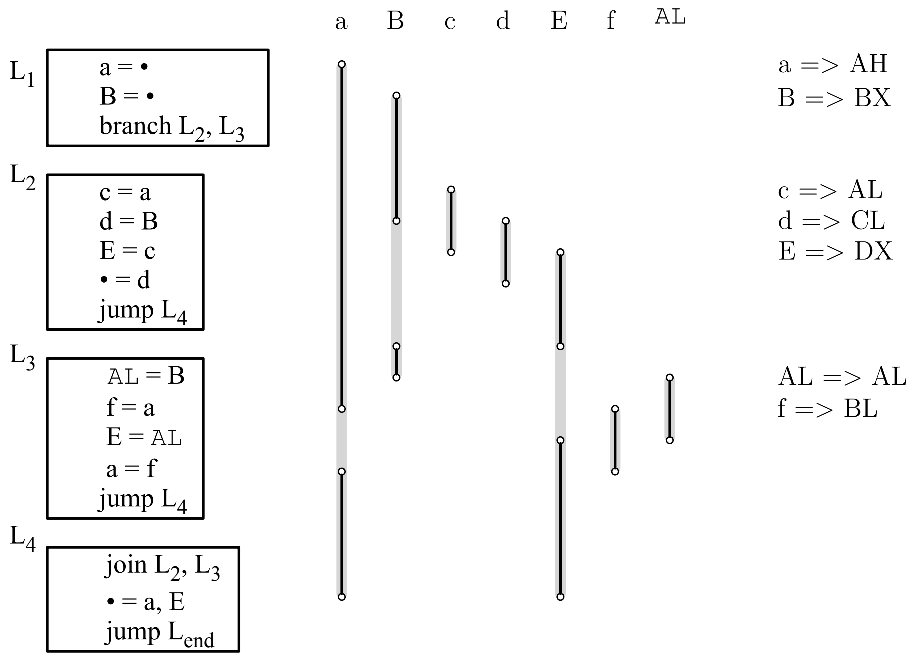


Figure 2.10: Linear Scan register allocation.

programs, and Sarkar and Barik [81] suggest a 20x slowdown. On the other hand, this speed pays a price in terms of quality of the code produced. Going back to our example in Figure 2.10, the original linear scan algorithm [77] would not be able to allocate variables B and E into the same register, even though these variables do not interfere, because their lifelines overlap. This omission is due to the original algorithm not handling *holes* in the live ranges of variables.

Later improvements on linear scan are able to handle holes in the live ranges of variables. An important extension is due to Traub *et al.* [87]. Traub's algorithm introduces the *binpacking* model: the machine registers are viewed as bins into which variable lifetimes can be packed. Thus, linear scan can assign two non-overlapping lifetimes to the same bin, or it can assign two lifetimes into the same bin if the live range that constitutes one of the lifetimes is completely contained in holes in the live range that constitutes the other lifetime. This model was later used by Mössenböck and Pfeiffer [64] in an algorithm that performs register allocation in programs in Static Single Assignment form.

Wimmer and Mössenböck have introduced several optimizations to linear scan [90]. Their algorithm handles holes in live ranges, but their most innovative extensions are optimal split positions, register hints and spill store elimination. Optimal split position is a technique to minimize the effects of spill code in the final program produced after register allocation. This optimization allows to move loads outside loops, for instance. Register hint is a simplified coalescing approach for linear scan. It is similar to biased coloring [16] in the sense that, when choosing a color to an interval i_1, if i_1 is connected to another interval i_2 via a move instruction, then the allocator attempts to assign to i_1 the color previously assigned to i_2. Spill store elimination is an optimization used to remove from the target program some store instructions that can be proved statically to be useless. A common example is multiple stores of a variable that is only defined once. In this case, all the store instructions can be replaced with a single store after the definition point of the variable.

The most recent addition to the family of linear scan algorithms is Sarkar and Barik [81] new method, called *Extended Linear Scan.* By inserting copy and swap instructions along the source program, this version of linear scan guarantees to use the minimal number of registers necessary to compile the source program. Sarkar's algorithm diverges from previous implementations because it has a preference towards inserting extra move instructions to avoid inserting spill code.

2.2.3 Register allocation via Integer Linear Programming

Quoting Dasgupta *et al.* [28], linear programming, together with dynamic programming, are the two sledgehammers of the algorithm craft. The linear programming framework fits a vast number of different optimization problems, and a subclass of this model, the 0-1 integer programming, has been used to solve register allocation. The basic idea of this approach is to model the interactions between registers and variables as constraints in a system of integer linear equations. For instance, given a register AX we define the following variables:

- $AX_r(v, p)$ is one if variable v reaches and leaves program point p allocated to register AX and is zero otherwise.
- $AX_l(v, p)$ is one if variable v reaches program point p in memory, but leaves it allocated to register AX. It is zero otherwise. The letter l indicates a load.
- $AX_s(v, p)$ is one if variable v reaches program point p allocated to register AX, but leaves it in memory. It is zero otherwise. The letter s indicates a store.

The constraints must guarantee that only one variable will be allocated into register AX at any time; thus, we add to the linear system the constraint:

$$\forall(v, p), \sum(AX_r(v, p) + AX_s(v, p) + AX_l(v, p)) \leq 1$$

Goodwin and Wilken [43] gave the first formulation of register allocation as a 0-1-integer programming problem. Their constraint set could handle several facets of register allocation, such as coalescing, spilling, rematerialization [15] and aliasing. Goodwin's register allocator produces code of very good quality; however, as integer linear programming is NP-complete [53], it presents a worst case exponential running time, and can take hours to find an optimal solution.

Two years after Goodwin's work, Kong and Wilken described a constraint set broad enough to encompass all the irregularities of the x86 architecture [57]. Posteriorly, Appel and George [3] introduced a different approach for IP-based register allocation: the separation of phases between spilling and register assignment. The constraint solver is responsible for defining which variables stay in registers and which variables are spilled. The non-spilled variables are mapped to registers in the next phase. This approach is faster than the previous IP-based algorithms; however, it may produce an undesirably large number of move instructions transferring values between registers. Appel and George use a variation of the optimistic coalescing of Park and Moon [69], which is not optimal, to remove redundant copies.

Finally, Fu and Wilken [36] presented a new IP formulation that keeps the optimal guarantees of the initial algorithm, e.g [43], but is 150 times faster. It is important to point that this "fast" algorithm is still much slower than traditional allocators such as linear scan, and even graph coloring. Therefore, IP-based register allocation has not yet seen use in industrial strength compilers. Nevertheless, it has been successfully used in research in embedded systems [65, 67], and in bit-width aware register allocation [5].

2.2.4 Register allocation via Partitioned Quadratic Programming

The Partitioned Boolean Quadratic Problem (PBQP) is a kind of Quadratic Assignment Problem (QAP) [22]. PBQP is NP-complete; however, a subclass of these problems can be solved in polynomial time. Quoting Hames and Scholtz [49], the input for PBQP is a set of discrete variables $X = \{x_1, \ldots, x_n\}$ and their finite domains $\{D_1, \ldots, D_n\}$, where $m_i = |D_i|$. A solution of PBQP is a function $h : X \rightarrow D$, where $D = D_1 \cup D_2 \cup \ldots \cup D_n$. For each variable x_i, h finds an element $d_i \in D_i$. The challenge of PBQP is to find a function h of minimal cost, where the cost c is controlled by two sets of terms:

- the cost of assigning variable x_i to d_i. This cost is measured by a *local* cost function $l(x_i, d_i)$;
- the cost given by the interactions between two variables. This is the cost of assigning variable x_i to d_i and assigning variable x_j to d_j. This cost is measured by a *related* cost function $r(x_i, x_j, d_i, d_j)$.

Thus, the total cost of an assignment is:

$$c = \sum_{1 \leq i \leq n} l(x_i, h(x_i)) + \sum_{1 \leq i < j \leq n} r(x_i, x_j, h(x_i), h_(x_j))$$

Although PBQP is NP-complete in general, there is a class of assignments that can be solved in polynomial time. The algorithm introduced in [82] is able to identify these problems, and solve them in $O(nm^3)$, where n is the number of variables and m is the maximum size of any domain.

PBQP has seen use in two different compiler related problems: instruction selection [30, 32, 31] and register allocation [49, 50, 82]. For register allocation, the solver introduced by Scholz *et al.* [82] associates a *cost matrix* C to each edge of the interference graph of the source program. Each cost matrix C_{uv} describes the tradeoffs of assigning different registers to variables u and v. As an example, lets build the cost matrix for variables a and B in the program of Figure 2.2, assuming a bank of registers with two registers only, AX and BX. We assume that each of these two registers have two disjoint aliases, in the same

$$
\begin{array}{c|ccc|}
 & \mathtt{sp} & \mathtt{AX} & \mathtt{BX} \\
\hline
\mathtt{sp} & 20 & 10 & 10 \\
\mathtt{AH} & 10 & \infty & 10 \\
\mathtt{AL} & 10 & \infty & 10 \\
\mathtt{BH} & 10 & 0 & \infty \\
\mathtt{BL} & 10 & 0 & \infty \\
\hline
\end{array}
$$

Cost of assigning B to one of these registers

Cost of assignign A to one of these registers

Figure 2.11: Some example cost matrices for the program in Figure 2.2.

configuration found in x86, that is, AX alias AH and AL, and BX alias BH and BL. Figure 2.11 shows the cost matrix.

The complexity of PBQP for register allocation is $O(|V|K^3)$, where $|V|$ is the number of variables in the source program, and K is the number of registers in the target architecture. Experiments performed by Hames *et al.* [49] showed that their implementation of a PBQP solver could find the optimal register allocation in 97.4% of the functions in SPEC CPU 2000. For the cases that no solution could be proved to be optimal, a branch-and-bound heuristics was used to find a satisfactory register mapping.

2.2.5 Register allocation via Multi-Flow of Commodities

An interesting way to see register allocation is as a *Multi-Flow of Commodities* (MFC) problem. This idea was introduced by Koes and Goldstein [55] in order to perform local register allocation. Local allocation is the version of register allocation that is restricted to basic blocks only, in contrast to global register allocation, that is concerned about the whole program. The same authors later extended their previous work to incorporate global allocation into the initial model [56]. Multi-flow of commodities has a close relation with register allocation.

For instance, it was the starting point of Lee *et al.*'s proof that aliasing register allocation is NP-complete [59]. MCF is a NP-complete problem, as shown by Even *et al.* [33], but it is possible to find solutions that, although non-optimal, are satisfactory enough, via heuristics. Koes and Goldstein have refined the heuristic approach with a progressive algorithm: their allocator uses a simple heuristics to find an initial allocation, and, if it is given extra time, it can improve this solution until reaching the optimal.

In the MCF approach, a program is seem as a collection of K pipes, thru which the allocator must pass a number of indivisible commodities. Each pipe corresponds to a physical location, either register or memory, and each commodity corresponds to a variable. Thus, a flow of a commodity represents the detailed allocation of the variable that the commodity encodes. The multi-commodite network flow naturally models several aspected of register allocation, including live-range splitting, rematerialization [15] and pre-colored registers, although it is unclear how this technique would model register banks with aliasing. Koes and Goldstein have optimized their progressive allocator to reduce the size of the target programs, and they have shown that this method is consistently able to produce code of smaller size than a graph coloring based allocator [56].

2.3 SSA based register allocation

An important breakthrough in register allocation happened in 2005, when three different research groups [10, 19, 48] proved independently that the interference graphs of programs in Static Single Assignment (SSA) form are chordal. This result is important because chordal graphs can be colored in polynomial time [37]. In this section we describe the main developments in the area of SSA-based register allocation.

SSA form. The *Static Single Assignment* (SSA) form is an intermediate representation in which each variable is defined at most once in the program code [27, 79]. Many industrial compilers use the SSA form as an intermediate representation: Gcc 4.0 [44], Sun's HotSpot JVM [85], IBM's Java Jikes RVM [86] and LLVM [58]. However, these compilers perform register allocation in *Post-SSA programs*, that is, programs in which ϕ-functions have been replaced with copy instructions, as shown in Figure 2.12(a). In SSA-based register allocators [10, 18, 46, 76] we observe an inversion of phases: ϕ-functions are replaced after registers have been assigned to variables, as illustrated in Figure 2.12(b). We will use *colored-SSA-form* to denote the variation of SSA-form in which each variable is associated with a physical location, which can be a register or a memory address. Figure 2.13 shows an SSA-form program and a corresponding colored-SSA-form program.

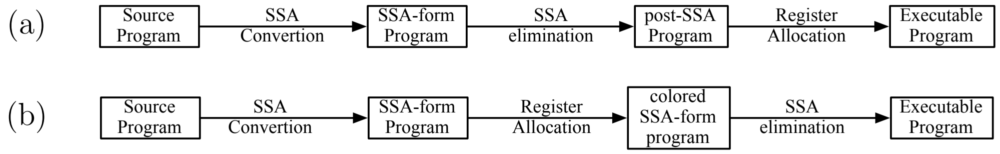


Figure 2.12: (a) Traditional register allocation, (b) SSA-based register allocation.

ϕ-functions SSA form uses ϕ-functions to join the live ranges of different names that represent the same value. We will describe the syntax and semantics of ϕ-functions using the matrix notation introduced by Hack et al. [46]. Figure 2.13 (a) outlines a ϕ-matrix. An equation such as $V = \phi M$, where V is a n-dimensional vector, and M is a $n \times m$ matrix, contains n ϕ-functions and m *parallel copies*, as outlined in Figure 2.14. Columns in the matrix correspond to control flow paths. The ϕ symbol works as a multiplexer. It will assign to each element v_i of V an element v_{ij} of M, where j is determined by the actual path taken

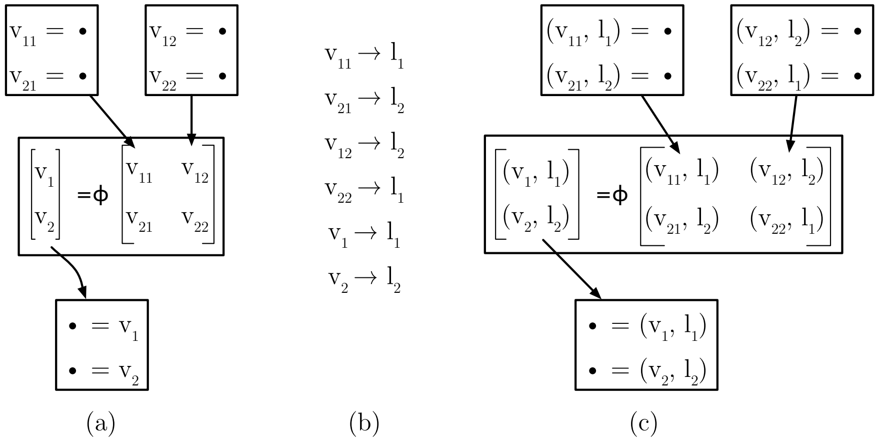


Figure 2.13: (a) SSA-form program. (b) Register assignment. (c) colored-SSA–form program.

during the program's execution. The semantics of ϕ-functions have been nicely described in [2]. The parameters of a ϕ-function are evaluated simultaneously, at the beginning of the basic block where the ϕ-function is defined. Thus, a ϕ-equation $V = \phi M$, where M has n columns encodes n parallel copies. If the path leading to column j is taken during program execution, all the elements in that column are copied to V in parallel.

Chordal Graphs A graph is chordal if every cycle with four or more edges has a *chord*, that is, an edge which is not part of the cycle but which connects two vertices on the cycle. (Chordal graphs are also known as 'triangulated', 'rigid-circuit', 'monotone transitive', and 'perfect elimination' graphs.) The graph in Figure 2.15(a) is chordal because the edge *ac* is a chord in the cycle *abcda*. The graph in Figure 2.15(b) is non-chordal because the cycle *abcda* is chordless. Finally, the graph in Figure 2.15(c) is non-chordal because the cycle *abcda* is chordless, just like in Figure 2.15(b).

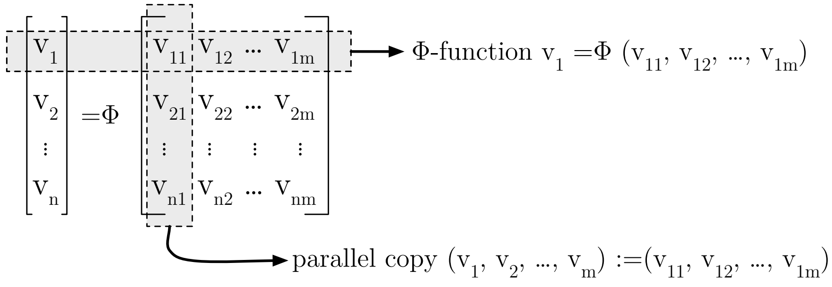
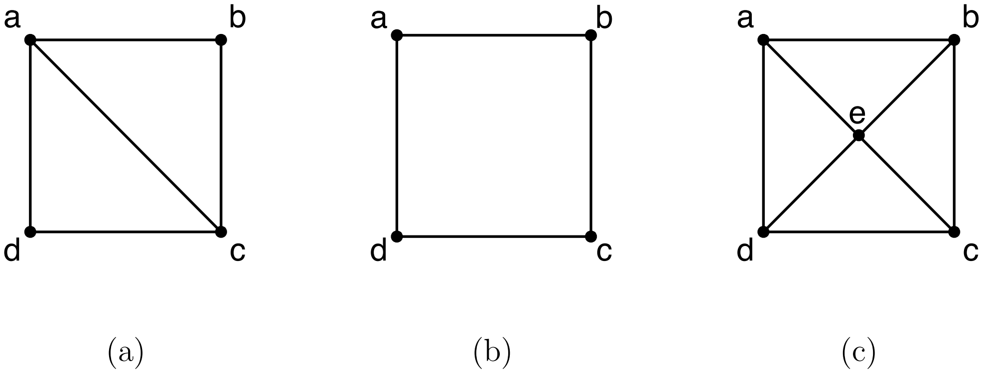

v1
v2
⋮
vn
=Φ
v11 v12 ... v1m
v21 v22 ... v2m
⋮ ⋮ ⋮ ⋮
vn1 vn2 ... vnm
Φ-function v1 =Φ (v11, v12, ..., v1m)
parallel copy (v1, v2, ..., vm) :=(v11, v12, ..., v1m)

Figure 2.14: The ϕ-matrix.



(a) (b) (c)

Figure 2.15: (a) A chordal graph. (b-c) Two non-chordal graphs.

Chordal graphs have several useful properties. Problems such as *minimum coloring*, *maximum clique*, *maximum independent set* and *minimum covering by cliques*, which are NP-complete in general, can be solved in polynomial time for chordal graphs [37]. In particular, optimal coloring of a chordal graph $G = (V, E)$ can be done in $O(|E| + |V|)$ time.

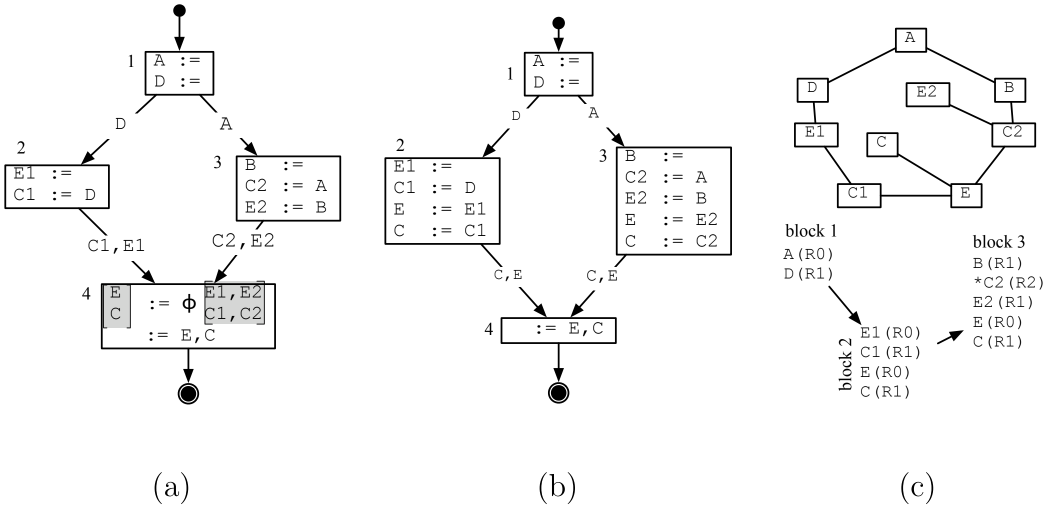
The Dominator Tree A basic block L_i *dominates* another basic block L_j if very path from the start of the program to L_j goes through L_i [4, p.379]. A basic block L_d is the *immediate dominator* of another block L if L_d dominates L, and for all other basic blocks L_i in the program, if L_i dominates L, then L_i

also dominates L_d. Every basic block in a program, except its entry point, has an immediate dominator, and this immediate dominator is unique. Thus, this relation of immediate dominance allows us to build a tree $T = (V, E)$ whose vertices are the basic blocks of a program, and an edge (L_s, L_t) is in E if basic block L_s is the immediate dominator of basic block L_t. This tree is called the *Dominator Tree* of the program.

The key insight to understand why SSA-form programs have chordal interference graphs lays on a well known characterization of chordal graphs: these are the intersection graphs of subtrees on a tree, as shown by Gavril in 1974 [38]. The live ranges of a SSA-form program are subtrees of the *dominator tree* of the program. By applying Gavril's result to the dominator trees of programs, Bouchez *et al.* [10], Brisk *et al.* [19] and Hack *et al.* [48] proved that the interference graph of a SSA-form program is chordal. The opposite direction is also true: a chordal graph is the interference graph of some SSA-form program [76].

2.3.1 The Advantages of SSA-Based Register Allocation

We illustrate the simplicity and elegance of SSA-based register allocation with an example. The program in Figure 2.16 was taken from [73]. Figure 2.16 (b) is the same program in post-SSA form, that is, after ϕ-functions have been replaced using the algorithm described by Appel and Palsberg [4]. Figure 2.16 (c) shows the sequence of assignments that would be performed by a linear scan algorithm that handles holes in live ranges of variables [90]. This algorithm would traverse blocks 1, 2, 4(where it executes no action), and finally block 3. When allocating registers in block 3, the allocator has to deal with temporaries `C` and `E` that have already been assigned machine registers. The graph in Figure 2.16 (c) cannot be colored with two colors. Its chromatic number is 3.

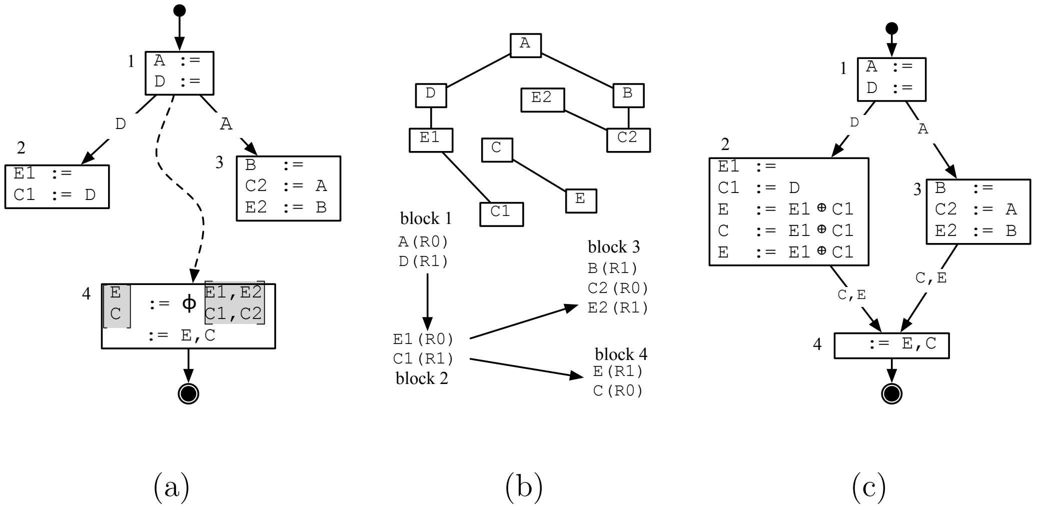


(a) (b) (c)

Figure 2.16: (a) SSA-form program taken from Pereira and Palsberg [73]. (b) Program after the SSA elimination phase. (c) Interference graph and sequence of register assignments.

Figure 2.17 (a) outlines the dominator tree of our example program. Figure 2.17 (b) shows the allocation produced by the same algorithm used in Figure 2.16. The basic blocks are visited in a pre-order traversal of the dominator tree. This way to assign registers to variables is called *tree-scan*, to distinguish it from the linear-scan strategy. The interference graph of the SSA-form program can be compiled with two registers: one register less than the minimum necessary in the post-SSA-form program. Figure 2.17 (c) shows the final program, after one swap sequence has been inserted in order to copy the values of `C1` and `E1` into `C` and `E`. The ability to swap registers is necessary in order to keep the register pressure low, as described by Bouchez *et al.* [11]. In this example, swaps are implemented using three xor operations.

Any of the register allocation models described in Section 2.2 can be adapted to run on SSA-form programs. The SSA-based register allocators described by



(a) (b) (c)

Figure 2.17: (a) Dominator tree of the example program. (b) Interference graph of the SSA-form program, and assignment sequence. (c) Final program after SSA-based register allocation.

Pereira and Palsberg [73], and Hack *et al.* [47] follow the graph coloring model. The SSA-based allocator described by Grund *et al.* [45] uses integer linear programming, and the puzzle solving algorithm introduced by Pereira and Palsberg [76] is based on the tree-scan model. Register allocators can benefit from the chordality of SSA-form programs in three main ways: (i) lower register pressure; (ii) separation between spilling and register assignment. (iii) simpler register assignment algorithms.

First, the SSA-form program never requires more registers than the original program, and often it will require less, as we showed in the previous example. The *register pressure* at any program point is the minimal number of registers necessary to allocate all the variables alive at that program point. The total number of registers necessary to allocate all the variables in a SSA-form program P equals the maximum register pressure at any point of P [48]. This value is equivalent to the size of the largest clique in the interference graph of P, and it can be determined in time proportional to the number of edges of this graph [37]. This relation is valid for register banks with no aliasing. The problem of determining the maximal register pressure for an architecture with aliased registers is NP-complete for SSA-form programs [59].

Another advantage of SSA-based register allocators is the potential separation of phases between spilling, register assignment and coalescing. The register pressure at any point of a SSA-form program is known locally. This fact allows the register allocator to remove live ranges from the program until the register pressure equals the number of available registers. Thus, the allocator is able to take spill decisions without having to actually assign registers to variables. In a subsequent phase, registers are assigned to variables, and the SSA properties guarantee that no further spilling will happen. Once registers are assigned to variables, a third phase takes charge of improving the initial register assignment, so that variables related by copies are given the same register. Figure 2.18 illustrates a typical SSA-based register allocator. This algorithm has five phases:

Build builds the interference graph using liveness analysis information.

Spill remove live ranges if the register pressure is greater than the number of available registers.

MCS finds an ordering of the nodes of the graph that can be optimally colored by a greedy algorithm using the Maximum Cardinality Search algorithm [7].

Color assign registers to variables using a trivial greedy coloring algorithm.

Coalesce exchange registers between variables in order to maximize the number of variables related by copy instructions that are given the same register.

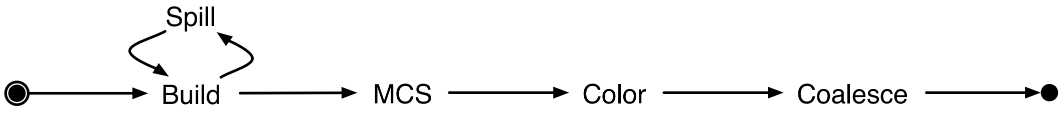


Figure 2.18: A graph coloring SSA-based register allocator.

Figure 2.18 sheds light on a third advantage of SSA-based register allocation: simplicity. The iterations between the spilling and the register assignment phases complicate the design of register allocators in general. This problem is particularly evident in graph coloring based allocators, as a comparison between Figure 2.18 and Figures 2.6, 2.8, 2.9 would reveal. Because spill decisions are taken independently of assignment decisions in a SSA-based allocator, the implementation of these algorithms tend to be simpler.

2.4 NP-Completeness Results

Spill Free Register Allocation has polynomial time solution for SSA-form programs [10, 19, 48], but it is NP-complete for programs in general [24]. One point that must be emphasized is that these two problems are obviously non-equivalent. Any program can be converted into SSA-form via a polynomial time transformation [27]. However, a register assignment for a SSA-form program cannot be converted back to an optimal register assignment of the original program in polynomial time unless P=NP.

An illustrative analogy is between the coloring of interval graphs and the coloring of circular-arc graphs. An interval graph is built on the following way: given a number of intervals on a line, assign a vertex to each interval. If two lines overlap, then connect their corresponding vertices. Circular-arc graphs are defined in a similar way, but using a circle instead of a line, and arcs on the circle instead of intervals. Finding the chromatic number of a circular-arc graph is a NP-complete problem [60], whereas the same problem for interval graphs has polynomial time solution [42]. To close our analogy, lets imagine that we can “cut” the base circle and all the arcs of a given circular-arc graph G_c on a given point. This cut effectively changes G_c into an interval graph G_i. We can find the chromatic number of G_i in polynomial time, and this value will never be larger than the chromatic number of G_c; however, finding the chromatic number of G_c is a NP-complete problem. Transforming a circular-arc graph to an interval graph in this way is analogous to converting a program into SSA-form.

Unfortunately, many register allocation related problems are still NP-complete even for SSA-form programs. Because SSA-form programs are a subset of general programs, these problems are, naturally, also NP-complete in general. We describe a number of these problems in this section.

Spilling The first of our NP-complete problems is *spill minimization*. When spills happen, loads and stores are inserted into the source program to transfer values to and from memory. If we assume that each load and store has a cost, then the problem of minimizing the total cost added by spill instructions is NP-complete, even for basic blocks in SSA-form, as shown by Farach and Liberatore [35]. If the cost of loads and stores is not taken into consideration, then a simplified version of the spilling problem is to determine the minimum number of variables that must be removed from the source program so that the program can be allocated with K registers. This problem is equivalent to determining if a graph G has a K-colorable induced subgraph, which is NP-complete for chordal graphs, but has polynomial time solution for interval graphs, as demonstrated by Yannakakis and Gavril [91].

Coalescing Coalescing is another part of register allocation that remains NP-complete even for SSA-form programs [29]. Ferriere *et al.* have proved that aggressive coalescing is NP-complete for SSA-form programs when the size of the ϕ-functions is unbounded [29]. Bouchez *et al.* [12] took Ferriere's study to the extreme, proving that all the best known variations of the coalescing problem, which are described in Section 2.2.1.1, are NP-complete. The proofs used in [12] have the extra appeal of relying on bounded structures such as ϕ-functions of size at most three.

Live range splitting SSA-based register allocators rely on the ability to swap the live ranges of variables without using extra registers to store temporary values. For instance, the live ranges of variables `E1` and `C1` are swapped in the basic block four of Figure 2.17 (c). Pereira and Palsberg [74] have proved that if swaps are not used by the register allocator, then the problem of deciding if a SSA-form program can be compiled with K registers is NP-complete, although the problem of deciding if a program can be compiled with $K - 1$ registers has polynomial time solution, as long as the K-th register is used as a temporary storage location. The proof in [74] considers that live ranges of variables can be split only at the end of basic blocks. Bouchez *et al.* [11] later refined this proof to show that the problem remains NP-complete even if live ranges are allowed to be split at any program point.

Aliasing Another factor that complicates register allocation is aliasing, described in Section 2.1.1. The problem of finding an optimal register assignment for a target architecture that allows registers to alias is NP-complete even for basic blocks in SSA-form as proved by Lee *et al.*

Pre-coloring Register allocation with pre-coloring is equivalent to the *pre-coloring extension problem* for graphs. In this problem we are given a graph G, an integer K and a partial function φ that associates some vertices of G to colors. The challenge is to extend φ to a total function φ' such that (1) φ' is a valid coloring of G and (2) φ' uses less than K colors. Biró *et al.* [8] have shown that pre-coloring extension is NP-complete for interval graphs, and thus, register assignment for basic blocks in SSA-form with pre-colored registers is also NP-complete. Interestingly, pre-coloring extension is NP-complete even for unit interval graphs [62], that is, interval graphs in which each interval has the same size.

Aliasing and pre-coloring cause SSA-based register allocation to be NP-complete; however, these problems have polynomial time solution if we use a program representation more restrictive than SSA. This format is called *Elementary Form*, and it is the subject of our next section.

CHAPTER 3

Puzzle Solving

In this chapter we show that register allocation can be viewed as solving a collection of puzzles. We model the register file as a puzzle board and the program variables as puzzle pieces; pre-coloring and register aliasing fit in naturally. For architectures such as PowerPC, x86, and StrongARM, we can solve the puzzles in polynomial time, and we have augmented the puzzle solver with a simple heuristic for spilling. For SPEC CPU 2000, the compilation time of our implementation is as fast as that of the extended version of linear scan used by LLVM, which is the JIT compiler in the openGL stack of Mac OS 10.5. Our implementation produces x86 code that is of similar quality to the code produced by the slower, state-of-the-art iterated register coalescing of George and Appel with the extensions proposed by Smith, Ramsey, and Holloway in 2004.

3.1 Introduction

This Chapter introduces a new abstraction for register allocation: the puzzle solving paradigm. We model the register file as a puzzle board and the program variables as puzzle pieces. The result is a collection of puzzles with one puzzle per instruction in the intermediate representation of the source program. We will show that puzzles are easy to use, that we can solve them efficiently, and that they produce code that is competitive with the code produced by state-of-the-art algorithms. Specifically, we will show how for architectures such as PowerPC, x86, and StrongARM we can solve each puzzle in linear time in the number of registers, how we can extend the puzzle solver with a simple heuristic for spilling, and how *pre-coloring* and *register aliasing* fit in naturally.

We have implemented a puzzle-based register allocator. Our register allocator has four steps:

1. transform the program into an *elementary program* by augmenting it with ϕ-functions [27], π-functions [9], and parallel copies (using the technique described in Section 3.2.2);

2. transform the elementary program into a collection of puzzles (using the technique described in Section 3.2.2);

3. do puzzle solving, spilling, and coalescing (using the techniques described in Sections 3.3 and 3.4); and finally

4. transform the elementary program and the register allocation result into assembly code (by implementing ϕ-functions, π-functions, and parallel copies using the algorithm described in Chapter 5.

For SPEC CPU2000, our implementation is as fast as the extended version of linear scan used by LLVM [58], which is the JIT compiler in the openGL stack of Mac OS 10.5. We compare the x86 code produced by gcc, our puzzle solver, the version of linear scan used by LLVM [34], the iterated register coalescing algorithm of George and Appel [40] with the extensions proposed by Smith, Ramsey, and Holloway [83], and the partitioned Boolean quadratic optimization algorithm [49]. The puzzle solver produces code that is, on average, faster than the code produced by extended linear scan, and of similar quality to the code produced by iterated register coalescing.

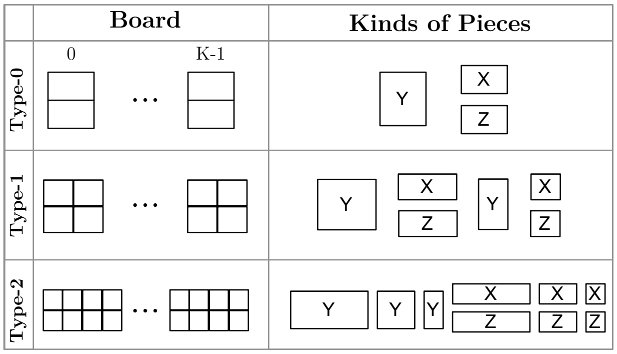


Figure 3.1: Three types of puzzles.

The key insight of the puzzles approach lies in the use of elementary programs, which are described in Section 3.2.2. In an elementary program, all live ranges are small and that enables us to define and solve one puzzle for each instruction in the program.

In the next section we define our puzzles and in Section 3.3 we show how to solve them. In Section 3.4 we present our approach to spilling and coalescing. We show experimental results in Section 3.6, and we discuss related work in Section 3.7.

3.2 Puzzles

A puzzle consists of a *board* and a set of *pieces*. Pieces cannot overlap on the board, and a subset of the pieces are already placed on the board. The *challenge* is to fit the remaining pieces on the board.

We will now explain how to map a register file to a puzzle board and how to map program variables to puzzle pieces. Every resulting puzzle will be of one of the three types illustrated in Figure 3.1 or a hybrid.

3.2.1 From Register File to Puzzle Board

The bank of registers in the target architecture determines the shape of the puzzle board. Every puzzle board has a number of separate *areas*, where each area is divided into two rows of *squares*. We will explain in Section 3.2.2 why an area has exactly *two* rows. The register file may support aliasing, which determines the number of columns in each area, the valid shapes of the pieces, and the rules for placing the pieces on the board. We distinguish three types of puzzles: type-0, type-1 and type-2, where each area of a type-n puzzle has 2^n columns.

Type-0 puzzles. The bank of registers used in PowerPC and the bank of integer registers used in ARM are simple cases because they do not support register aliasing. Figure 3.2 (a) shows the puzzle board for PowerPC. Every area has just one column that corresponds to one of the 32 registers. Both PowerPC and ARM give a type-0 puzzle for which the pieces are of the three kinds shown in Figure 3.1. We can place a X-piece on any square in the upper row, we can place a Z-piece on any square in the lower row, and we can place a Y-piece on any column. It is straightforward to see that we can solve a type-0 puzzle in linear time in the number of areas by first placing all the Y-pieces on the board and then placing all the X-pieces and Z-pieces on the board.

Type-1 puzzles. Figure 3.2 (b) shows the puzzle board for the floating point registers used in the ARM architecture. This register bank has 32 single precision registers that can be combined into 16 pairs of double precision registers. Thus, every area of this puzzle board has two columns, which correspond to the two registers that can be paired. For example, the 32-bit registers S0 and S1 are in the same area because they can be combined into the 64-bit register D0. Similarly, because S1 and S2 cannot be combined into a double register, they denote columns in different areas. ARM gives a type-1 puzzle for which the

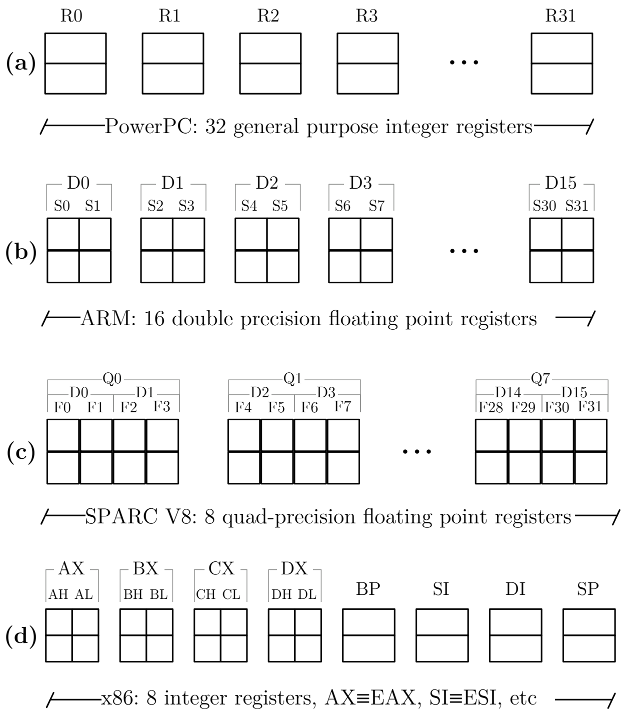

(a)
R0
R1
R2
R3
R31
PowerPC: 32 general purpose integer registers
(b)
D0
S0 S1
D1
S2 S3
D2
S4 S5
D3
S6 S7
D15
S30 S31
ARM: 16 double precision floating point registers
(c)
Q0
D0
D1
F0 F1 F2 F3
Q1
D2
D3
F4 F5 F6 F7
Q7
D14
D15
F28 F29 F30 F31
SPARC V8: 8 quad-precision floating point registers
(d)
AX
AH AL
BX
BH BL
CX
CH CL
DX
DH DL
BP
SI
DI
SP
x86: 8 integer registers, AX≡EAX, SI≡ESI, etc

Figure 3.2: Examples of register banks mapped into puzzle boards.

pieces are of the six kinds shown in Figure 3.1. We define the *size* of a piece as the number of squares that it occupies on the board. We can place a size-1 X-piece on any square in the upper row, a size-2 X-piece on the two upper squares of any area, a size-1 Z-piece on any square in the lower row, a size-2 Z-piece on the two lower squares of any area, a size-2 Y-piece on any column, and a size-4 Y-piece on any area. Section 3.3 explains how to solve a type-1 puzzle in linear time in the number of areas.

Type-2 puzzles. SPARC V8 [51, pp 33] supports two levels of register aliasing: first, two 32-bit floating-point registers can be combined to hold a single 64-bit value; then, two of these 64-bit registers can be combined yet again to hold a 128-bit value. Figure 3.2 (c) shows the puzzle board for the floating point registers of SPARC V8. Every area has four columns corresponding to four registers that can be combined. This architecture gives a type-2 puzzle for which the pieces are of the nine kinds shown in Figure 3.1. The rules for placing the pieces on the board are a straightforward extension of the rules for type-1 puzzles. Importantly, we can place a size-2 X-piece on either the first two squares in the upper row of an area, or on the last two squares in the upper row of an area. A similar rule applies to size-2 Z-pieces.

Hybrid puzzles. The x86 gives a hybrid of type-0 and type-1 puzzles. Figure 3.2 (d) shows the puzzle board of x86. The registers `AX`, `BX`, `CX`, `DX` give a type-1 puzzle, while the registers `EBP`, `ESI`, `EDI`, `ESP` give a type-0 puzzle. We treat the `EAX`, `EBX`, `ECX`, `EDX` registers as special cases of the `AX`, `BX`, `CX`, `DX` registers; values in `EAX`, `EBX`, `ECX`, `EDX` take up to 32 bits rather than 16 bits. Notice that x86 does not give a type-2 puzzle because even though we can fit four 8-bit values into a 32-bit register, x86 does not provide register names for the upper 16-bit portion of that register. For a hybrid of type-1 and type-0 puzzles, we first solve the type-0 puzzles and then the type-1 puzzles.

The floating point registers of SPARC V9 [88, pp 36-40] are a hybrid of a type-2 and a type-1 puzzle because half the registers can be combined into quad precision registers.

3.2.2 From Program Variables to Puzzle Pieces

We map program variables to puzzle pieces in a two-phase process: first we convert a source program into an *elementary program* and then we map the elementary program into puzzle pieces.

From a source program to an elementary program. We can convert an ordinary program into an *elementary program* in three steps. First, we transform the source program into static single assignment (SSA) form [27] by renaming variables and adding ϕ-functions at the beginning of each basic block. Second, we transform the SSA-form program into static single information (SSI) form [1]. Programs in SSI-form have the property that if a variable is alive in two basic blocks, then one of these blocks dominates the other. In our flavor of SSI form, every basic block ends with a π-function that renames the variables that are live going out of the basic block. The π-functions are the dual of ϕ-functions. Whereas a ϕ-function has the functionality of a variable multiplexer, a π-function is analogous to a demultiplexer that performs a parallel assignment depending on the execution path taken. For example, consider the assignment

$$[(v_{11}, \ldots, v_{n1}) : L_1, \ldots (v_{1m}, \ldots, v_{nm}) : L_m] = \pi(v_1, \ldots, v_n)$$

which represents m π-nodes such as $(v_{i1} : L_1, \ldots, v_{im} : L_m) \leftarrow \pi(v_i)$. This instruction has the effect of assigning to each variable $v_{ij} : L_j$ the value in v_i if control flows into block L_j. (The name π-assignment was coined by Bodik *et al.* [9]. It was originally called σ-function in [1], and *switch operators* in [52].) Finally, we transform the SSI-form program into an elementary program by inserting a parallel copy between each pair of consecutive instructions in a basic block, and renaming the variables alive at that point. Appel and George used the idea of inserting parallel copies everywhere in their ILP-based approach to register allocation with optimal spilling [3].

In summary, in an elementary program, every basic block begins with a ϕ-function, has a parallel copy between each consecutive pair of instructions, and ends with a π-function. Figure 3.3 (a) shows a program, and Figure 3.3 (b) gives the corresponding elementary program. As an optimization, we have removed useless ϕ-functions from the beginning of blocks with a single predecessor.

Cytron *et al.* [27] gave a polynomial time algorithm to convert a program

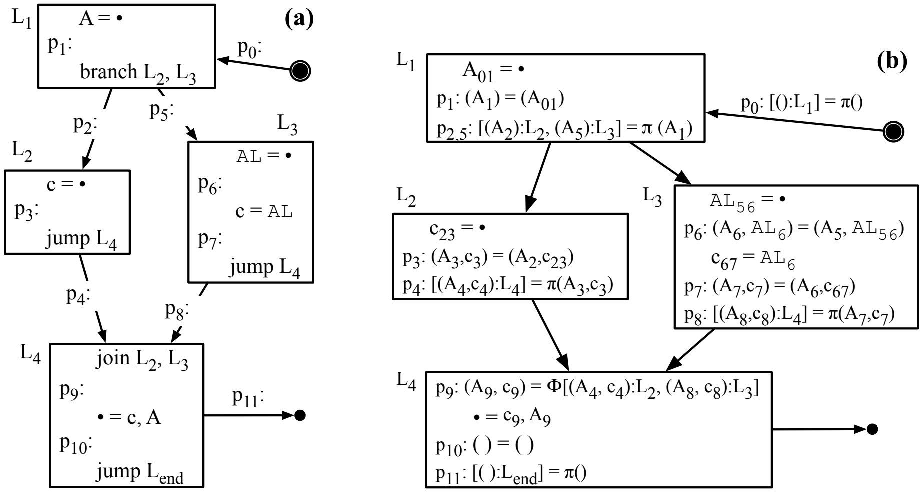

(a)
L_1
A = •
p_1:
branch L_2, L_3
p_0:
p_2:
p_5:
L_2
L_3
c = •
p_3:
jump L_4
AL = •
p_6:
c = AL
p_7:
jump L_4
p_4:
p_8:
L_4
join L_2, L_3
p_9:
• = c, A
p_{10}:
jump L_{end}
p_{11}:
(b)
L_1
A_{01} = •
p_1: $(A_1) = (A_{01})$
$p_{2,5}$: $[(A_2):L_2, (A_5):L_3] = \pi (A_1)$
p_0: $[():L_1] = \pi()$
L_2
L_3
c_{23} = •
p_3: $(A_3,c_3) = (A_2,c_{23})$
p_4: $[(A_4,c_4):L_4] = \pi(A_3,c_3)$
AL_{56} = •
p_6: $(A_6, AL_6) = (A_5, AL_{56})$
$c_{67} = AL_6$
p_7: $(A_7,c_7) = (A_6,c_{67})$
p_8: $[(A_8,c_8):L_4] = \pi(A_7,c_7)$
L_4
p_9: $(A_9, c_9) = \Phi[(A_4, c_4):L_2, (A_8, c_8):L_3]$
• = c_9, A_9
p_{10}: ( ) = ( )
p_{11}: $[( ):L_{end}] = \pi()$

Figure 3.3: (a) Original program. (b) Elementary program.

into SSA form, and Ananian [1] gave a polynomial time algorithm to convert a program into SSI form. We can perform the step of inserting parallel copies in polynomial time as well.

From an elementary program to puzzle pieces. A variable v is said to be *live-in* at instruction i if its live range contains a program point that precedes i; v is *live-out* at i if v's live range contains a program point that succeeds i. For each instruction i in an elementary program we create a puzzle that has one piece for each variable that is live in or live out at i (or both). The live ranges that end at i become X-pieces; the live ranges that begin at i become Z-pieces; and the live ranges that cross i become Y-pieces. Figure 3.4 gives an example of a program fragment that uses six variables, and it shows their live ranges and the resulting puzzle pieces.

We can now explain why each area of a puzzle board has exactly two rows. We can assign a register both to one live range that ends in the middle and to one live range that begins in the middle. We model that by placing a X-piece in

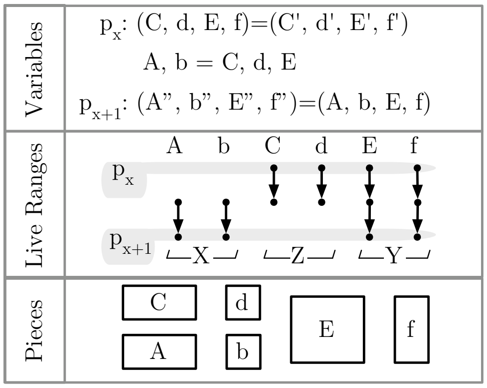

Variables
px: (C, d, E, f)=(C', d', E', f')
A, b = C, d, E
px+1: (A", b", E", f")=(A, b, E, f)
Live Ranges
A b C d E f
px
px+1
X
Z
Y
Pieces
C
d
A
b
E
f

Figure 3.4: Mapping program variables into puzzle pieces.

the upper row and a Z-piece right below in the lower row. However, if we assign a register to a long live range, then we cannot assign that register to any other live range. We model that by placing a Y-piece, which spans *both* rows.

The sizes of the pieces are given by the types of the variables. For example, for x86, an 8-bit variable with a live range that ends in the middle becomes a size-1 X-piece, while a 16 or 32-bit variable with a live range that ends in the middle becomes a size-2 X-piece. Similarly, an 8-bit variable with a live range that begins in the middle becomes a size-1 Z-piece, while a 16 or 32-bit variable with a live range that ends in the middle becomes a size-2 Z-piece. An 8-bit variable with a long live range becomes a size-2 Y-piece, while a 16-bit variable with a long live range becomes a size-4 Y-piece. Figure 3.8 (a) shows the puzzles produced for the program in Figure 3.3 (b).

3.2.3 Register Allocation and Puzzle Solving are Equivalent

Theorem 1 ***(Equivalence)*** *Spill-free register allocation for an elementary program is equivalent to solving a collection of puzzles.*

Proof. See Appendix A.1. □

3.3 Solving Type-1 Puzzles

Figure 3.7 shows our algorithm for solving type-1 puzzles. Our algorithmic notation is visual rather than textual. The goal of this section is to explain how the algorithm works and to point out several subtleties. We will do that in two steps. First we will define a visual language of puzzle solving programs that includes the program in Figure 3.7. After explaining the semantics of the whole language, we then focus on the program in Figure 3.7 and explain how seemingly innocent changes to the program would make it incorrect.

We will study puzzle-solving programs that work by completing *one area at a time*. To enable that approach, we may have to *pad* a puzzle before the solution process begins. If a puzzle has a set of pieces with a total area that is less than the total area of the puzzle board, then a strategy that completes one area at a time may get stuck unnecessarily because of a lack of pieces. So, we pad such puzzles by adding size-1 X-pieces and size-1 Z-pieces, until these two properties are met: (i) the total area of the X-pieces equals the total area of the Z-pieces; (ii) the total area of all the pieces is $4K$, where K is the number of areas on the board. Note that *total area* includes also pre-colored squares. Figure 3.5 illustrates padding. In the full version [75] we show that a puzzle is solvable if and only if its padded version is solvable.

3.3.1 A Visual Language of Puzzle Solving Programs

We say that an area is *complete* when all four of its squares are covered by pieces; dually, an area is *empty* when none of its four squares are covered by pieces.

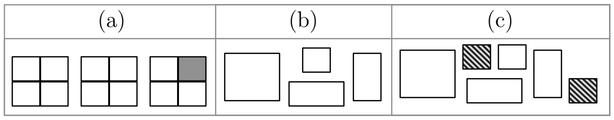


Figure 3.5: Padding: (a) puzzle board, (b) pieces before padding, (c) pieces after padding. The new pieces are marked with stripes.

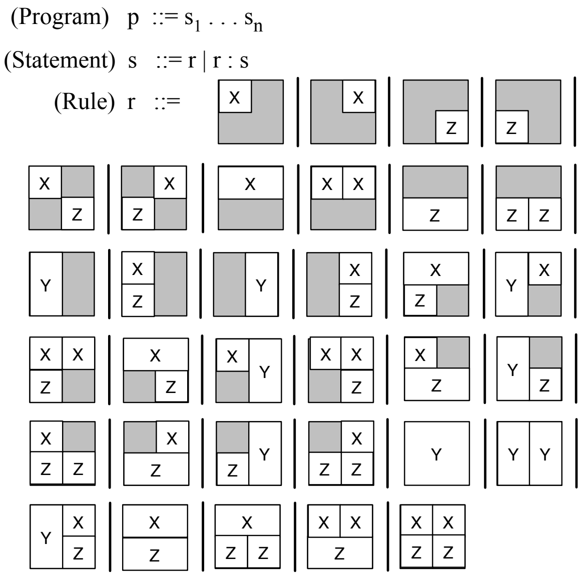


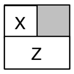
Figure 3.6: A visual language for programming puzzle solvers.

The grammar in Figure 3.6 defines a visual language for programming type-1 puzzle solvers: a program is a sequence of statements, and a statement is either a rule r or a conditional statement $r : s$. We now informally explain the meaning of rules, statements, and programs.

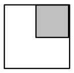
Rules. A rule explains how to complete an area. We write a rule as a two-by-two diagram with two facets: a *pattern*, that is, dark areas which show the squares (if any) that have to be filled in already for the rule to apply; and a *strategy*, that is, a description of how to complete the area, including which pieces to use and where to put them. We say that the pattern of a rule *matches* an area a if the pattern is the same as the already-filled-in squares of a. For a rule r and an area a where the pattern of r matches a,

- the application of r to a *succeeds*, if the pieces needed by the strategy of r are available; the result is that the pieces needed by the strategy of r are placed in a;
- the application of r to a *fails* otherwise.

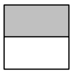
For example, the rule



has a pattern consisting of just one square—namely, the square in the top-right corner, and a strategy consisting of taking one size-1 X-piece and one size-2 Z-piece and placing the X-piece in the top-left corner and placing the Z-piece in the bottom row. If we apply the rule to the area



and one size-1 X-piece and one size-2 Z-piece are available, then the result is that the two pieces are placed in the area, and the rule succeeds. Otherwise, if one or both of the two needed pieces are not available, then the rule fails. We cannot apply the rule to the area



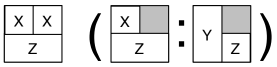
because the pattern of the rule does not match this area.

Statements. For a statement that is simply a rule r, we have explained above how to apply r to an area a where the pattern of r matches a. For a conditional statement $r : s$, we require all the rules in $r : s$ to have the *same* pattern, which we call the pattern of $r : s$. For a conditional statement $r : s$ and an area a where the pattern of $r : s$ matches a, the application of $r : s$ to a proceeds by first applying r to a; if that application succeeds, then $r : s$ succeeds (and s is ignored); otherwise the result of $r : s$ is the application of the statement s to a.

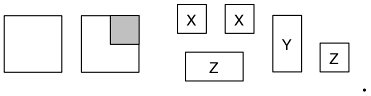
Programs. The execution of a program $s_1 \ldots s_n$ on a puzzle Pz proceeds as follows:

- For each i from 1 to n:
  - For each area a of Pz such that the pattern of s_i matches a:
    - ∗ apply s_i to a
    - ∗ if the application of s_i to a failed, then terminate the entire execution and report failure

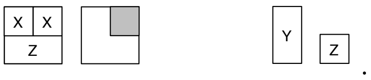
Example. Let us consider in detail the execution of the program



on the puzzle



The first statement has a pattern which matches only the first area of the puzzle. So, we apply the first statement to the first area, which succeeds and results in the following puzzle.



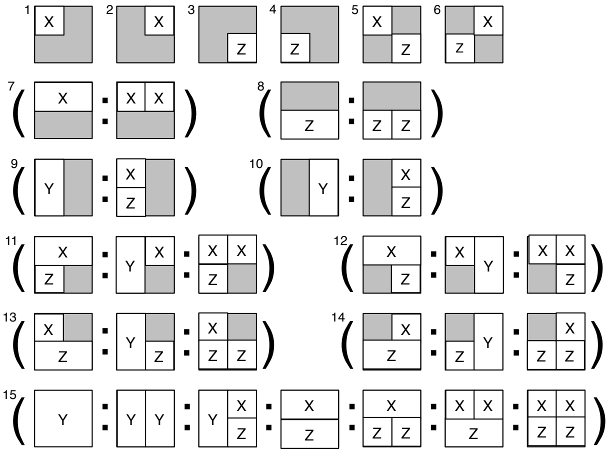


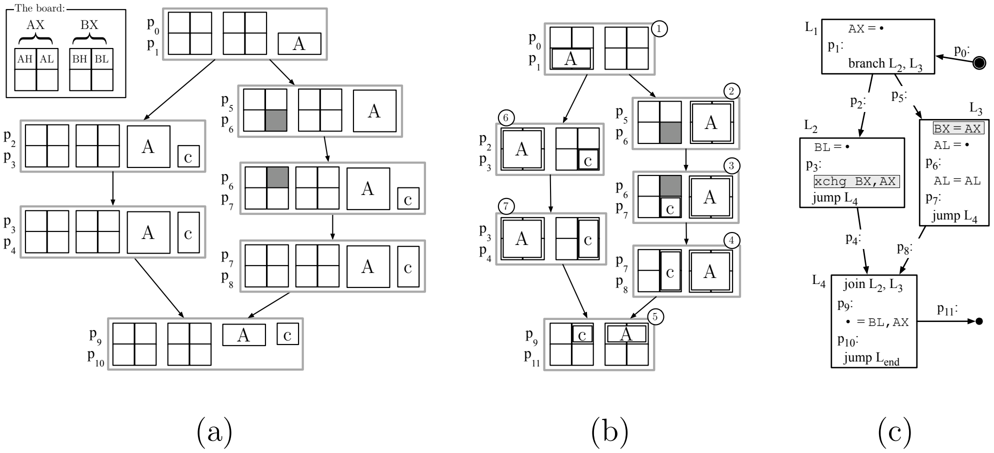
Figure 3.7: Our puzzle solving program

The second statement has a pattern which matches only the second area of the puzzle. So, we apply the second statement to the second area. The second statement is a conditional statement, so we first apply the first rule of the second statement. That rule fails because the pieces needed by the strategy of that rule are not available. We then move on to apply the second rule of the second statement. That rule succeeds and completes the puzzle.

Time Complexity. It is straightforward to implement the application of a rule to an area in constant time. A program executes $O(1)$ rules on each area of a board. So, the execution of a program on a board with K areas takes $O(K)$ time.

3.3.2 Our Puzzle Solving Program

Figure 3.7 shows our puzzle solving program, which has 15 numbered statements. Notice that the 15 statements have pairwise different patterns; each statement completes the areas with a particular pattern. While our program may appear

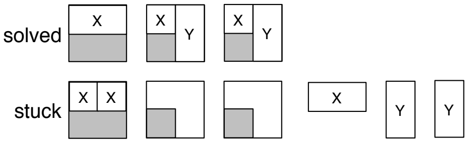


(a) (b) (c)

Figure 3.8: (a) The puzzles produced for the program given in Figure 3.3 (b). (b) An example solution. (c) The final program.

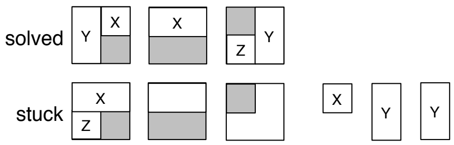
simple and straightforward, the ordering of the statements and the ordering of the rules in conditional statements are in several cases crucial for correctness. In general our program tries to fill the most constrained patterns first. For example, statements 1–8 can only be filled in one way, while the other statements admit two or more solutions. We will discuss four such subtleties.

First, it is imperative that in statement 7 our program prefers a size-2 X-piece over two size-1 X-pieces. Suppose we replace statement 7 with a statement 7′ which swaps the order of the two rules in statement 7. The application of statement 7′ can take us from a solvable puzzle to an unsolvable puzzle, for example:



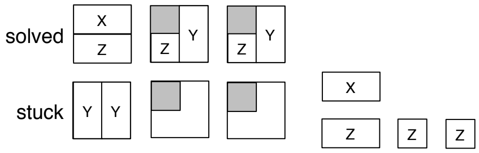
Because statement 7 prefers a size-2 X-piece over two size-1 X-pieces, the example is impossible. Notice that our program also prefers the size-2 pieces over the size-1 pieces in statements 8–15 for reasons similar to our analysis of statement 7.

Second, it is critical that statements 7–10 come before statements 11–14. Suppose we swap the order of the two subsequences of statements. The application of rule 11 can now take us from a solvable puzzle to an unsolvable puzzle, for example:



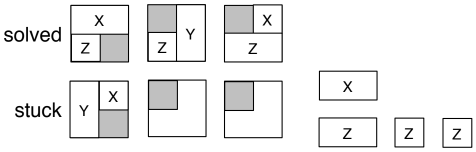
Notice that the example uses an area in which two squares are filled in. Because statements 7–10 come before statements 11–14, the example is impossible.

Third, it is crucial that statements 11–14 come before statement 15. Suppose we swap the order such that statement 15 comes before statements 11–14. The application of rule 15 can now take us from a solvable puzzle to an unsolvable puzzle, for example:



Notice that the example uses an area in which one square is filled in. Because statements 11–14 come before statement 15, the example is impossible.

Fourth, it is essential that in statement 11, the rules come in exactly the order given in our program. Suppose we replace statement 11 with a statement 11′ which swaps the order of the first two rules of statement 11. The application of statement 11′ can take us from a solvable puzzle to an unsolvable puzzle. For example:



When we use the statement 11 given in our program, this situation cannot occur. Notice that our program makes a similar choice in statements 12–14; all for reasons similar to our analysis of statement 11.

Theorem 2 ***(Correctness)*** *A type-1 puzzle is solvable if and only if our program succeeds on the puzzle.*

Proof. See Appendix A.2. □

For an elementary program P, we generate $|P|$ puzzles, each of which we can solve in linear time in the number of registers. So, we have Corollary 3.

Corollary 3 ***(Complexity)*** *Spill-free register allocation with pre-coloring for an elementary program P and $2K$ registers is solvable in $O(|P| \times K)$ time.*

A solution for the collection of puzzles in Figure 3.8 (a) is shown in Figure 3.8 (b). For simplicity, the puzzles in Figure 3.8 are not padded.

3.4 Spilling and Coalescing

We now present our approach to spilling and coalescing. Figure 3.9 shows the combined step of puzzle solving, spilling, and coalescing.

Spilling. If the polynomial-time algorithm of Theorem 3 succeeds, then all the variables in the program from which the puzzles were generated can be placed in registers. However, the algorithm may fail, implying that the need for registers exceeds the number of available registers. In that situation, the register allocator must spill variables.

We use a simple spilling heuristic. The heuristic is based on the observation that when we convert a program P into elementary form, each of P's variables is represented by a *family of variables* in the elementary program. For example, the variable c in Figure 3.3 (a) is represented by the family of variables $\{c_{23}, c_3, c_4, c_{67}, c_7, c_8, c_9\}$ in Figure 3.3 (b). When we spill a variable in an elementary program, we choose to simultaneously spill *all* the variables in its family, thereby reducing the number of pieces in many puzzles at the same time. The problem of *register allocation with pre-coloring and spilling of families of variables* is to perform register allocation with pre-coloring while spilling as few families of variables as possible.

Theorem 4 ***(Hardness)*** *Register allocation with pre-coloring and spilling of families of variables for an elementary program is NP-complete.*

Proof. See Appendix A.3. □

Theorem 4 justifies our use of a spilling heuristic rather than an algorithm that solves the problem optimally. Figure 3.9 contains a while-loop that implements the heuristic. It is straightforward to see that this method visits each puzzle once, that it always terminates, and that when it terminates, all puzzles have been solved.

In order to avoid separating registers to reload spilled variables only certain pieces can be removed from an unsolved puzzle. These pieces represent variables that are neither used nor defined in the instruction that gave origin to the puzzle. For instance, only the Y piece f can be removed from the puzzle in Figure 3.4. When choosing a piece to be removed from a puzzle, we use the "furthest-first"

```
• $S$ = empty
• For each puzzle $p$, in a preorder traversal of the dominator tree of the program:
    – while $p$ is not solvable:
        ∗ choose and remove a piece $s$ from $p$, and for every subsequent puzzle $p'$ that contains a variable $s'$ in the family of $s$, remove $s'$ from $p'$.
    – $S'$ = a solution of $p$, guided by $S$ according to the algorithm given in Chapter 4.
    – $S = S'$
```

Figure 3.9: Register allocation with spilling and local coalescing

strategy of Belady [6] that was later used by Poletto and Sarkar [77] in linear-scan register allocation. The furthest-first strategy spills a family of variables whose live ranges extend the furthest, according to a linearization determined by a depth first traversal of the dominator tree of the source program. We do not give preference to any path. Giving preference to a path would be particularly worthwhile when profiling information is available.

The total number of puzzles that will be solved during a run of our heuristic is bounded by $|P|+|F|$, where $|P|$ denotes the number of puzzles and $|F|$ denotes the number of families of variables, that is, the number of variables in the source program.

Coalescing. Traditionally, the task of register coalescing is to assign the same register to the variables x and y in a copy statement $x = y$, thereby avoiding the generation of code for that statement. An elementary program contains many parallel copy statements and therefore many opportunities for a form of register coalescing. We use an approach that we call *local coalescing*. The goal of local coalescing is to allocate variables in the same family to the same register, as much as possible. Local coalescing traverses the dominator tree of the elementary program in preorder and solves each puzzle guided by the solution to the *previous puzzle*, as shown in Figure 3.9. In Figure 3.8 (b), the numbers next to each puzzle denote the order in which the puzzles were solved.

The pre-ordering has the good property that every time a puzzle corresponding to statement i is solved, all the families of variables that are defined at program points that dominate i have already been given at least one location. The puzzle solver can then try to assign to the piece that represents variable v the same register that was assigned to other variables in v's family. For instance, in Figure 3.3(b), when solving the puzzle between p_2 and p_3, the puzzle solver tries to match the registers assigned to A_2 and A_3. This optimization is possible because A_2 is defined at a program point that dominates the definition site of A_3, and thus is visited before. For empty puzzle boards, the problem of maximizing the number of matches across two successive puzzles can be solved optimally, as we show in Chapter 4. As we show in Section 3.6, approximately 90% of the puzzles found in common benchmarks have an empty board initially.

Figure 3.8 (c) shows the assembly code produced by the puzzle solver for our running example. We have highlighted the instructions used to implement parallel copies. The x86 instruction `xchg` swaps the contents of two registers.

3.5 Implementation Details

In this section we describe some implementation details that we found useful in our register allocator. In addition to these particularities, our puzzle solver uses three optimizations that are novel: store hoisting, load lowering and redundant memory transfer elimination. These optimizations are explained in Section 5.6.1.

Bin-packing allocation. During the traversal of the dominator tree, the physical location of each live variable is kept in a vector. If a spilled variable is reloaded when solving a puzzle, it stays in a register until another puzzle, possibly many instructions after the reloading point, forces it to be evicted again. Our approach to handling reloaded variables is somewhat similar to the second-chance allocation described by Traub *et al.* [87].

Size of the intermediate representation. An elementary program has many more variable names than an ordinary program; fortunately, we do not have to keep any of these extra names. Our solver uses only one puzzle board at any time: given an instruction i, variables alive before and after i are renamed when the solver builds the puzzle that represents i. Once the puzzle is solved, we use its solution to rewrite i and we discard the extra names. The parallel copy between two consecutive instructions i_1 and i_2 in the same basic block can be implemented right after the puzzle representing i_2 is solved.

Critical Edges and Conventional SSA-form. Before solving puzzles, our algorithm performs two transformations in the target control flow graph that, although not essential to the correctness of our allocator, greatly simplify the elimination of ϕ-functions and π-functions. The first transformation, commonly described in the literature, (see [14, p.873]), removes critical edges from the control flow graph. These are edges between a basic block with multiple successors and a basic block with multiple predecessors. The second transformation converts the target program into a variation of SSA-form called *Conventional SSA-form* (CSSA) [84]. A fast algorithm to perform the SSA-to-CSSA conversion is given in [21]. These two transformations are necessary during the so called SSA elimination phase, that replaces parallel copies, including ϕ and π functions, with ordinary assembly instructions. Our SSA elimination method is described in Chapter 5.

Implementing ϕ-functions and π-functions. The allocator maintains a table with the solution of the first and last puzzles found in each basic block. These solutions are used to guide the elimination of ϕ-functions and π-functions, which is performed by the algorithm **ImplementComponent** from Section 5.5. During the implementation of parallel copies, the ability to swap register values is necessary to preserve the register pressure found during the register assignment phase [11, 74]. Some architectures, such as x86, provide instructions to swap the values in registers. In systems where this is not the case, swaps can be performed using xor instructions.

First-chance Coalescing We will say that two variables are ϕ-related if they are syntactically related by a common ϕ-function. The ϕ-relation is formally defined in Section 5.3. For instance, an instruction such as $v = \phi(\ldots, u, \ldots, w, \ldots)$ would cause the variables u, v and w to be ϕ-related. We will call *Global Coalescing* the optimization that attempts to minimize the number of physical registers assigned to families of ϕ-related variables. In our case it is always safe to assign the same physical register to ϕ-related variables, because the CSSA-form guarantees that these variables cannot interfere.

Our instance of the global coalescing problem is NP-complete, as proved by Bouchez *et al.* [12]. Thus, in order to preserve the fast compilation time, we use a biased coloring heuristics to implement a limited form of global coalescing. In our representation, every variable is part of an equivalence class of ϕ-related names. If a variable is never defined nor used by any ϕ-function, then its equivalence class is a singleton. We associate to each non-singleton ϕ-equivalence class a *preferred slot*, which is the position on the puzzle board that is allocated to most of the elements in the same ϕ-equivalence class. Our puzzle solver attempts to match Z pieces with preferred slots whenever it does not compromise the optimal solution of the puzzle. Notice that the preferred slot of a class might change during the register assignment phase, as it follows an on-line majority rule. Because the number of preferences is finite, e.g K, reading and updating the preferred slot is a constant time operation, and it requires $O(K)$ space per ϕ-equivalence class, where K is the number of registers available. This is a standard algorithm for computing on-line majority [13].

3.6 Experimental Results

Experimental platform. We have implemented our register allocator in the LLVM compiler framework [58], version 1.9. LLVM is the JIT compiler in the openGL stack of Mac OS 10.5. Our tests are executed on a 32-bit x86 Intel(R) Xeon(TM), with a 3.06GHz cpu clock, 3GB of free memory (as shown by the linux command `free`) and 512KB L1 cache running Red Hat Linux 3.3.3-7.

Benchmark characteristics. The LLVM distribution provides a broad variety of benchmarks: our implementation has compiled and run over 1.3 million lines of C code. LLVM 1.9 and our puzzle solver pass the same suite of benchmarks. In this section we will present measurements based on the SPEC CPU2000 benchmarks. Some characteristics of these benchmarks are given in Figure 3.10. All the figures use short names for the benchmarks; the full names are given in

Figure 3.10. We order these benchmarks by the number of non-empty puzzles that they produce, which is given in Figure 3.12.

Puzzle characteristics. Figure 3.11 counts the types of puzzles generated from SPEC CPU2000. A total of 3.45% of the puzzles have pieces of different sizes plus pre-colored areas so they exercise all aspects of the puzzle solver. Most of the puzzles are simpler: 5.18% of them are empty, *i.e.*, have no pieces; 58.16% have only pieces of the same size, and 83.66% have an empty board with no pre-colored areas. Just 226 puzzles contained only short pieces with precolored areas and we omit them from the chart.

As we show in Figure 3.12, 94.6% of the nonempty puzzles in SPEC CPU2000 can be solved in the first try. When this is not the case, our spilling heuristic allows for solving a puzzle multiple times with a decreasing number of pieces until a solution is found. Figure 3.12 reports the average number of times that the puzzle solver had to be called per nonempty puzzle. On average, we solve each nonempty puzzle 1.05 times.

Number of moves/swaps inserted by the puzzle solver. Figure 3.13 shows the number of copy and swap instructions inserted by the puzzle solver in each of the compiled benchmarks. Local copies denote instructions used by the puzzle solver to implement parallel copies between two consecutive puzzles inside the same basic block. Global copies denote instructions inserted into the final program during the SSA-elimination phase in order to implement ϕ-functions and π-functions. Target programs contains one copy or swap per each 14.7 puzzles in the source program, that is, on average, the puzzle solver has inserted 0.025 local and 0.043 global copies per puzzle.

The effect of First-Chance Global Coalescing. Figure 3.14 outlines static improvements on the code produced by the puzzle solver due to the global

	Benchmark	`LoC`	`asm`	`btcode`
gcc	176.gcc	224,099	12,868,208	2,195,700
plk	253.perlbmk	85,814	7,010,809	1,268,148
gap	254.gap	71,461	4,256,317	702,843
msa	177.mesa	59,394	3,820,633	547,825
vtx	255.vortex	67,262	2,714,588	451,516
twf	300.twolf	20,499	1,625,861	324,346
crf	186.crafty	21,197	1,573,423	288,488
vpr	175.vpr	17,760	1,081,883	173,475
amp	188.ammp	13,515	875,786	149,245
prs	197.parser	11,421	904,924	163,025
gzp	164.gzip	8,643	202,640	46,188
bz2	256.bzip2	4,675	162,270	35,548
art	179.art	1,297	91,078	40,762
eqk	183.equake	1,540	91,018	45,241
mcf	181.mcf	2.451	60,225	34,021

Figure 3.10: Benchmark characteristics. `LoC`: number of lines of C code. `asm`: size of x86 assembly programs produced by LLVM with our algorithm (bytes). `btcode`: program size in LLVM's intermediate representation (bytes).

coalescing heuristics described in Section 3.5. On average, first-chance coalescing reduces the number of memory accesses, mostly store instructions, by 3.4%, and the number of move instructions by 7.1%. Notice that this form of coalescing does not reduce the number of spilled variables, although it reduces the number of spill instructions.. The reduction in spill instructions happens during the elimination of π and ϕ-functions, because there are more variables in the same

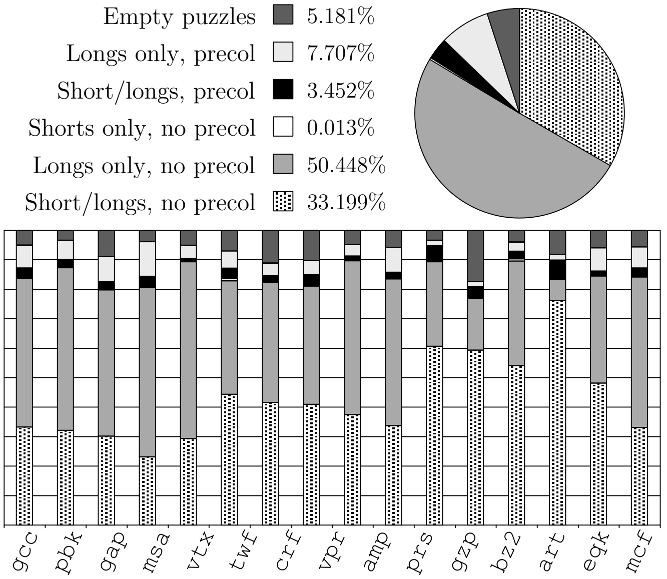


Figure 3.11: The distribution of the 1,486,301 puzzles generated from SPEC CPU2000.

physical location across basic blocks.

Three other register allocators. We compare our puzzle solver with three other register allocators, all implemented in LLVM 1.9 and all compiling and running the same benchmark suite of 1.3 million lines of C code. The first is LLVM's default algorithm, which is an industrial-strength version of linear scan that uses extensions by Wimmer *et al.* [90] and Evlogimenos [34]. The algorithm does aggressive coalescing before register allocation and handles holes in live ranges by filling them with other variables whenever possible. We use ELS (Extended Linear Scan) to denote this register allocator.

The second register allocator is the iterated register coalescing of George and Appel [40] with extensions by Smith, Ramsey, and Holloway [83] for handling register aliasing. We use EIRC (Extended Iterated Register Coalescing) to denote this register allocator. The third register allocator is based on partitioned

Benchmark	#puzzles	`avg`	`max`	`once`
gcc	476,649	1.03	4	457,572
perlbmk	265,905	1.03	4	253,563
gap	158,757	1.05	4	153,394
mesa	139,537	1.08	9	125,169
vortex	116,496	1.02	4	113,880
twolf	60,969	1.09	9	52,443
crafty	59,504	1.06	4	53,384
vpr	36,561	1.10	10	35,167
ammp	33,381	1.07	8	31,853
parser	31,668	1.04	4	30,209
gzip	7,550	1.06	3	6,360
bzip2	5,495	1.09	3	4,656
art	3,552	1.08	4	3,174
equake	3,365	1.11	8	2,788
mcf	2,404	1.05	3	2,120
	1,401,793	1.05	10	1,325,732

Figure 3.12: Number of calls to the puzzle solver per nonempty puzzle. #puzzles: number of nonempty puzzles. `avg` and `max`: average and maximum number of times the puzzle solver was used per puzzle. `once`: number of puzzles for which the puzzle solver was used only once.

Boolean quadratic programming (PBQP), and is implemented after Hames and Scholz [49]. We use this algorithm to gauge the potential for how good a register allocator can be. Lang Hames and Bernhard Scholz produced the implementations of EIRC and PBQP that we are using.

Stack size comparison. The top half of Figure 3.15 compares the maximum

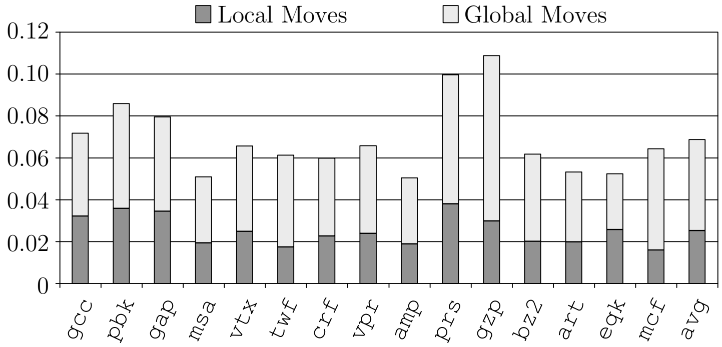


Figure 3.13: Number of copy and swap instructions inserted per puzzle.

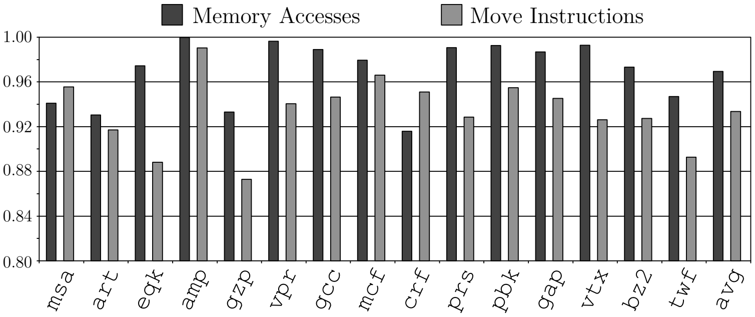


Figure 3.14: Static improvement due to First-Chance Coalescing. The bars are normalized to register allocation without global coalescing.

amount of space that each assembly program reserves on its call stack. The stack size gives an estimate of how many different variables are being spilled by each allocator. The puzzle solver and extended linear scan (LLVM's default) tend to spill more variables than the other two algorithms.

Spill-code comparison. The bottom half of Figure 3.15 compares the number of load/store instructions in the assembly code. The puzzle solver inserts marginally fewer memory-access instructions than PBQP, 1.2% fewer memory-access instructions than EIRC, and 9.6% fewer memory-access instructions than

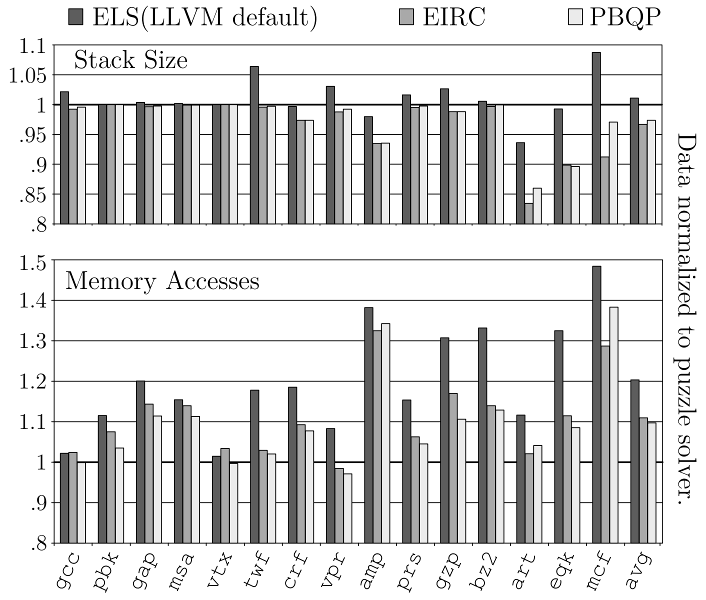


Figure 3.15: In both charts, the bars are relative to the puzzle solver; shorter bars are better for the other algorithms. **Stack size:** Comparison of the maximum amount of bytes reserved on the stack. **Number of memory accesses:** Comparison of the total static number of load and store instructions inserted by each register allocator.

extended linear scan (LLVM's default). Note that although the puzzle solver spills more variables than the other allocators, it removes only part of the live range of a spilled variable.

Run-time comparison. Figure 3.16 compares the run time of the code produced by each allocator. Each bar shows the average of five runs of each benchmark; smaller is better. The base line is the run time of the code when compiled with gcc -O3 version 3.3.3. Note that the four allocators that we use (the puzzle solver, extended linear scan (LLVM's default), EIRC and PBQP) are implemented in LLVM, while we use gcc, an entirely different compiler, only for

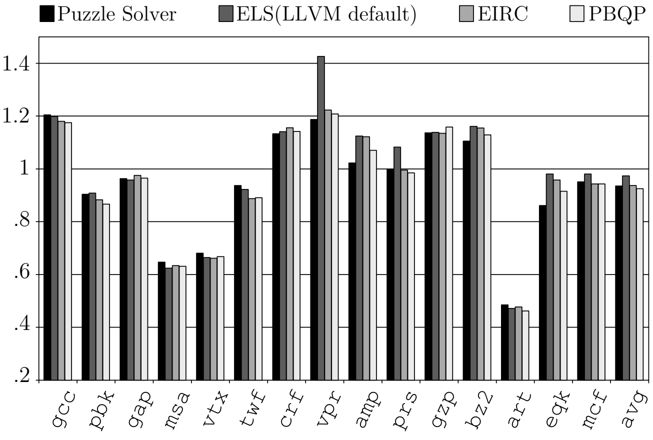


Figure 3.16: Comparison of the running time of the code produced with our algorithm and other allocators. The bars are relative to gcc -O3; shorter bars are better.

reference purposes. Considering all the benchmarks, the four allocators produce faster code than gcc; the fractions are: puzzle solver 0.944, extended linear scan (LLVM's default) 0.991, EIRC 0.954 and PBQP 0.929. If we remove the floating point benchmarks, *i.e.*, `msa, amp, art, eqk`, then gcc -O3 is faster. The fractions are: puzzle Solver 1.015, extended linear scan (LLVM's default) 1.059, EIRC 1.025 and PBQP 1.008. We conclude that the puzzle solver produces faster code than the other polynomial-time allocators, but slower code than the time demanding PBQP.

We have found that the puzzle solver does particularly well on sparse control-flow graphs. We can easily find examples of basic blocks where the puzzle solver outperforms even PBQP, which is a slower algorithm. For instance, with two register pairs (`AL, AH, BL, BH`) available, the puzzle solver allocates the program in Figure 3.17 without spilling, while the other register allocators (ELS, EIRC and PBQP) spill at least one variable. In this example, the puzzle solver inserts

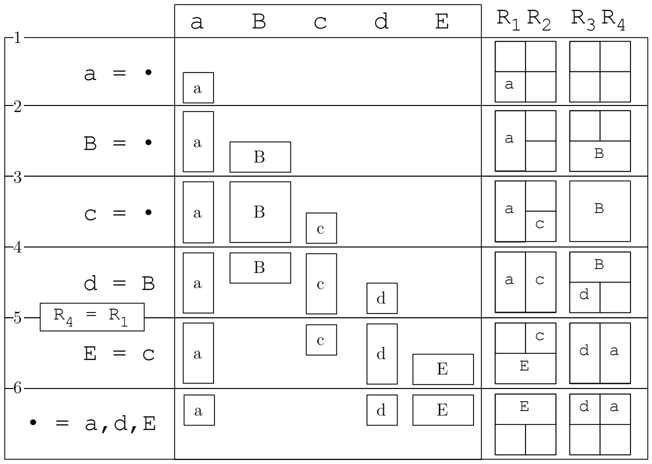


Figure 3.17: (left) Example program. (center) Puzzle pieces. (right) Register assignment.

one copy between instructions four and five to split the live range of variable *a*.

Compile-time comparison. Figure 3.18 compares the register allocation time and the total compilation time of the puzzle solver and extended linear scan (LLVM's default). On average, extended linear scan (LLVM's default) is less than 1% faster than the puzzle solver. The total compilation time of LLVM with the default allocator is less than 3% faster than the total compilation time of LLVM with the puzzle solver. We note that LLVM is industrial-strength and highly tuned software, in contrast to our puzzle solver.

We omit the compilation times of EIRC and PBQP because the implementations that we have are research artifacts that have not been optimized to run fast. Instead, we gauge the relative compilation speeds from statements in previous papers. The experiments shown in [49] suggest that the compilation time of PBQP is between two and four times the compilation time of extended iterated register coalescing. The extensions proposed by Smith *et al.* [83] can be implemented in a way that adds less than 5% to the compilation time of a graph-coloring allocator.

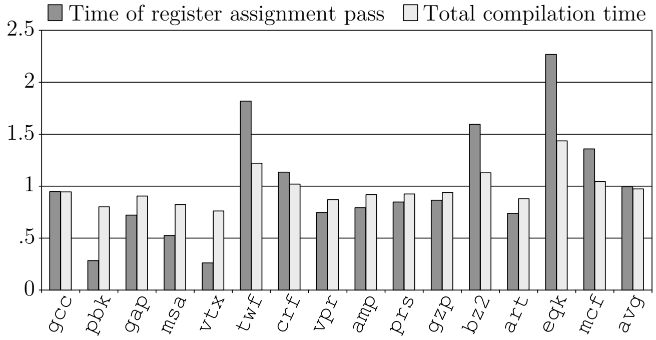


Figure 3.18: Comparison between compilation time of the puzzle solver and extended linear scan (LLVM's default algorithm). The bars are relative to the puzzle solver; shorter bars are better for extended linear scan.

Timing comparisons between graph coloring and linear scan (the core of LLVM's algorithm) span a wide spectrum. The original linear scan paper [77] suggests that graph coloring is about twice as slow as linear scan, while Traub *et al.* [87] gives an slowdown of up to 3.5x for large programs, and Sarkar and Barik [81] suggests a 20x slowdown. From these observations we conclude that extended linear scan (LLVM's default) and our puzzle solver are significantly faster than the other allocators.

3.7 Related Work

We now discuss work on relating programs to graphs and on complexity results for variations of graph coloring. Figure 3.21 summarizes most of the results.

Register allocation and graphs. Figure 3.19 shows the interference graph of the elementary program in Figure 3.3 (b). Any graph can be the interference graph of a general program [24]. SSA-form programs have chordal interference graphs [10, 19, 48, 73], and the interference graphs of SSI-form programs are

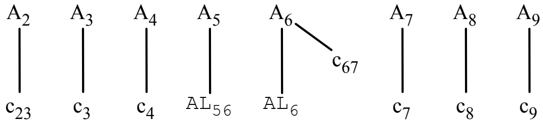


Figure 3.19: Interference graph of the program in Figure 3.3 (b).

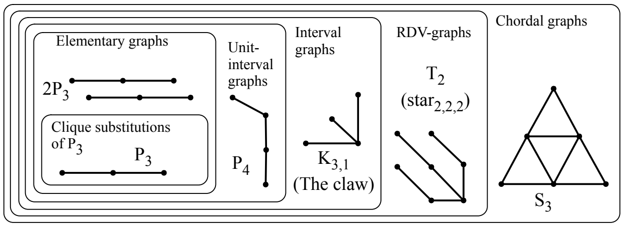


Figure 3.20: Elementary graphs and other intersection graphs. RDV-graphs are intersection graphs of directed lines on a tree [63].

interval graphs [20]. We call the interference graph of an elementary program an *elementary graph*. Each connected component of an elementary graph is a clique substitution of P_3, the simple path with three nodes. We construct a clique substitution of P_3 by replacing each node of P_3 by a clique, and connecting all the nodes of adjacent cliques.

Elementary graphs are a proper subset of interval graphs, which are contained in the class of chordal graphs. Figure 3.20 illustrates these inclusions. Elementary graphs are also *Trivially Perfect Graphs* [41]. In a trivially perfect graph, the size of the maximal independent set equals the size of the number of maximal cliques.

Aligned 1-2-coloring Extension. The combination of 1-2-aligned coloring and pre-coloring extension is called *aligned 1-2-coloring extension*. This problem, when restricted to elementary graphs, is equivalent to solving type-1 puzzles;

	Class of graphs			
Program	general	SSA-form	SSI-form	elementary
Problem	general	chordal	interval	elementary
Aligned 1-2-coloring extension	NP-complete [53]	NP-complete [8]	NP-complete [8]	linear [TD]
Aligned 1-2-coloring	NP-complete [53]	NP-complete [59]	NP-complete [59]	linear [TD]
Coloring extension	NP-complete [53]	NP-complete [8]	NP-complete [8]	linear [TD]
Coloring	NP-complete [53]	linear [38]	linear [38]	linear [38]

Figure 3.21: Algorithms and hardness results for graph coloring. TD = this dissertation.

thus, it has a polynomial time solution.

3.8 Final Remarks

In this chapter we have introduced register allocation by puzzle solving. We have shown that our puzzle-based allocator runs as fast as the algorithm used in an industrial-strength JIT compiler and that it produces code that is competitive with state-of-the-art algorithms. A compiler writer can model a register file as a puzzle board, and easily transform a source program into elementary form and then into puzzle pieces. For a compiler that already uses SSA-form as an intermediate representation, the extra step to elementary form is small. Our puzzle solver works for architectures such as x86, ARM, and PowerPC. Recently Jens Palsberg and Siddharth Tiwary showed that the aligned 1-2 graph coloring problem has polynomial time solution for type-2 puzzles. Their proof demonstrates that spill-free register allocation has polynomial time solution also for member of the SPARC family of processors. This work has not yet been published. Our puzzle solver produces competitive code even though we use simple approaches to spilling and coalescing. We speculate that if compiler writers implement a puzzle solver with advanced approaches to spilling and coalescing, then the produced code will be even better.

CHAPTER 4

Local Coalescing

In this chapter we present a exact polynomial time algorithm for the problem of local coalescing. Given a solved register allocation puzzle, which we will call the guider, local coalescing is concerned in finding a solution to the next puzzle, the follower. This solution must minimize the number of instructions that are inserted between guider and follower to preserve the semantics of the target program. We can solve this problem exactly for type-1 puzzles if the follower has an empty board, what happens in about 90% of the puzzles found in SPEC CPU 2000. Local coalescing in nonempty boards and higher order puzzles remains an open problem. Global coalescing, that is, the problem of minimizing the number of instructions inserted across the whole program is NP-complete, even for straight line code.

4.1 Introduction

As we discussed in Section 2.1.4, coalescing is the act of assigning the same register to variables related by copy instructions. When doing register allocation in elementary form, coalescing has a broader meaning: it consists in assigning the same registers to variables in the same family across two successive puzzles, as we discussed in Section 3.4. If a variable is assigned to the same register across two consecutive puzzles, we call it a *fixed point*. As an example, the solution found

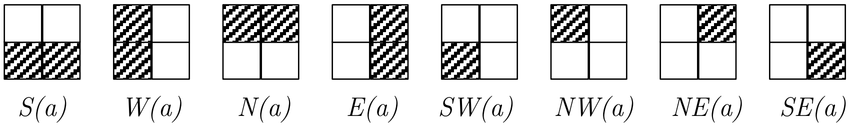


Figure 4.1: The different parts of the puzzle board.

by the puzzle solver for the program in Figure 3.17 contains eleven fixed points. *Local coalescing* is the problem of maximizing the number of fixed points in a puzzle, which is called the follower, given a solution to the predecessor puzzle, which is called the guider. Put in other words, local coalescing can be defined as the problem of minimizing the number of instructions that the puzzle solver must insert between two puzzles in order to preserve the semantics of the source program. Continuing with our example from Figure 3.17, that register assignment is an optimal solution to the local coalescing problem, as an exhaustive search shows that no register assignments in this case can produce a solution with less than one copy. The objective of this chapter is to show that, in the absence of pre-coloring, we can always find an optimal solution for local coalescing.

We will be using the notation shown in Figure 4.1 in order to name the blocks inside an area of the puzzle board. Blocks are named South (S), West (W), North (N) and East (E) according to their position inside the puzzle area.

4.2 Solving Uncolored Puzzles

The puzzle solving program given in Figure 3.7 uses the statement shown in Figure 4.2 to handle puzzles that do not contain pre-colored areas on the board. We will call this program P_u, and we will number its rules from one to seven, as shown in Figure 4.2. The objective of this section is to show that the order in

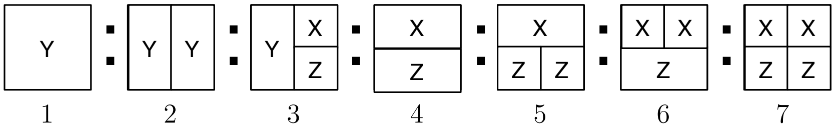


Figure 4.2: Program P_u that solves puzzles containing no pre-coloring.

which the rules are applied to the puzzle is not important.

Lemma 5 *Let Pz be a solvable puzzle. If the application of any rule r of P_u on Pz succeeds, then the resulting puzzle Pz' is solvable.*

Proof. If Pz is solvable, then there must exist a function S that maps pieces to areas such that S is a solution of Pz. In Figure 4.2 we see all the possible ways in which S can fill areas of Pz. In each of these configurations, size-1 pieces always come in pairs. For each rule r, we must show that its application does not invalidate preservation. We will describe in details rule 7, as it is the most involved, and we will illustrate the cases for rules 2-5 in Figures 4.4 and 4.5. Rule 1 is immediate, as there is only one way in which S can arrange a size-4 Y piece, and rule 6 is similar to rule 5.

($r = 7$): If r succeeds, then Pz contains an empty area, two size-1 X pieces (p_1 and p_2), and two size-1 Z pieces (p_3 and p_4). There are only a finite number of ways in which S can place the pieces p_1, p_2, p_3 and p_4 on the board. If S fills any area with four size-1 pieces, we simply swap these pieces with $\{p_1, p_2, p_3, p_4\}$. Otherwise, Figure 4.3 illustrates the remaining possible configurations. In each possible case, we can swap pieces in order to obtain the configuration produced by rule 7. For instance, let a_1, a_2 and a_3 be areas of the puzzle board. If p_1 is paired with another size-1 X piece in a_1, then we have that either p_4 is paired with a

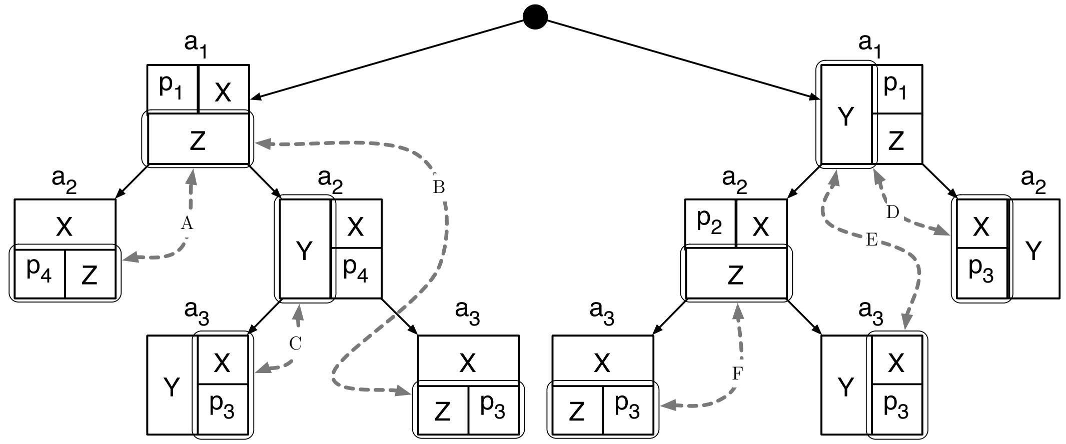


Figure 4.3: The case $r = 7$ of Lemma 5.

size-1 Z piece in a_2, or p_4 is paired with another size-1 X piece in a_2. In the former case, a swap between $S(a_1)$ and $S(a_2)$ produces the intended configuration, as illustrated by the dashed line A in Figure 4.3. Otherwise, we have two more cases to consider, depending on the configuration of p_3. If p_3 is paired with another size-1 X piece in a_3, then we use swap C in Figure 4.3, otherwise p_3 must be paired with a size-1 Z piece in a_3, and we use swap B. The case in which p_1 is paired with a Z piece in a_1 is handled in a similar way. The possible swaps, in this case, are labeled D, E and F in Figure 4.3. □

4.3 Optimal Local Coalescing

A good puzzle-based register allocator should strive to maximize the number of variables assigned to the same board positions across successive puzzles. If a piece is assigned to different board positions in two successive puzzles, then either a copy or a swap instruction will be necessary to preserve the semantics of the

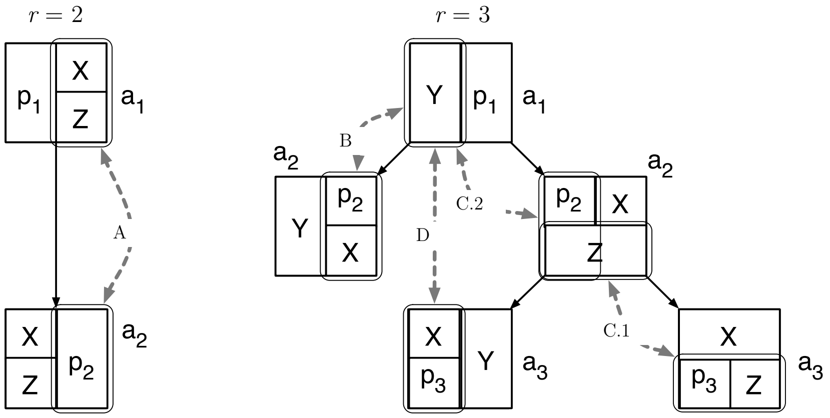


Figure 4.4: The cases $r = 2$ (left) and $r = 3$ (right) of Lemma 5.

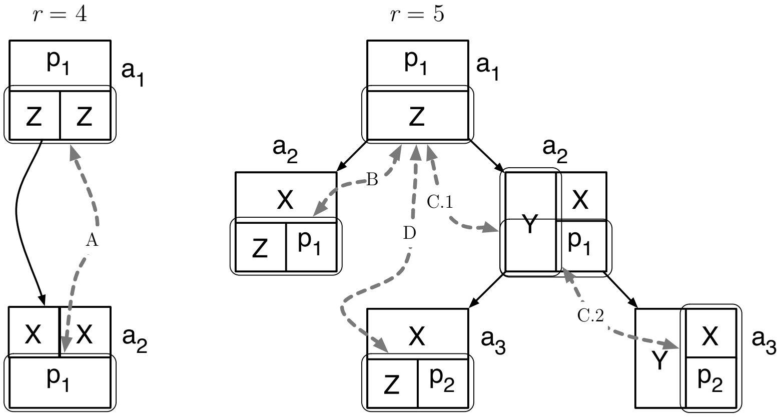


Figure 4.5: The cases $r = 4$ (left) and $r = 5$ (right) of Lemma 5. The case $r = 6$ is similar to $r = 5$ and was omitted.

target code. If a piece v is assigned to area a in the guider, then we say that a is the preferred area for v in follower. For instance, in Figure 3.17, area R2 is the preferred position for piece c in puzzle 4, because c was assigned to R2 in puzzle 3. Y pieces always have preferred areas, whereas Z pieces never have it. The X pieces used in the padding of the puzzle do not have preferred areas, but the X

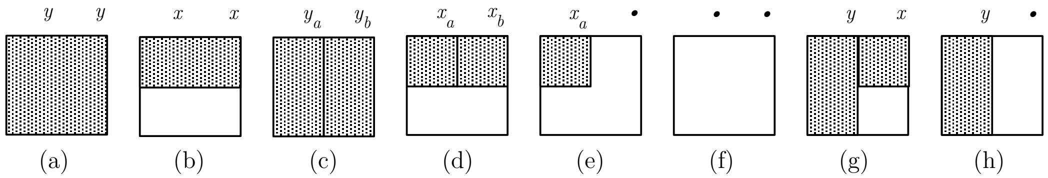


Figure 4.6: Configurations of preferences in a board without pre-coloring.

pieces that are part of the original, unpadded puzzle always have it.

In order to tackle the program of local coalescing, we extend the notation introduced in Chapter 3 to include preferences between pieces and board areas. We show all the possible configurations of preferences in Figure 4.6. The dotted areas are not part of our notation for puzzle solving programs. They only indicate the shape of the preferred pieces for each area. In this figure, the two little indices above each area indicate the preference for the two columns that make that area.

If a puzzle piece has no preference, we call it an *anonymous piece*, to contrast with *labeled pieces* which have preference for some area. Anonymous pieces are marked with •, and labeled pieces are given the name of the variable that they represent. We will consider puzzle solving programs that solve one area of the board at a time. If the preferred area of a piece v is filled with a piece different from v, and v is still available to fill other areas, we remove the name of v and mark it as an anonymous piece. This step is illustrated in Figure 4.7.

Although Lemma 5 has shown that pre-colored puzzle boards can be filled in any order, some ordering is necessary to maximize the number of variables locally coalesced. This ordering is defined in the puzzle solving program P_c shown in Figure 4.8. Program P_c attempts to assign pieces to areas according to the preferences in the puzzle. For instance, statement one handles registers that

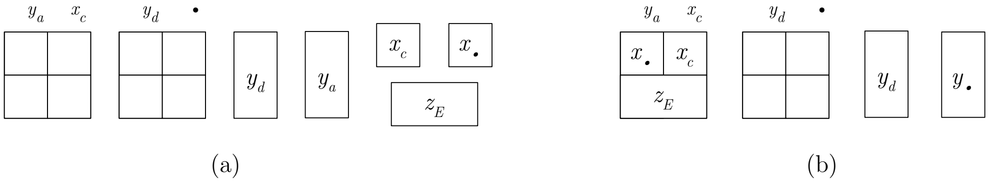


Figure 4.7: (a) The puzzle formed for the instruction `E := c` in Figure 3.17. (b) Puzzle after one area has been filled.

have been assigned to a long variable by the previous puzzle, e.g, this statement applies to registers `R1,R2` in the third puzzle of Figure 3.17. The successful execution of statement one means that we can assign a size-4 Y variable to its preferred registers. A simple inspection of the statements and rules of program P_c reveal that size-4 Y pieces and size-2 X pieces are always assigned to areas with matching preference. The rules that break local coalescing in Figure 4.8 are in statements seven and eight, where we have size-2 Z pieces displacing size-2 Y pieces, and size-2 Y pieces displacing size-1 X pieces.

Lemma 6 ***(Progress)*** *If Pz is solvable, then there is a rule $r \in P_c$ that can be applied to Pz.*

Proof. We will use the characters a, b, c and d to name rules in statements of program P_c that have multiple rules. For instance, the first rule of statement 2 will be called rule 2.a. We assume that Pz is produced from a original puzzle Pz_o by successive application of rules of P_c. A rule of P_c can be applied if its required area a and the required pieces $\{v_1, \ldots, v_n\}$ are available. If the application of some rule of P_c causes a piece v with a preferred area to become an anonymous piece, we say that v is *displaced.* The proof is a case analysis on the patterns p shown in Figure 4.6.

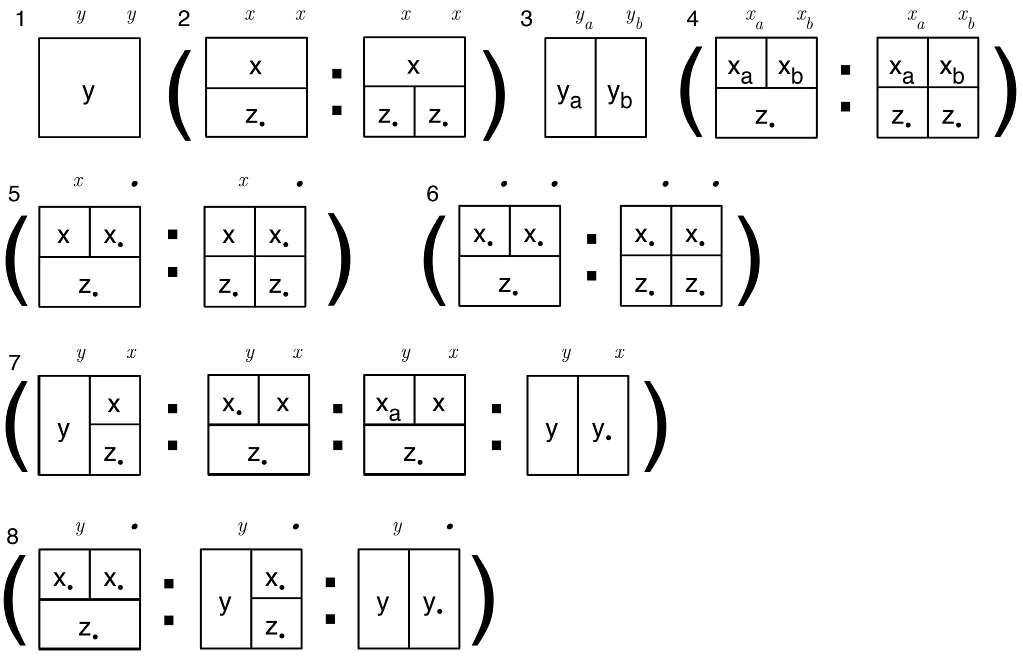


Figure 4.8: Program P_c that maximizes local coalescing.

($p = a$): this pattern is handled by statement one of P_c. An application of statement one can only be preceded by other applications of this statement. Y pieces always have a preferred area, and only areas with preferences are taken by statement one. Thus, the size-4 piece y must still be available in Pz.

($p = b$): this pattern is handled by statement 2 of P_c. Applications of statement two can only be preceded by applications of statements one and two. Piece x is available because all the rules used so far only assign pieces to their preferred areas. The size-2 Z piece, in case of rule 2.a, or the two size-1 Z pieces in case of rule 2.b exist because the puzzle is solvable.

($p = c$): similar to $p = a$. Pattern c is handled by statement three of P_c. Rules that precede statement three only assign pieces to their preferred areas; therefore, the two size-2 Y pieces must still be available in Pz.

($p = d$): similar to $p = b$.

($p = e$): these patterns are handled by statement five of P_c. We want to show that Pz contains a size-1 X piece $x_\bullet$ with no preference for any area, i.e, this piece was created due to padding (Claim 2). With this purpose, lets consider the first time when P_c uses statement five. The piece x_a must still be available, because all the rules that precede statement five only assign size-1 X pieces to their preferred areas. We must also show that Pz contains either one size-2 Z piece, or two size-1 Z pieces (Claim 1).

- Claim 1: *Pz contains either one size-2 Z piece or two size-1 Z pieces.* If we assume the contrary, then Pz must contain a size-2 Y piece plus a size-1 Z piece, because Pz is solvable by hypothesis. We will call this piece y_1. Y pieces always have a preferred area; thus, in a solution of Pz, the preferred area of y_1 must be covered with, either (i) another size-1 Y piece, (ii) one size-1 X piece plus one size-1 Z piece, or (iii) a size-2 Z piece plus a size-1 X piece. In case (i), we let the new size-2 Y piece be called y_2, and look at how the preferred area of y_2 is covered in a solution of Pz. Because the number of Y pieces is finite, we will have that some y_n piece is covered as in (ii) or (iii), which contradicts the initial assumption.

- Claim 2: *Pz contains one size-1 X piece with no preference for any area.* As seem in Claim 1, Pz contains either one size-2 Z piece or two size-1 Z pieces. Because Pz is a padded puzzle, it must contain a size-1 X piece x_1. By hypothesis of this claim, x_1 has preference for some area a_1. Because size-4 Y pieces and size-2 X pieces have been already placed by previous statements, a solution of Pz can cover a_1 in one of the following ways: (i) with a size-2 Y piece y_2 with preference for some area a_2, (ii) with a size-1 X piece x_2 with preference for some area a_2. We look at how a_2 is covered in a solution of Pz to find more pieces like in (i) or (ii). By repeating this argument indefinitely, we derive a contradiction, because the number of pieces in Pz is finite.

From Claim 1 and Claim 2 we know that statement five of P_c is always able to complete the pattern e of Figure 4.6. Moreover, from Claim 2 we know that no size-1 piece with a preference can be displaced by statement five, i.e, it is always possible to find a piece x_b without preferences to fill pattern e.

($p = f$): similar to $p = e$. We show that Pz either contains a size-2 Z piece or two size-1 Z pieces, as in Claim 1, and we show that Pz contains two size-1 X pieces, using the reasoning seen in Claim 2.

($p = g$): these patterns are handled by statement seven, which is divided into four rules: 7.a, 7.b, 7.c and 7.d. Statement seven contains all the three possible patterns that can be assembled with the pieces not used by statements 1-6. We only have to show that the preference requirements described in statement seven can be fulfilled with these remaining pieces. We look into the rules of statement seven in the order in which they can be applied to Pz.

- (7.a) Pz contains y and x , because no rule used before 7.a can displace such pieces. If Pz contains a size-1 Z piece, rule 7.a can be applied.

- (7.b) If Pz contains a size-2 Z piece, then it must contain two size-1 X pieces, as Pz is a solvable puzzle, and all the size-2 X pieces have been already assigned by statement two. Pz must contain x_b, because no size-1 piece with preference has been displaced by the rules that precede 7.b. If Pz contains a size-1 X piece without preference, rule 7.b can be applied.

- (7.c) If Pz contains a size-2 Z piece, then it must contain two size-1 X pieces, as Pz is a solvable puzzle, and all the size-2 X pieces have been already assigned by statement two. If Pz contains x_b, rule 7.c can be applied.

- (7.d) Pz contains y_a, because no size-2 Y piece with a preferred area has been allocated to an area that is not its preference so far. If $x \in Pz$, then Pz would contain a size-1 Z piece, as seen in the proof of rule 7.a. Given that Pz does not contain x, this piece must have been used in rule 7.c, that is the only rule that displaces size-1 X pieces. However, rule 7.c has also the effect of displacing a size-2 Y piece. Therefore, Pz contains a piece $y_\bullet$.

($p = h$): these patterns are handled by statement eight, which is divided into three rules: 8.a, 8.b, and 8.c. We have that all the size-1 X pieces that remain in Pz have no preference, because the pieces that had preference have either been assigned to their preferred areas by statements five and seven, or have been displaced in statement seven. We look into the rules of statement eight in the order in which they can be applied to Pz.

- (8.a) If Pz contains a size-2 Z piece, then it contains two size-1 X pieces, because Pz is solvable by hypothesis.

- (8.b) Pz must contain the size-2 Y piece y, because no size-2 Y piece with a preferred area has been used to cover an area that is not its preference so far. If Pz contains a size-1 X piece, it must contain a size-1 Z piece, because Pz is solvable by hypothesis.

- (8.c) If Pz contains only size-2 Y pieces, than we have that Pz contains n areas, and $2n$ pieces, because Pz is solvable by hypothesis. We have that half of these pieces still have their preferred areas available, because no size-2 Y piece with a preferred area available has been misplaced by any of the rules seen so far, and only patterns $p = h$ remain to be filled. The size-2 Y pieces without preference have been displaced by either statement seven (rules 7.a, 7.b and 7.c), or rule 8.a.

□

Lemma 7 ***(Preservation)*** *If Pz is solvable, and P_c applies rule r on Pz to produce Pz', then Pz is solvable.*

Proof. This follows directly from Lemma 5, as any of the patterns used in P_c is listed among the rules of P_u. □

Lemma 8 *If Pz can be solved with n displaced pieces, and rule r is applied producing Pz' and causing k displaced pieces, then Pz' can be solved with no more than $n - k$ displaced pieces.*

Proof. The proof is a case analysis on each rule of P_c. By hypothesis, we know that there exists a solution S for Pz that displaces n pieces. At each statement, we let the area been filled by that statement be called a. When swapping pieces of S in order to obtain a new solution S', we determine the profit of this swap as the number of displaced pieces in S minus the number of displaced pieces in S'.

$(r = 1)$: if $S(y) = a$, then we are done, otherwise, lets assume that $S(y) = a'$. We get a positive profit of at least +1 by swapping the contents of a and a'.

$(r = 2.a)$: if S does not assign x to a, then there must exist an area a' such that $S(x) = N(a')$. Furthermore, there must exist an area a" such that $S(z_\bullet) = a$". We get a better solution than S by performing the following sequence of swaps: swap $S(a')$ and $S(a")$, swap a' and a.

$(r = 2.b)$: in this case, Pz contains no size-2 Z piece. If S assigns x to an area $N(a')$ other than $N(a)$, we have that $S(a')$ must contain two size-1 Z pieces. We get a better solution than S by swapping the contents of a and a'.

($r = 3$): if S does not assign y_a and y_b to a, then we have that y_a has been assigned to an area a', and y_b has been assigned to an area a". Without loss of generality, we assume that y_a is assigned to $W(a')$, and y_b is assigned to $E(a")$. We get a better solution than S by performing the following sequence of swaps: swap $E(a')$ and $W(a")$ (may have a negative profit of -1), swap a and a' (has a positive profit of $\S + 2S$).

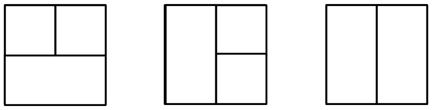
($r = 4.a$): similar to previous rules. Without loss of generality, we assume that $S(x_a) = NW(a_a)$, $S(x_b) = NE(a_b)$ and $S(z_\bullet) = S(a_z)$. We apply the following sequence of swaps: swap $NW(a_a)$ and $NW(a_z)$ (may have a negative profit of -1), swap $NE(a_b)$ and $NE(a_z)$ (may have a negative profit of -1), swap a_z and a (has a positive profit of $+2$).

($r = 4.b$): similar to $r = 4.a$, but Pz contains no size-2 X or Z pieces. Thus, if $S(x_a) = NE(a_a)$, we know that $SE(a_a)$ is a size-1 Z piece. Likewise, if $S(x_b) = NW(a_b)$, we know that $SW(a_b)$ is a size-1 Z piece. We apply the following sequence of swaps: swap a_a and a (may displace one piece in S, but balances this displacement by adding a match for x_a), swap $W(a_b)$ with $E(a)$ (positive profit of $+1$ because matches a_b).

($r = 5.a$): we assume that $S(z_\bullet) = S(a_z)$. Because Pz no longer contains size-2 X pieces, we have that $N(a_z)$ is filled with two size-1 X pieces. At least one of the pieces in $N(a_z)$ is anonymous, i.e $x_\bullet$, because areas with two preferences for size-1 X pieces have been handled by statement four. We swap $x_\bullet$ with x_a for a positive profit of at least $+1$.

($r = 5.b$): We assume that $S(z_\bullet) = S(a_z)$. Let $S(x) = NW(a_x)$. We let $S(x_\bullet) = NW(a_\bullet)$ for some area $a_\bullet \in Pz$. We first swap $E(a_x)$ and $W(a_\bullet)$. This swap may have a negative profit of at most -1. We then swap the contents of a_x and a, which has a positive profit of $+1$.

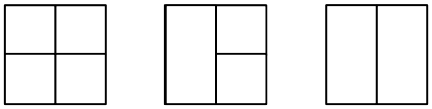
($r = 6.a$): we assume that $S(z_\bullet) = S(a_z)$. If a_z has no preferences, we swap the contents of a_z and a. Otherwise, a_z must be the pattern (g) in Figure 4.6. In this case, a_z might contain at most one matching size-1 X piece. The solution S may be covering a in one of the three different ways shown below:



(a) (b) (c)

In case of (a), we swap $S(a)$ and $S(a_z)$. In case of (b), we swap ($W(a)$ plus $SE(a)$) with ($NW(a_z)$ plus $S(a_z)$). In case of (c), we swap the contents of a and a_z. Case (c) may have a negative profit of -1; however, we can swap one of the size-2 Y pieces newly assigned to a_z with the size-2 Y piece that has the preference for a_z, given that this area is the pattern (g) in Figure 4.6.

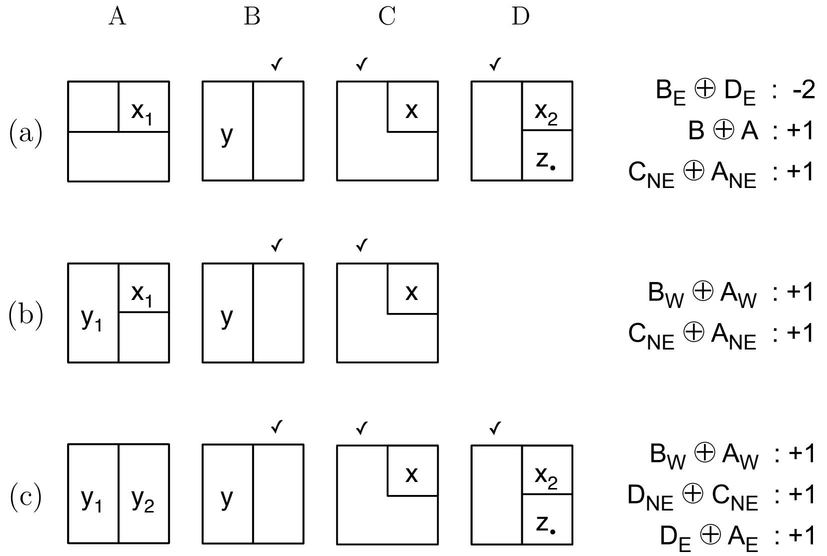
($r = 6.b$): in this case we do not have any size-2 X or Z piece in Pz. Again, S can arrange pieces in area a in three different ways, which are shown below:



(a) (b) (c)

There is nothing to be done in case (a). In case (b), we swap the size-2 Y piece with whatever pieces are occupying the preferred area for this piece. This swap is always possible because Pz contains no piece spanning two columns. This swap has a positive profit of $+1$. We repeat this procedure until a contains four size-1 puzzle pieces. Case (c) is similar to case (b).

($r = 7.a$): S may cover a in three different ways, and for each way we illustrate in the figure below how swaps can be inserted to convert S into S'. The pieces have been arranged in the board without any loss of generality, and we use the operator $\oplus$ to indicate a swap. The minimum profit of each swap is show next to its description. The little marks above some of the areas indicate possible, but not necessary, matchings in S.



($r = 7.b$): if rule 7.a cannot be used, it means that Pz does not contain any size-1 Z piece. In this case, either (i) a is covered with two size-2 Y pieces, e.g y and y_{aux}, or (ii) with one size-2 Z piece plus two size-1 X pieces. Showing that swaps on the second case do not produce a worse solution than S is immediate. On the first case, we have that x must be paired with a size-2 Z piece plus another size-1 X piece in some area a'. Given that the only patterns listed in Figure 4.6 that remain in our puzzle are (g) and (h), we know that a' can have at most one size-1 X piece matching its preference. In this case, a' is a (g) pattern. By swapping the contents of a and a' we may remove at most two matches from S, and we add one match for x - a minimum profit of -1. We then swap y_aux with the Y piece that has the preference for a' to produce a solution as good as S.

($r = 7.c$): same as $r = 7.b$. We have not used in the proof of $r = 7.b$ the fact that Pz contains a size-1 X piece with no preferred area.

($r = 7.d$): if Pz contains no Z pieces, it cannot contain X pieces either, otherwise Pz would be unsolvable. Furthermore, Pz contains only the patterns (g) and (h) of Figure 4.6. In this case it is trivial to show that the maximum number of matching pieces in any area a is one.

($r = 8.a, 8.b, 8.c$): all the preference patterns in Pz are like Figure 4.6(h). Each size-2 Z piece will displace a size-2 Y piece, and there must be at most one matching size-2 Y piece per area of Pz. Thus, the three rules that constitute statement eight of P_c are optimal.

□

4.3.1 On The Number of Pieces Displaced

The number of pieces displaced by the program P_c can be directly computed from the types of preference patterns found in the input puzzle, where the eight possible preference patterns are listed in Figure 4.6. We divide the puzzle board into groups according to the eight preference patterns. For instance, all the areas that fit the pattern in Figure 4.6(a) are placed into a set called P_a. The other seven patterns determine the sets $P_b, \ldots, P_h$. We let Z_2 be the set of size-2 Z pieces in a type-1 puzzle Pz.

The minimum number of pieces n_z displaced in a solution of a type-1 puzzle is determined uniquely by the types of patterns and the number of size-2 Z pieces found in the puzzle. The algorithm to compute n_d is given below:

- let $n_d = |Z_2| - (|P_b| + |P_d| + |P_e| + |P_f|)$
- if $n_d \leq 0$
  - then $n_z \leftarrow 0$

- else if $|P_h| \geq n_d$
  - then $n_z \leftarrow n_d$
  - else $n_z \leftarrow P_h + 2 \times (n_d - P_h)$

In order to see that the formula above produces the right number of displaced pieces, we point that, according to the program P_c, size-2 Z pieces are the only reason for displacements. This happens because the preferences for a given puzzle Pz_2 are determined by the arrangement of pieces in a previous puzzle Pz_1. Thus, if Pz_2 contains no size-2 Z pieces, all the pieces with preferred areas can be fit onto its board, given that they already fit on the board of Pz_1. P_c attempts to fit size-2 Z pieces in patterns of type P_b, P_d, P_e and P_f, which do not cause the displacement of any piece with a preferred area. We let n_d be the difference $|Z_2| - (|P_b| + |P_d| + |P_e| + |P_f|)$.

If n_d is greater than zero, the number of displacements depends on P_h - the number of areas with the preference pattern (h). Notice that P_c fills areas with the preference pattern (g) before than (h). If a size-2 Z piece is inserted into an area with preference pattern (g), then a size-2 Y piece y is displaced. If Pz contains an area a_h with the preference pattern (h), than y can be fit onto a_h without causing any further displacements. The same reasoning applies if Pz contains two areas of type (h) and one size-2 Z piece. Therefore, as long as Pz contains areas of type (h), a size-2 Z piece causes at most one displacement. Otherwise, if Pz contains only areas of type (g), any size-2 Z piece forces the displacement of at least two pieces.

4.4 Final Remarks

This chapter has presented an optimal strategy for implementing local coalescing in empty type-1 puzzle boards. Our method gives the maximum number of fixed points in about 90% of the puzzles found in SPEC CPU 2000. Optimal local coalescing for nonempty puzzle boards, and for puzzle boards of high order puzzles, e.g type-2 and up, remains an open problem.

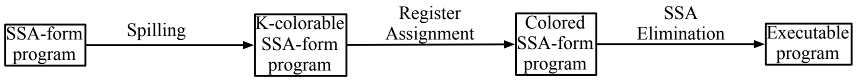
CHAPTER 5

SSA Elimination after Register Allocation

Intermediate representations such as SSA-form, or elementary form use notational abstractions called ϕ-functions and parallel copies. These instructions have no analogous in actual machine instruction sets, and they must be replaced by ordinary instructions at some point of the compilation path. This process is called *SSA elimination*. Compilers usually performs SSA elimination before register allocation. But the order could as well be the opposite: our puzzle based register allocator performs SSA elimination after register allocation. SSA elimination before register allocation is straightforward and standard, while the state-of-the-art approaches to SSA elimination after register allocation have several shortcomings. In this chapter we present *spill-free SSA elimination*, a simple and efficient algorithm for SSA elimination after register allocation that avoids increasing the number of spilled variables. We also present three optimizations that enhance the quality of the code produced by the core algorithm. We have implemented the algorithms described in this Chapter in our puzzle based register allocator. Our experiments show that spill-free SSA elimination takes less than five percent of the total compilation time of a JIT compiler. Our optimizations reduce the number of memory accesses by more than 9% and improve the program execution time by more than 1.8%.

5.1 Introduction

One of the main advantages of SSA based register allocation is the separation of phases between spilling and register assignment. The two-phase approach works because the number of registers needed for a program in SSA-form equals the maximum of the number of registers needed at any given program point. Thus spilling reduces to the problem of ensuring that for each program point, the needed number of registers is no more than the total number of registers. The register assignment phase can then proceed without additional spills. The next figure illustrates the phases of SSA-based register allocation.



SSA elimination *before* register allocation is easier than *after* register allocation. The reason is that after register allocation when some variables have been spilled to memory, SSA elimination may need to copy data from one memory location to another. The need for such copies is a problem for many computer architectures, including x86, that do not provide memory-to-memory copy or swap instructions. The problem is that at the point where it is necessary to transfer data from one memory location to another, all the registers may be in use! In that case, no register is available as a temporary location for performing a two-instruction sequence of a load followed by a store. One solution would be to permanently reserve a register to implement memory-to-memory transfers. We have evaluated that solution by reducing the number of available x86 integer registers from seven to six, and we observed an increase of 5.2% in the lines of spill code (load and store instructions) that LLVM [58] inserts in SPEC CPU 2000.

Brisk [17, Ch.13] has presented a flexible solution that spills a variable on demand during SSA elimination, uses the newly vacant register to implement memory transfers, and later reloads the spilled variable when a register is available. We are unaware of any implementation of Brisk's approach, but have gauged its potential quality by counting the minimal number of basic blocks where spilling would have to happen during SSA elimination in LLVM, independent on the assignment of physical locations to variables. For x86, such a basic block contains thirteen or more ϕ-functions. We found that for SPEC CPU 2000, memory-to-memory transfers are required for all benchmarks except `181.mcf` - the smallest program in the set. We also found that the lines of spill code must increase by at least 0.2% for SPEC CPU 2000, and we speculate that an implementation of Brisk's algorithm would reveal a higher number. In our view, the main problem with Brisk's approach is not the extra spill lines, but the fact that its second spilling phase substantially complicates the design of a register allocator.

This chapter describes an algorithm that improves on these two previous techniques. We will present *spill-free SSA elimination*, a simple and efficient algorithm for SSA elimination after register allocation. Spill-free SSA elimination never needs an extra register, entirely eliminates the need for memory-to-memory transfers, and avoids increasing the number of spilled variables. The next figure summarizes the three approaches to SSA elimination.

	Accommodates optimal register assignment	Avoids spilling during SSA elimination
Spare register	No	Yes
On-demand spilling [17]	Yes	No
Spill-free SSA elimination	Yes	Yes

The starting point for our approach to SSA-based register allocation is *Conventional SSA (CSSA)-form* [84] rather than the SSA form from the original paper [27] (and text books [4]). CSSA-form ensures that variables in the same ϕ-function do not interfere. We show how CSSA-form simplifies the task of replacing ϕ-functions with copy or swap instructions. We also assume that the CSSA-form program contains no *critical edges*. A critical edge is a control-flow edge from a basic block with multiple successors to a basic block with multiple predecessors. Algorithms for removing critical edges are standard [4].

This chapter also discusses three optimizations that are implemented on top of our SSA elimination algorithm. In the register allocator presented in Chapter 3 we convert the source program to CSSA-form before register assignment. Our experiments show that our approach to SSA elimination takes less than five percent of the total compilation time of LLVM. Our optimizations reduce the number of memory accesses by more than 9% and improve the program execution time by more than 1.8%.

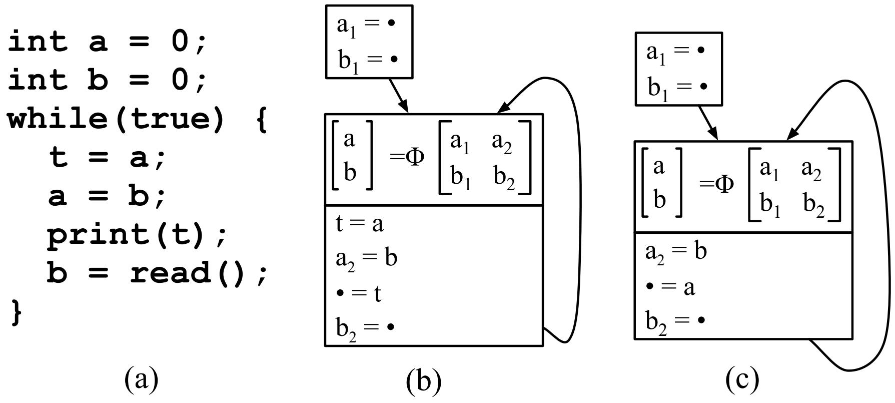
Our SSA elimination framework works for any SSA-based register allocator such as [48], but the implementation of ϕ-functions in SSA-based register allocators is not the only use of parallel copies in register allocation. The framework described in this chapter can also be used to insert the fixing code required by register allocators that follow the bin-packing model [56, 76, 81, 87]. Bin-packing allocators allow variables to reside in different registers at different program points. A variable may move between registers due to two main factors: to avoid interferences with pre-colored registers and due to high register pressure. The price of this flexibility is the necessity of inserting fixing code at basic block boundaries. The insertion of fixing code follows the same principles that rule the implementation of ϕ-functions in SSA-based register allocators.

5.2 Example

We now present an example that illustrates the main difficulty of doing SSA elimination after register allocation. Figure 5.1 (a) contains a program that continually reads values from the input and prints these values. We built the loop using a somehow artificial arrangement of the variables a, b and t in order to show how compiler optimizations might change the SSA representation of a program in a way that complicates the elimination of ϕ-functions. Figure 5.1 (b) shows the control flow graph of the example program, this time converted to SSA form. The program in Figure 5.1 (b) presents an interesting property: variables in the same ϕ-function, such as a, a_1 and a_2 never interfere. Programs that have this property are said to be in *Conventional Static Single Assignment* (CSSA)-form; this representation is formally defined in Section 5.3.

SSA elimination is very simple for programs in conventional SSA-form, as we will show in the remainder of this Chapter, but not every SSA-form program has the conventional property. The original SSA construction algorithm proposed by Cytron *et al.* [27] always builds CSSA-form programs; however, compiler optimizations might break the conventional representation. Figure 5.1 (c) shows the same program after it underwent a pass of copy propagation. This optimization replaced the use of variable t with a use of variable a, and removed the copy $t = a$. After this optimization, our example program is no longer in CSSA-form, because the variables a and a_2, which are part of the same ϕ-function, interfere.

We will be performing SSA elimination after register allocation. This means that each of our program variables will be bound to a physical location that can be either a machine register or a memory address. We call a program with such bindings a *colored program.* Figure 5.2 (a) shows a possible colored representation of our example program, assuming a target architecture with only one register r.



```
int a = 0;
int b = 0;
while(true) {
  t = a;
  a = b;
  print(t);
  b = read();
}
```

Figure 5.1: (a) Example program in high level language. (b) Control flow of program converted to SSA-form. (c) Program after constant propagation.

As we have seen in Section 2.3, each ϕ-matrix encodes one parallel copy per column. Thus, in order to perform SSA elimination on colored programs we must implement the parallel copies between physical locations. Figure 5.2 (b) shows the two parallel copies that we must implement in our running example: if control reaches block B_2 coming from block B_1, then the parallel copy $(r, m) := (r, m)$, which is a no-op, must be implemented, otherwise the parallel copy $(r, m) := (r, m_2)$ must be implemented. SSA elimination algorithms normally replace these parallel copies by inserting sequential instructions in the program points where the parallel copies are defined. Notice that this is the most natural approach to SSA elimination; however, the replacement code could be inserted anywhere inside the source program, as long as it maintains the program's semantics.

Figure 5.2 (c) shows our example program after SSA elimination with on-demand spilling. Notice that one of the parallel copies has been replaced with four instructions that implement a copy from m_2 to m. The need for that copy happens at a program point where the only register r is occupied by b_2. So we

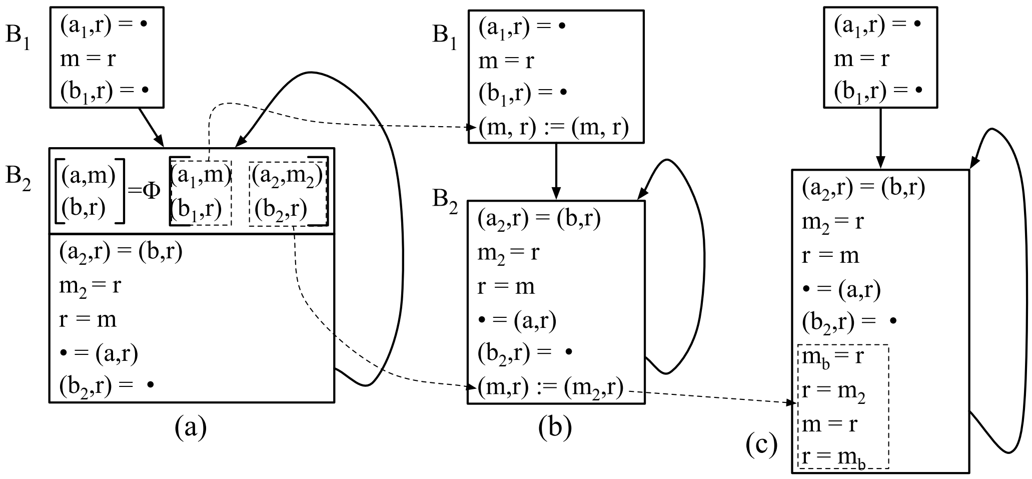


Figure 5.2: (a) A possible colored representation of the example program. (b) SSA elimination seen as the implementation of parallel copies. (c) SSA elimination with on-demand spilling.

must first spill r to m_b, then we can copy from m_2 to m via the register r, and finally we can load m_b back into r.

Now we go on to illustrate that spill-free SSA elimination can do better. Figure 5.3 (a) shows the same program as in Figure 5.2 (a), but this time in CSSA-form. To convert the source program into CSSA-form we had to split the live range of variable a_2; this was done by inserting a new copy instruction $a_3 = a_2$, followed by renaming uses of a_2 past the new copy. Figure 5.3 (b) shows the program after spilling and register assignment, and Figure 5.3 (c) shows the program after spill-free SSA elimination. Notice how, in Figure 5.3 (b), CSSA makes a difference by requiring the extra instruction that copies from a_2 to a_3. We now do register allocation and assign each of a, a_1, and a_3 the same memory location m because those variables do not interfere. In Figure 5.3 (b), the value of a_2 arrives in memory location m_2, and is then copied to memory location m

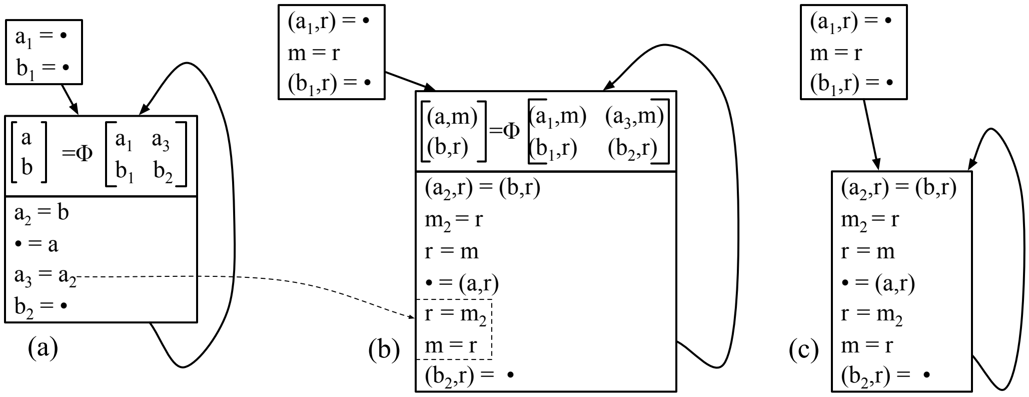


Figure 5.3: SSA-based register allocation and spill-free SSA elimination.

via the register r. The point of the copy is to let both elements of the first row of the ϕ-matrix to be represented in m, just like both elements of the second row of the ϕ-matrix are represented in r. We finally arrive at Figure 5.3 (c) without any further spills.

5.3 Our SSA Elimination Framework

We now show that for programs in CSSA-form, the problem of replacing each ϕ-function with copy and swap instructions is significantly simpler than for programs in SSA-form (Theorem 10). Along the way, we will define all the concepts and notations that we use. We use the matrix notation from Section 2.3 to represent ϕ-functions.

Conventional Static Single Assignment Form The CSSA representation was first described by Sreedhar et al. [84] who used it to facilitate register coalescing. In order to define CSSA-form, we first define an equivalence relation $\equiv$ over the set of variables used in a program. We define $\equiv$ to be the smallest equivalence relation such that for every set of ϕ-functions $V = \phi M$, where V is a vector of length n with entries v_i, and M is an $n \times m$ matrix with entries v_{ij}, we have

$$\text{for each } i \in 1..n : v_i \equiv v_{i1} \equiv v_{i2} \equiv \ldots \equiv v_{im}.$$

Sreedhar et al. use *ϕ-congruence classes* to denote the equivalence classes of $\equiv$.

Definition 9 *A program is in CSSA-form if and only if for every pair of variables v_1, v_2 that occur in the same ϕ-function, we have that if $v_1 \equiv v_2$, then v_1 and v_2 do not interfere.*

The program in Figure 5.4 (a) is not in CSSA-form, because the ϕ-related variables a and a_2 interfere; however, the programs in Figures 5.4 (b) and (c) are. A SSA-form program can be converted to CSSA-form via a very simple algorithm, called the “Method I” or “naive algorithm” by Sreedhar *et al.* [84, p.199]. This algorithm splits live ranges of variables used or defined in ϕ-functions. Sreedhar *et al.* have shown that this live range splitting is sufficient to convert a SSA-form program to a program in conventional-SSA-form. The transformed control flow graph contains one ϕ-related equivalence class for each ϕ-function, and one equivalence class for each virtual v that does not participate in any ϕ-function. Figure 5.4 (b) shows how the equivalence class of the ϕ-function $a = \phi(a_1, a_2)$ could be converted into CSSA-form using Sreedhar’s “Method I”.

Although the naive algorithm produces correct programs, it is excessively conservative: two virtuals are ϕ-related if, and only if, they are used in the same ϕ-function. If a copy inserted by the naive method can be removed without creating interferences between ϕ-related variables, we call it *redundant.* Budimlic *et al.* [21] gave a fast algorithm to remove redundant copies. The program in Figure 5.4 (c) was produced according to this method.

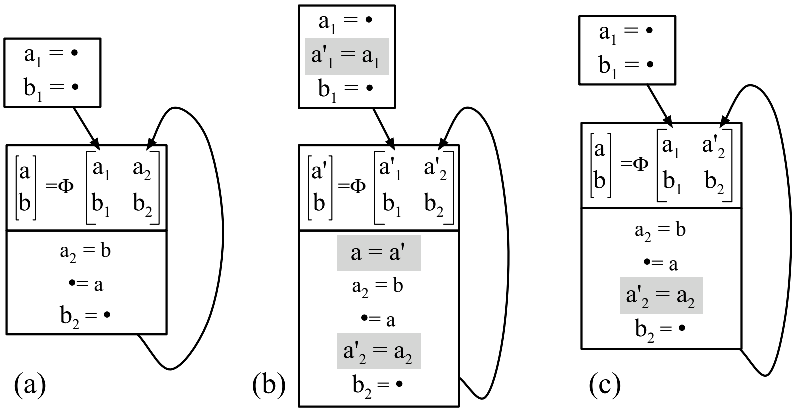


Figure 5.4: (a) Original, non-CSSA-form program. (b) Program converted into CSSA via Sreedhar's "Method I". (c) Redundant copies removed via Budimlic's fast copy coalescing.

Frugal register allocators and Spartan parallel copies A register allocator for a CSSA-form program can assign the same location to all the variables $v_i, v_{i1}, \ldots, v_{im}$, for each $i \in 1..n$, because none of those variables interfere. We say that register allocation is *frugal* if it uses at most *one* memory location together with any number of registers as locations for $v_i, v_{i1}, \ldots, v_{im}$, for each $i \in 1..n$.

The problem of doing SSA-elimination consists of implementing one parallel copy for each column in each ϕ-matrix. We can implement each parallel copy independently of the others. We will use the notation

$$(l_1, \ldots, l_n) := (l'_1, \ldots, l'_n)$$

for a single parallel copy, in which $l_i, l'_i, i \in 1..n$, range over $R \cup M$, where $R = \{r_1, r_2, \ldots, r_k\}$ is a set of registers, and $M = \{m_1, m_2, \ldots\}$ is a set of memory locations. We say that a parallel copy is *well defined* if all the locations on its left side are pairwise distinct. We will use ρ to denote a *store* that maps elements of

$R \cup M$ to values. If ρ is a store in which $l'_1, \dots, l'_n$ are defined, then the meaning of a parallel copy $(l_1, \dots, l_n) = (l'_1, \dots, l'_n)$ is $\rho[l_1 \leftarrow \rho(l'_1), \dots l_n \leftarrow \rho(l'_n)]$.

We say that a well-defined parallel copy $(l_1, \dots, l_n) = (l'_1, \dots, l'_n)$ is *spartan* if

1. for all l'_a, l'_b, if $l'_a = l'_b$, then $a = b$;

2. for all l_a, l'_b such that l_a and l'_b are memory locations, we have $l_a = l'_b$ if and only if $a = b$.

Informally, condition (1) says that the locations on the right-hand side are pairwise distinct, and condition (2) says that a memory location appears on both sides of a parallel copy if and only if it appears at the same index.

Theorem 10 *After frugal register allocation, the ϕ-functions used in a program in CSSA-form can be implemented using spartan parallel copies.*

Proof. We must show that the parallel copies that we derive from a CSSA-form program after frugal register allocation meet the two properties that define spartan parallel copies:

1. The CSSA-form is a subset of SSA-form; thus, every variable is defined at most once. This implies that all the parallel copies must be well defined.

2. Given a set of ϕ-functions $V = \phi M$, a frugal register allocator assigns the same memory slot to spilled variables in row i of M, and in index i of V. If a variable in row $j, j \neq i$ is spilled, it must be allocated to a memory spot different than the one reserved for variables in the i-th row, as variables in the same column interfere. The same is true for variables in V.

□

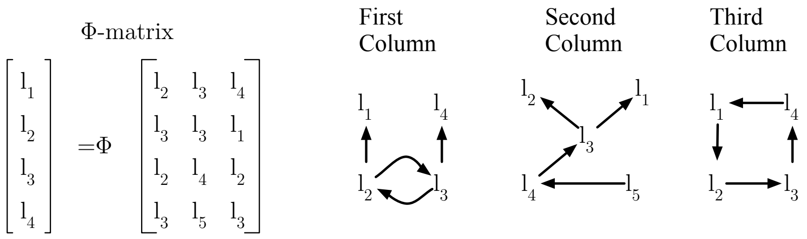


Figure 5.5: A ϕ-matrix and its representation as three location transfer graphs.

5.4 From windmills to cycles and paths

We now show that a spartan parallel copy can be represented using a particularly simple form of graph that we call a spartan graph (Theorem 12).

We will represent each parallel copy by a *location transfer graph.*

Definition 11 ***Location Transfer Graph***. *Given a well-defined parallel copy* $(l_1, \ldots, l_n) := (l'_1, \ldots, l'_n)$*, the corresponding location transfer graph* $G = (V, E)$ *is a directed graph where* $V = \{l_1, \ldots, l_n, l'_1, \ldots, l'_n\}$*, and* $E = \{(l'_a, l_a) \mid a \in 1..n\}$*.*

Figure 5.5 contains a ϕ-matrix and its representation as three location transfer graphs. The location transfer graphs that represent well-defined parallel copies form a family of graphs known as *windmills* [78]. This name is due to the shape of the graphs: each connected component has a central cycle from which sprout trees, like the blades of a windmill.

The location transfer graphs that represent spartan parallel copies form a family of graphs that is significantly smaller than windmills. We say that a location transfer graph G is *spartan* if

- the connected components of G are cycles and paths;

- if a connected component of G is a cycle, then either all its nodes are in R, or it is a self loop (m, m);
- if a connected component of G is a path, then only its first and/or last nodes can be in M; and
- if (m_1, m_2) is an edge in G, then $m_1 = m_2$.

Notice that the first and second graphs in Figure 5.5 are not spartan because they contain nodes with out-degree 2. In contrast, the third graph in Figure 5.5 is spartan (if l_1, l_2, l_3, l_4 are registers), because it is a cycle.

Theorem 12 *A spartan parallel copy has a spartan location transfer graph.*

Proof. It is straightforward to prove the following properties:

1. the in-degree of any node is at most 1;
2. the out-degree of any node is at most 1; and
3. if a node is a memory location m then:
   (a) the sum of its out-degree and in-degree is at most 1, or
   (b) G contains an edge (m, m).

The result is immediate from (1)–(3). □

5.5 SSA elimination

Our goal is to implement spartan parallel copies in the language `Seq` that contains just four types of instructions: register-to-register moves $r_1 := r_2$, loads $r := m$, stores $m := r$, and register swaps $r_1 \oplus r_2$. Notice that `Seq` does not contain instructions to swap or copy the contents of memory locations in one step. We use ι to range over instructions. A `Seq` program is a sequence I of instructions that modify a store ρ according to the following rules:

$$\frac{\langle \iota, \rho \rangle \rightarrow \rho'}{\langle \iota; I, \rho \rangle \rightarrow \langle I, \rho' \rangle}$$

$$\langle l_1 := l_2, \rho \rangle \rightarrow \rho[l_1 \leftarrow \rho(l_2)]$$

$$\langle r_1 \oplus r_2, \rho \rangle \rightarrow \rho[r_1 \leftarrow \rho(r_2), r_2 \leftarrow \rho(r_1)]$$

The problem of implementing a parallel copy can now be stated as follows.

Implementation of a Spartan Parallel Copy

Instance: a spartan parallel copy $(l_1, \ldots, l_n) = (l'_1, \ldots, l'_n)$.

Problem: find a `Seq` program I such that for all stores ρ,

$$\langle I, \rho \rangle \rightarrow^* \rho[l_1 \leftarrow \rho(l'_1), \ldots l_n \leftarrow \rho(l'_n)].$$

Our algorithm **ImplementSpartan** uses a subroutine **ImplementComponent** that works on each connected component of a spartan location transfer graph and is entirely standard.

Algorithm 1 – ImplementComponent: Input: G, Output: I

```
Require: $G$ is a cycle or a path
Ensure: $I$ is a Seq program.
if $G$ is a path $(l_1, r_2), \ldots, (r_{n-2}, r_{n-1}), (r_{n-1}, l_n)$ then
    $I = (l_n := r_{n-1}; r_{n-1} := r_{n-2}; \ldots; r_2 := l_1)$
else if $G$ is a cycle $(r_1, r_2), \ldots, (r_{n-1}, r_n), (r_n, r_1)$ then
    $I = (r_n \oplus r_{n-1}; r_{n-1} \oplus r_{n-2}; \ldots; r_2 \oplus r_1)$
end if
```

Algorithm 2 – **ImplementSpartan**: Input: G, Output: program I

```
Require: $G$ is a spartan location transfer graph.
Require: $G$ has connected components $C_1, \ldots, C_m$.
Ensure: $I$ is a Seq program.
$I = \textbf{ImplementComponent}(C_1); \ldots ; \textbf{ImplementComponent}(C_m)$;
```

Theorem 13 (Correctness) *For a spartan location transfer graph G, **ImplementSpartan**(G) is a correct implementation of G.*

Proof. See Appendix A.4. □

Once we have implemented each spartan parallel copy, all that remains to complete spill-free SSA elimination is to replace the ϕ-functions with the generated code. As illustrated in Figure 5.3, the generated code for a parallel copy must be inserted at the end of the basic block that leads to the parallel copy.

5.5.1 SSA Elimination and Critical Edges

Critical edges are edges that connect a basic block with multiple successors to a basic block with multiple predecessors. Briggs *et al.* [14] have shown that the existence of critical edges in the source program may lead to the production of incorrect code during the replacement of ϕ-functions by copy instructions. As demonstrated by Sreedhar et al [84], the CSSA-form allows to handle the problems pointed by Briggs *et al.* without requiring the elimination of critical edges from the source program. Nonetheless, the absence of critical edges greatly simplifies SSA elimination after register allocation. For instance, if the code produced by **ImplementComponent** is always inserted at the end of basic blocks, then it can produce wrong code, as the example in Figure 5.6 shows. In this example, the elimination of the ϕ-function $(a_2, r_2) = \phi[(a_1, r_1), \ldots]$ requires the contents of register r_1 to be moved into register r_2 in the control-flow path connecting blocks 1 and 4. However, such transfer cannot be inserted at the end of block 1, or it would overwrite the value of b, nor at the beginning of block 4, or it would overwrite the value of a_3.

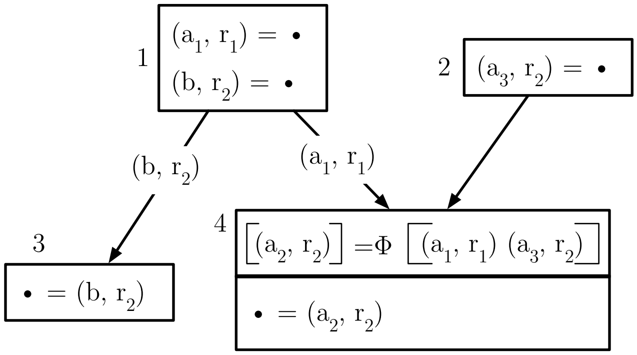


Figure 5.6: The presence of critical edges leads to incorrect code.

Another problem of critical edges is that they may cause an increase in the global register pressure of the source program, even if ϕ-functions are eliminated before register allocation. The register pressure at some program point, as defined in Section 2.3.1, is the minimal number of registers necessary to allocate all the variables alive at that point. The global register pressure of a program is the maximum number of registers necessary to allocate all the variables in the program. For example, the program in Figure 5.7 (a) illustrates the *swap-problem*, pointed by Briggs *et al.* [14], and Figure 5.7 (b) shows the same program, converted into CSSA-form. The interference graph of the latter program has chromatic number 3, whereas the graph of the former program has chromatic number 2. Figure 5.7 (c) shows the same program, after the critical edge forming the loop has been removed. The interference graph of this program has chromatic number 2. If the source SSA-form program has no critical edges, then its register pressure is guaranteed to remain the same after the convertion into CSSA-form, as we show in Theorem 14.

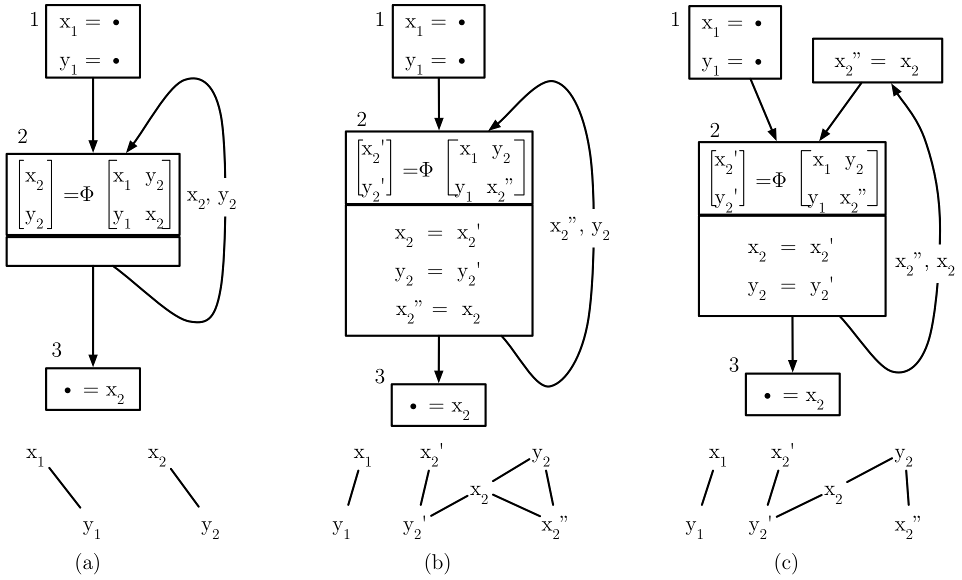


Figure 5.7: (a) Example program. (b) Program in CSSA-form. (c) Program after critical edge is eliminated. The interference graph of each program is shown below its control flow graph.

Theorem 14 ***(Register Pressure)*** *Let P be a program whose control flow graph does not contain critical edges. It is possible to convert P to CSSA-form without increasing its register pressure.*

Proof. See Appendix A.5. □

5.6 Optimizations

We will present three optimizations of the **ImplementSpartan** algorithm. Each optimization (1) has little impact on compilation time, (2) has a significant positive impact on the quality of the generated code, (3) can be implemented as constant-time checks, and (4) must be accompanied by a small change to the register allocator.

5.6.1 Store hoisting

Each variable name is defined only once in an SSA-form program; therefore, the register allocator needs to insert only one store instruction per spilled variable. However, algorithm **ImplementSpartan** inserts a store instruction for each edge (r, m) in the location transfer graph. We can change **ImplementComponent** to avoid inserting store instructions:

```
if $G$ is a path $(l_1, r_2), \ldots, (r_{n-2}, r_{n-1}), (r_{n-1}, m)$ then
    $I = (r_{n-1} := r_{n-2}; \ldots; r_2 := l_1)$
    $\ldots$
end if
```

For this to work, we must change the register allocator to explicitly insert a store instruction after the definition point of each spilled variable. On the average, store hoisting removes 12% of the store instructions in SPEC CPU 2000.

5.6.2 Load Lowering

Load lowering is the dual of store hoisting: it reduces the number of load and copy instructions inserted by the **ImplementSpartan** Algorithm. There are situations when it is advantageous to reload a variable right before it is used, instead of during the elimination of ϕ-functions. Load lowering is particularly useful in algorithms that follow the bin-packing model [56, 76, 81, 87], because they allow the same variable to reach join points in different physical locations.

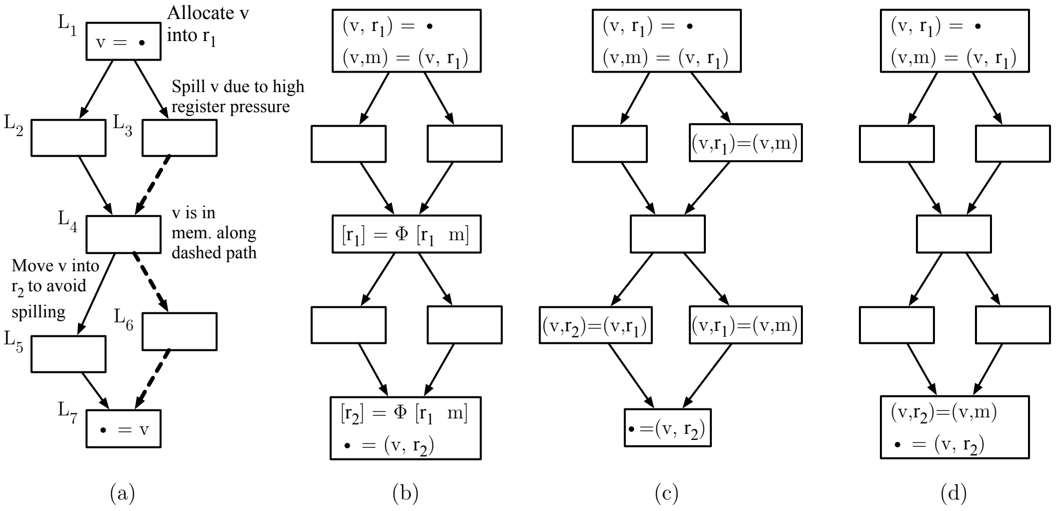


Figure 5.8: (a) Example program (b) Program augmented with mock ϕ-functions. (c) SSA elimination without load-lowering. (d) Load-lowering in action.

In Figure 5.8 we simulate the different locations of variable v by inserting mock ϕ-functions at the beginning of basic blocks L_2 and L_7, as pointed in Figure 5.8 (b). The fixing code will be naturally inserted when these ϕ-functions are eliminated. The load lowering optimization would replace the instructions used to implement the ϕ-functions, shown in Figure 5.8 (c), with a single load before the use of v at basic block L_7, as outlined in Figure 5.8 (d).

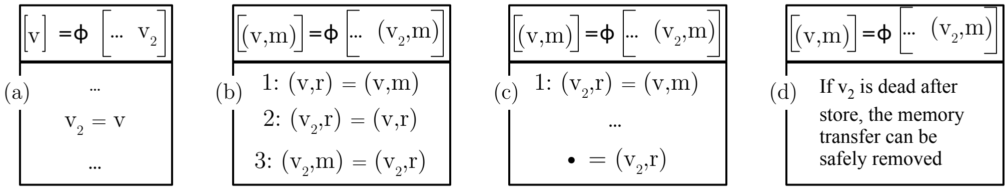
Variables can be lowered according to the nesting depth of basic blocks in loops, or the static number of instructions that could be saved. The SSA elimination algorithm must remember, for each node l in the location transfer graph, which variable is allocated into l. During register allocation we mark all the variables v that would benefit from lowering, and we avoid inserting loads for locations that have been allocated to v. Instead, the register allocator must insert reloads before each use of v. These reloads may produce redundant memory transfers, which are eliminated by the memory coalescing pass described in Section 5.6.3.

The updated elimination algorithm is outlined below:

```
if  $G$ is a path $(m, r_2), \ldots, (r_{n-2}, r_{n-1}), (r_{n-1}, l_n)$  then
    if $m$ is holding a variable marked to be lowered then
        $I = (l_n := r_{n-1}; r_{n-1} := r_{n-2}; \ldots ; r_3 := r_2)$
    else
        $I = (l_n := r_{n-1}; r_{n-1} := r_{n-2}; \ldots ; r_2 := m)$
    end if
    ...
end if
```

5.6.3 Memory coalescing

A memory transfer is a sequence of instructions that copies a value from a memory location m_1 to another memory location m_2. The transfer is redundant if these locations are the same. The CSSA-form allows us to coalesce a common occurrence of redundant memory transfers. Consider, for instance, the code that the compiler would have to produce in case variables v_2 and v, in the figure below, are spilled. In order to send the value of v_2 to memory, the value of v would have to be loaded into a spare register r, and then the contents of r would have to be stored, as illustrated in figure (b). However, v and v_2 are mapped to the same memory location because they are ϕ-related. The store instruction can always be eliminated, as in figure (c). Furthermore, if the variable that is the target of the copy - v_2 in our example - is dead past the store instruction, then the whole memory transfer can be completely eliminated, as we show in figure (d) below:



5.7 Experimental results

We rely on the SSA elimination framework described in this chapter to implement the ϕ-functions, π-functions and parallel copies that we use to produce the elementary form from Chapter 3. The experiments shown in this section were performed in the same execution environment described in Section 3.6.

Impact of our SSA Elimination Method Figure 5.9 summarizes static data obtained from the compilation of SPEC CPU 2000. Our SSA Elimination algorithm had to implement 197,568 location transfer graphs when compiling this benchmark suite. These LTGs contain 1,601,110 edges, out of which 855,414, or 53% are memory transfers. Due to properties of spartan location transfer graphs, edges representing memory transfers are self-loops, that is, an edge from a node m pointing to itself. Because our memory transfer edges have source and target pointing to the same address, the SSA Elimination algorithm does not have to insert any instruction to implement them. Potential spills could have happened in 11,802 location transfer graphs, or 6% of the total number of graphs, implying that, if we had used a spilling on demand approach instead of our SSA elimination framework, a second spilling phase would be necessary in all the benchmark programs. We mark as potential spills the location transfer graphs that contain memory transfers, and in which the register pressure is maximum, that is, all the physical registers are used in the right side of the parallel copy.

	gcc	pbk	gap	msa	vtx	twf	cfg	vpr	amp	prs	gzp	bz2	art
#ltg	72.6	40.3	22.1	15.6	15.8	6.8	7.7	4.5	4.0	5.2	.9	.73	.36
%sp	3.3	5.0	9.8	2.3	9.3	6.5	14.9	13.5	7.9	6.5	10.9	22.7	9.2
#edg	586.2	256.3	150.8	96.9	121.5	58.0	124.2	101.7	29.6	35.5	11.1	14.3	2.7
%mt	56.4	41.7	43.5	50.6	47.1	57.3	66.8	75.4	37.4	42.8	63.6	71.8	46.0

Figure 5.9: #ltg: number of location transfer graphs (in thousands), %sp: percentage of LTG's that are potential spills, #edg: number of edges in all the LTG's (in thousands), %mt: percentage of the edges that are memory transfers.

Time Overhead of SSA-Elimination The charts in Figure 5.10 show the time required by our compilation passes. Register allocation accounts for 28% of the total compilation time. This time is similar to the time required by the standard linear scan register allocator, as reported in previous works [77, 80]. The passes related to SSA elimination account for about 4.8% of the total compilation time. These passes are: (i) Sreedhar's "Method I", which splits the live ranges of all the variables that are part of ϕ-functions [84, pg.199]; (ii) a pass to remove critical edges; (iii) a pass to remove redundant copies, based on Budimlic's fast copy coalescing [21]; (iv) our spill-free SSA elimination pass. The amount of time taken by each of these passes is distributed as follows: (i) 0.2%, (ii) 0.5%, (iii) 1.6% and(iv) 2.5%.

Impact of the Optimizations Figure 5.11 shows the static reduction of load, store and copy instructions due to the optimizations described in Section 5.6. The criterion used to determine if a variable should be lowered or not is the number of reloads that would be inserted for that variable versus the number of uses of the variable. Before running the SSA-elimination algorithm we count the number of reloads that would be inserted for each variable. The time taken to get this measure is negligible compared to the time to perform SSA-elimination: loads

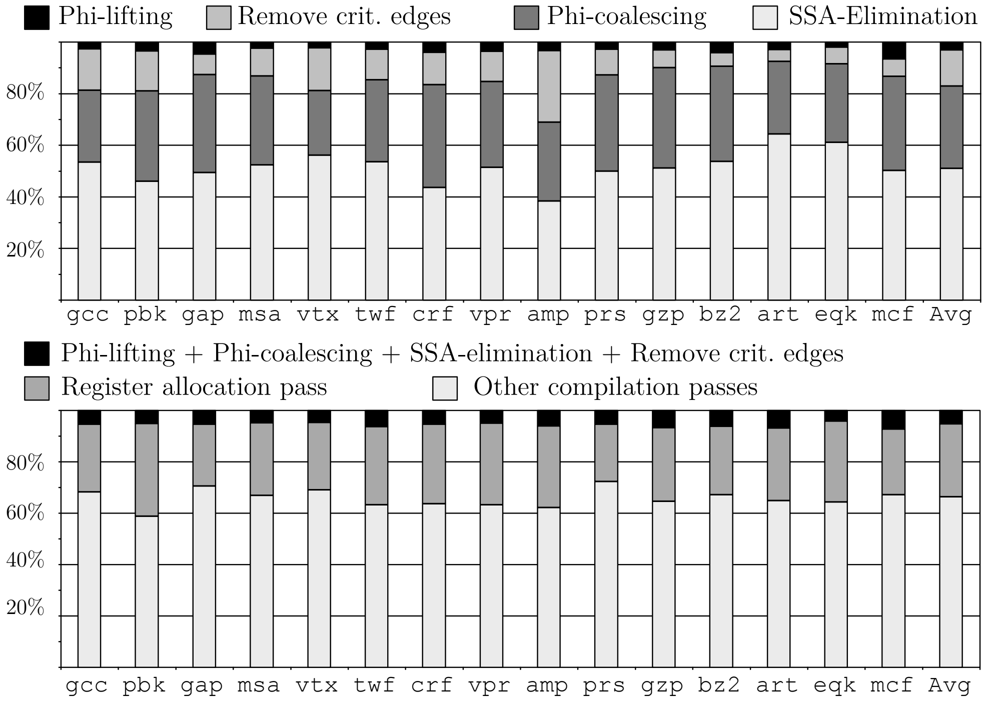


Figure 5.10: Execution time of different compilation passes.

can only be the last edge of a spartan location transfer graph (Theorem 12). A variable is lowered if its spilling causes the allocator to insert more reloads than the number of uses of that variable in the source program. Store hoisting (SH) alone eliminates on average about 12% of the total number of stores in the target program, which represents slightly less than 5% of the lines of spill code inserted. By plugging in the elimination of redundant memory transfers (RMTE) we remove other 2.6% lines of spill code. Finally, load lowering (LL), on top of these other two optimizations, eliminates 7.8% more lines of spill code. Load lowering also removes 5% of the copy instructions from the target programs.

The chart in the bottom part of Figure 5.11 shows how the optimizations influence the run time of the benchmarks. On the average, they produce a speed

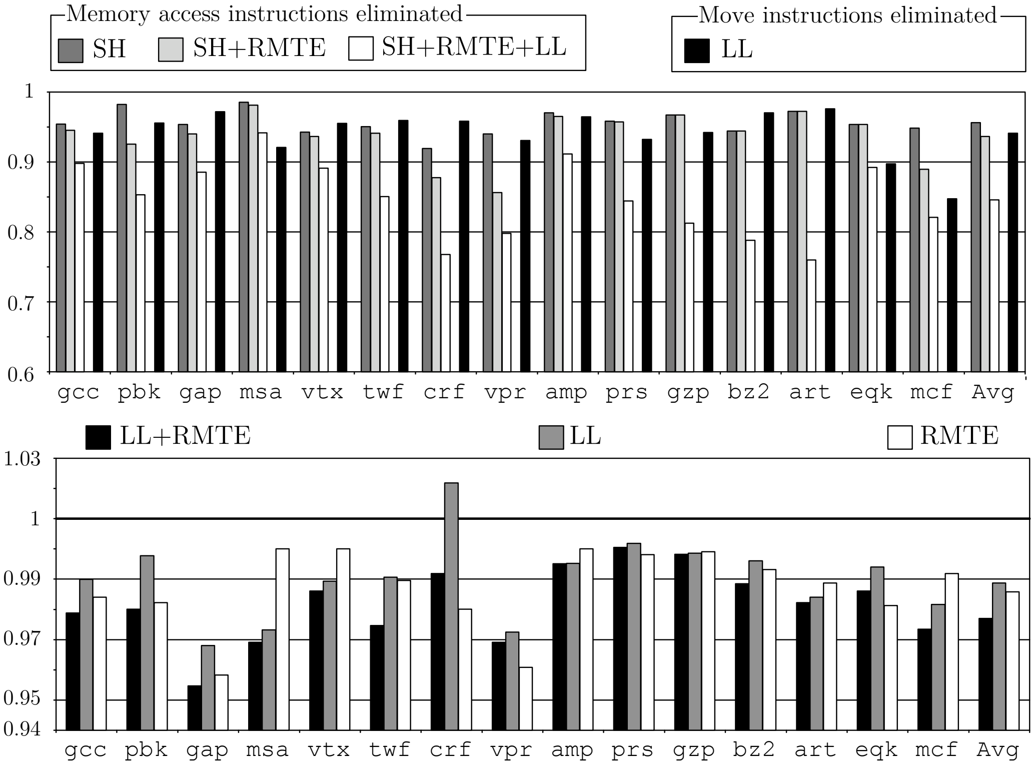


Figure 5.11: Impact of Load Lowering (LL) and Redundant Memory Transfer Elimination (RMTE) on the code produced after SSA-elimination. (Up) Code size. (Down) Run-time.

up of 1.9%. Not all the programs benefit from load lowering. For instance, load lowering increases the run time of `186.crafty` in almost 2.5%. This happens because, for the sake of simplicity, we do not take into consideration the loop nesting depth of basic blocks when lowering loads. We speculate that more sophisticated criteria would produce more substantial performance gains. Yet, these optimizations are being applied on top of a very efficient register allocator, and they do not incur in any measurable penalty in terms of compilation time.

5.8 Final Remarks

This chapter has presented spill-free SSA elimination, a simple and efficient algorithm for SSA elimination after register allocation that avoids increasing the number of spilled variables. Our algorithm runs in polynomial time and accounts for a small portion of the total compilation time.

Our approach relies on the ability to swap the contents of two registers. For integer registers, architectures such as x86 provide a swap instruction, while on other architectures swaps can be implemented with a sequence of three `xor` instructions. In contrast, for floating point registers, most architectures provide neither a swap instruction nor a `xor` instruction, so instead compiler writers have to use one of the other approaches to SSA-elimination, e.g: separate a temporary register or perform spilling on demand.

CHAPTER 6

Conclusion

In this dissertation we have shown that the spill free register allocation problem has polynomial time solution even in architectures as irregular and constrained as the x86. The key insight of our work is to show how the elementary program representation simplifies register allocation. Elementary graphs are the interference graphs of programs in this representation. We have shown that a large array of problems related to graph coloring has polynomial time solution when restricted to elementary graphs. This result is particularly surprising when we consider that most of these problems are already NP-complete for interval graphs, one of the simplest classes of intersection graphs.

6.1 Summary of Results

Along this research we have made a number of discoveries, which we summarize in this section.

Proofs of NP-completeness We have shown that a number of register allocation related problems is NP-complete:

- Register allocation after classic SSA-elimination is NP-complete [74]. This result holds if SSA elimination is performed by algorithms that rely only on move instructions to implement ϕ-functions. The algorithms discussed in Section 5 are able to do SSA Elimination without raising the demand for registers in the target program.

- Finding minimum number of instructions to convert a program from SSA to CSSA-form is NP-complete [72].

- Alias register allocation is NP-complete for chordal graphs. This proof was later improved by Lee *et al.* [59] who showed that alias register allocation is NP-complete even for interval graphs, a subset of chordal graphs.

- Spilling is NP-complete in elementary programs, as we show in Section A.3. This proof includes showing that any chordal graph is the interference graph of some SSA-form program.

Polynomial Time Algorithms We have shown that many problems that are NP-complete for general graphs have polynomial time solution when restricted to elementary graphs:

- Graph-coloring, 1-2 Aligned Coloring, Coloring extension and the 1-2 Aligned coloring extension for type-1 puzzles, as we show in Section A.1.

- We have been able to show that the *Max-Coloring Problem*, which is known to be NP-complete for interval graphs [70], has polynomial time solution for elementary graphs. This implies that the problem of determining the size of the smallest register bank, in bits, for a given application has polynomial time solution [71].

- Recently Jens Palsberg and Siddharth Tiwary showed that the aligned 1-2 graph coloring problem has polynomial time solution for type-2 puzzles. Their proof demonstrates that spill-free register allocation has polynomial time solution for all the most popular computer architectures currently in use, including x86, PowerPC, ARM, and Ultra-Sparc.

- The minimum number of sequential operations necessary to perform SSA elimination in a colored CSSA-form program. This result is valid for register files without aliasing. The analogous question referring to aliased register banks is an open problem.

Software Implementation Apart from the theoretical results, we have implemented a vast amount of software, currently available under the license of the Regents of the University of California, Los Angeles:

- a register allocator based on the coloring of chordal graphs [73]. This algorithm is implemented in the JoeQ compilation framework [89];
  `http://compilers.cs.ucla.edu/fernando/projects/APLAS2005/`

- a SSA-based register allocator implemented on PowerPC. This algorithm follows the description given by Hack *et al.* [48];
  `http://compilers.cs.ucla.edu/fernando/projects/soc/`

- a translator validator for register allocation. This tool is based on the type-checking algorithm described by Nandivada *et al.* [66].
  `http://compilers.cs.ucla.edu/fernando/projects/debugger/`

- the puzzle-based register allocator described in Chapter 3.
  `http://compilers.cs.ucla.edu/fernando/projects/puzzles/`

6.2 Limitations of our approach

Register assignment is not the single most important part of register allocation. We can successfully determine if all the variables in a program can be completely packed into registers, but, when that is not the case, we have to face the difficult task of deciding which variables must stay in registers, and which variables must be stored in memory. Minimizing the cost of memory access is a NP-complete problem, even in program representations as restricted as the elementary form.

Nevertheless, we have been able to show that the coupling of our register assignment algorithm and Belady's heuristics for spilling is powerful enough to compete with much slower algorithms, such as iterated register coalescing and partitioned boolean quadratic programming.

We have shown that local coalescing has polynomial time solution for type-1 puzzles that have an empty board. That should be enough, as 95% of the puzzles found in real benchmarks have empty board. The global coalescing problem is NP-complete. This is a simple reduction from the Aliased coloring problem.

Global coalescing, in contrast to the local coalescing problem discussed in Chapter 4, is NP-complete, as proved by Ferriére *et al.*.

Optimizing the translation out-of-ssa with renaming constraints

6.3 Open Problems

Finding the minimum number of copies to implement SSA Elimination with aliasing. Optimal coalescing for puzzles with pre-colored areas, and for type-n, $n > 1$.

APPENDIX A

Proofs

A.1 Proof of Theorem 1

We will prove Theorem 1 for register banks that give type-1 puzzles. Theorem 1 states:

> ***(Equivalence)*** *Spill-free register allocation with pre-coloring for an elementary program is equivalent to solving a collection of puzzles.*

In Section A.1.1 we define three key concepts that we use in the proof, namely aligned 1-2-coloring extension, clique substitution of P_3, and elementary graph. In Section A.1.2 we state four key lemmas and show that they imply Theorem 1. Finally, in four separate subsections, we prove the four lemmas.

A.1.1 Definitions

We first state again a graph-coloring problem that we mentioned in Section 3.7, namely aligned 1-2-coloring extension.

> Aligned 1-2-coloring extension
> **Instance**: a number of colors $2K$, a weighted graph G, and a partial aligned 1-2-coloring ϕ of G. **Problem**: Extend ϕ to an aligned 1-2-coloring of G.

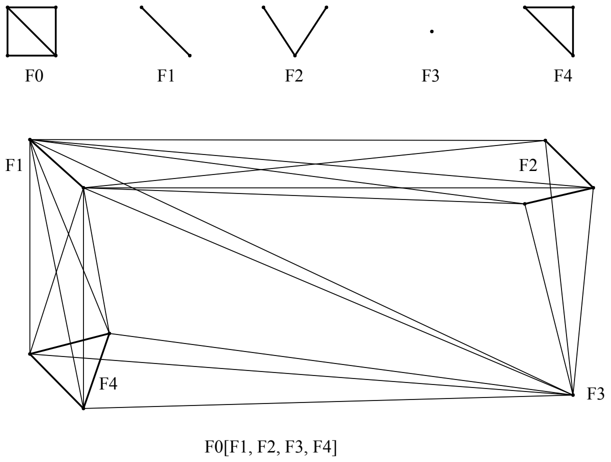


Figure A.1: Example of a composition graph (taken from [42]).

We use the notation $(2K, G, \phi)$ to denote an instance of the aligned 1-2-coloring extension problem. For a vertex v of G, if v is in the domain of ϕ, then we say that v is *pre-colored*.

Next we define the notion of a *clique substitution of* P_3. Let H_0 be a graph with n vertices $v_1, v_2, \ldots, v_n$ and let $H_1, H_2, \ldots, H_n$ be n disjoint graphs. The *composition graph* [42] $H = H_0[H_1, H_2, \ldots, H_n]$ is formed as follows: for all $1 \leq i, j \leq n$, replace vertex v_i in H_0 with the graph H_i and make each vertex of H_i adjacent to each vertex of H_j whenever v_i is adjacent to v_j in H_0. Figure A.1 shows an example of composition graph.

P_3 is the path with three nodes, e.g., $(\{x, y, z\}, \{xy, yz\})$. We define a clique substitution of P_3 as $P_{X,Y,Z} = P_3[K_X, K_Y, K_Z]$, where each K_S is a complete graph with $|S|$ nodes.

Definition 15 *A graph G is an elementary graph if and only if every connected component of G is a clique substitution of P_3.*

A.1.2 Structure of the Proof

We will prove the following four lemmas.

- Lemma 16: Spill-free register allocation with pre-coloring for an elementary program P is equivalent to the aligned 1-2-coloring extension problem for the interference graph of P.
- Lemma 23: An elementary program has an elementary interference graph.
- Lemma 25: An elementary graph is the interference graph of an elementary program.
- Lemma 27: Aligned 1-2-coloring extension for a clique substitution of P_3 is equivalent to puzzle solving.

We can now prove Theorem 1:

Proof. From Lemmas 16, 23, and 25 we have that spill-free register allocation with pre-coloring for an elementary program is equivalent to aligned 1-2-coloring extension for elementary graphs. From Lemma 27 we have that aligned 1-2-coloring extension for elementary graphs is equivalent to solving a collection of puzzles. □

A.1.3 From register allocation to coloring

Lemma 16 *Spill-free register allocation with pre-coloring for an elementary program P is equivalent to the aligned 1-2-coloring extension problem for the interference graph of P.*

Proof. Chaitin *et al.* [24] have shown that spill-free register allocation for a program P is equivalent to coloring the interference graph of P, where each color represents one physical register. To extend the spill-free register allocation to an architecture with a type-1 register bank, we assign weights to each variable in the interference graph, so that variables that fit in one register are assigned weight 1, and variables that fit in a register-pair are assigned weight 2. To include pre-coloring, we define $\phi(v) = r$, if vertex v represents a pre-colored variable, and color r represents the register assigned to this variable. Otherwise, we let $\phi(v)$ be undefined. □

A.1.4 Elementary programs and graphs

We will show in three steps that an elementary program has an elementary interference graph. We first give a characterization of clique substitutions of P_3 (Lemma 18). Then we show that a graph G is an elementary graph if and only if G has an *elementary interval representation* (Lemma 20). Finally we show that the interference graph of an elementary program has an elementary interval representation and therefore is an elementary graph (Lemma 23).

A.1.4.1 A Characterization of Clique Substitutions of P_3

We will give a characterization of a clique substitution of P_3 in terms of forbidden induced subgraphs. Given a graph $G = (V, E)$, we say that $H = (V', E')$ is an induced subgraph of G if $V' \subseteq V$ and, given two vertices v and u in V', $uv \in E'$ if, and only if, $uv \in E$. Given a graph F, we say that G is F-free if none of its induced subgraphs is isomorphic to F. In this case we say that F is a forbidden subgraph of G. Some classes of graphs can be characterized in terms of forbidden subgraphs, that is, a set of graphs that cannot be induced in any of the graphs in that class. In this section we show that any graph $P_{X,Y,Z}$ has three forbidden subgraphs: (i) P_4, the simple path with four nodes; (ii) C_4, the cycle with four nodes, and (iii) $3K_1$, the graph formed by three unconnected nodes. These graphs are illustrated in Figure A.2, along with the bipartite graph $K_{3,1}$, known as *the claw*. The claw is important because it is used to characterize many classes of graphs. For example, the interval graphs that do not contain any induced copy of the claw constitute the class of the unit interval graphs [42, p. 187]. A key step of our proof of Lemma 20 shows that elementary graphs are claw-free.

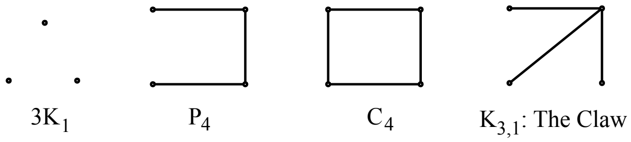


Figure A.2: Some special graphs.

We start our characterization by describing the class of the *Trivially Perfect Graphs* [41]. In a trivially perfect graph, the size of the maximal independent set equals the size of the number of maximal cliques.

Theorem 17 ***(Golumbic [41])*** *A graph G is trivially perfect if and only if G contains no induced subgraph isomorphic to C_4 or P_4.*

The next lemma characterizes $P_{X,Y,Z}$ in terms of forbidden subgraphs.

Lemma 18 *A graph G is a clique substitution of P_3 if and only if G contains no induced subgraph isomorphic to C_4, P_4, or $3K_1$.*

Proof. (⇒) Let G be a clique substitution of P_3, and let G be of the form $P_{X,Y,Z}$. Let us first show that G is trivially perfect. Note that G contains either one or two maximal cliques. If G contains one maximal clique, we have that G is of the form $P_{\emptyset,Y,\emptyset}$, and the maximal independent set has size 1. If G contains two maximal cliques, those cliques must be $X \cup Y$ and $X \cup Z$. In this case, the maximal independent set has two vertices, namely an element of $X - Y$ and an element of $Z - Y$. So, G is trivially perfect, hence, by Theorem 17, G does not contain either C_4 nor P_4 as induced subgraphs. Moreover, the maximum independent set of G has size one or two; therefore, G cannot contain an induced $3K_1$.

($\Leftarrow$) If G is C_4-free and P_4-free, then G is trivially perfect, by Theorem 17. Because G is $3K_1$-free, its maximal independent set has either one or two nodes. If G is unconnected, we have that G consists of two unconnected cliques; thus, $G = P_{X,\emptyset,Y}$. If G is connected, it can have either one or two maximal cliques. In the first case, we have that $G = P_{\emptyset,Y,\emptyset}$. In the second, let these maximal cliques be C_1 and C_2. We have that $G = P_{C_1-C_2,C_1\cap C_2,C_2-C_1}$. □

A.1.4.2 A Characterization of Elementary Graphs

We recall the definitions of an *intersection* graph and an *interval* graph [42, p.9]. Let Sl be a family of nonempty sets. The intersection graph of Sl is obtained by representing each set in Sl by a vertex and connecting two vertices by an edge if and only if their corresponding sets intersect. An *interval* graph is an intersection graph of a family of subintervals of an interval of the real numbers.

A *rooted tree* is a directed tree with exactly one node of in-degree zero; this node is called *root.* Notice that there is a path from the root to any other vertex of a rooted tree. The intersection graph of a family of directed vertex paths in a rooted tree is called *a rooted directed vertex path graph*, or *RDV* [63]. A polynomial time algorithm for recognizing RDV graphs was described in [39].

The family of RDV graphs includes the interval graphs, and is included in the class of chordal graphs. An example of RDV graph is given in Figure A.3.

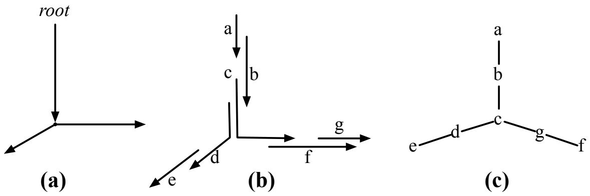


Figure A.3: (a) Directed tree T. (b) Paths on T. (c) Corresponding RDV graph.

Following the notation in [39], we let $L = \{\overline{v_1}, \ldots, \overline{v_n}\}$ denote a set of n directed paths in a rooted tree T. The RDV graph that corresponds to L is $G = (\{v_1, \ldots, v_n\}, E)$, where $v_i v_j \in E$ if and only if $\overline{v_i} \cap \overline{v_j} \neq \emptyset$. We call L the *path representation* of G. Because T is a rooted tree, each interval $\overline{v}$ has a well-defined start point $s(\overline{v})$, and a well-defined end point $e(\overline{v})$: $s(\overline{v})$ is the point of $\overline{v}$ closest to the root of T, and $e(\overline{v})$ is the point of $\overline{v}$ farthest from the root.

Given a connected graph $G = (V, E)$, the *distance* between two vertices $\{u, v\} \subseteq V$ is the number of edges in the shortest path connecting u to v. The *diameter* of G is the maximal distance between any two pairs of vertices of G. A key step in the proof of Lemma 20 below (Claim 3) shows that the diameter of any connected component of an elementary graph is at most 2.

We define *elementary interval representation* as follows:

Definition 19 *A graph G has an* elementary interval representation *if:*

1. G is a RDV graph.

2. If $uv \in E$, then $s(\overline{u}) = s(\overline{v})$, or $e(\overline{u}) = e(\overline{v})$.

3. If $uv \in E$, then $\overline{u} \subseteq \overline{v}$ or $\overline{v} \subseteq \overline{u}$.

Lemma 20 shows that any elementary graph has an elementary interval representation.

Lemma 20 *A graph G is an elementary graph if, and only if, G has an elementary interval representation.*

Proof. ($\Leftarrow$) We first prove six properties of G:

- Claim 1: If $a, b, c \in V$, $ab \in E$, $bc \in E$ and $ac \notin E$, then we have $(\overline{a} \cup \overline{c}) \subseteq \overline{b}$ in any path representation of G.
- Claim 2: G is P_4-free.
- Claim 3: Let $C = (V_C, E_C)$ be a connected component of G. Given $a, b \in V_C$ such that $ab \notin E_C$, then $\exists v$ such that $av \in E_C$ and $bv \in E_C$.
- Claim 4: G is claw-free.
- Claim 5: Every connected component of G is $3K_1$-free.
- Claim 6: G is C_4-free.

Proof of Claim 1. Let us first show that $\overline{b} \nsubseteq \overline{a}$. If $\overline{b} \subseteq \overline{a}$, then, from $ac \notin E$ we would have $bc \notin E$, which is a contradiction. Given that $ab \in E$ and $\overline{b} \nsubseteq \overline{a}$ we have that $\overline{a} \subseteq \overline{b}$. By symmetry, we have that $\overline{c} \subseteq \overline{b}$. We conclude that $(\overline{a} \cup \overline{c}) \subseteq \overline{b}$.

Proof of Claim 2. Assume G contains four vertices x, y, z and w that induce the path $\{xy, yz, zw\}$ in G. From Claim 1 we have $(\overline{x} \cup \overline{z}) \subseteq \overline{y}$; in particular, $\overline{z} \subseteq \overline{y}$. Similarly we have $(\overline{y} \cup \overline{w}) \subseteq \overline{z}$; in particular, $\overline{y} \subseteq \overline{z}$. So, $\overline{y} = \overline{z}$. From $zw \in E$ and $\overline{y} = \overline{z}$, we have $yw \in E$, contradicting that the set $\{x, y, z, w\}$ induces a path in G.

Proof of Claim 3. From Claim 2 we have that G is P_4-free, so any minimal-length path between two connected vertices contains either one or two edges. We have $a, b \in V_C$ so a, b are connected, and we have $ab \notin E_C$, so we must have a minimal-length path $\{av, vb\}$ for some vertex v.

Proof of Claim 4. Let L be G's directed path representation. Suppose G contains four vertices x, y, z, w that induce the claw $\{xy, xz, xw\}$. Without loss of generality, we assume $s(\overline{x}) = s(\overline{y})$. Because G is an RDV-graph, we must have $e(\overline{x}) = e(\overline{z})$. However, x and w interfere, yet, w cannot share the starting point with x, or it would interfere with y, nor can w share its end point with x, or it would interfere with z. So, the claw is impossible.

Proof of Claim 5. Let $C = (V_C, E_C)$ be a connected component of G. Assume, for the sake of contradiction, that there are three vertices $\{a, b, c\} \in V_C$ such that $ab \notin E_C$, $ac \notin E_C$ and $bc \notin E_C$. From Claim 3 we have that there exists a vertex v_{ab} that is adjacent to a and b. Likewise, we know that there exists a vertex v_{bc} that is adjacent to b and c. From Claim 1 we have that in any path representation of G, $(\overline{a} \cup \overline{b}) \subseteq \overline{v_{ab}}$. We also know that $(\overline{b} \cup \overline{c}) \subseteq \overline{v_{bc}}$. Therefore, $\overline{b} \subseteq (\overline{v_{ab}} \cap \overline{v_{bc}})$, so $v_{ab}v_{bc} \in E_C$, hence either $\overline{v_{ab}} \subseteq \overline{v_{bc}}$ or $\overline{v_{bc}} \subseteq \overline{v_{ab}}$. If the first case holds, $\{a, b, c, v_{bc}\}$ induces a claw in G, which is impossible, given Claim 4. In the second case, $\{a, b, c, v_{ab}\}$ induces a claw.

Proof of Claim 6. By definition, RDV graphs are chordal graphs, which are C_4 free.

Finally, we prove that every connected component of G is a clique substitution of P_3. By Lemma 18, a minimal characterization of clique substitutions of P_3 in terms of forbidden subgraphs consists of C_4, P_4, and $3K_1$. G is C_4-free, from Claim 6, and G is P_4-free, from Claim 2. Any connected component of G is $3K_1$-free, from Claim 5.

($\Rightarrow$) Let G be a graph with K connected components, each of which is a clique substitution of P_3. Let $P_{X,Y,Z}$ be one of G's connected components. We first prove that $P_{X,Y,Z}$ has an elementary interval representation. Let T be a rooted tree isomorphic to $P_4 = (\{a, b, c, d\}, \{ab, bc, cd\})$, and let a be its root. We build an elementary graph G_P, isomorphic to $P_{X,Y,Z}$ using intervals on T. We let $\overrightarrow{v_1 v_2 \ldots v_n}$ denote the directed path that starts at node v_1 and ends at node v_n. We build an elementary interval representation of $P_{X,Y,Z}$ as follows: for any $x \in X$, we let $\overline{x} = \overrightarrow{ab}$. For any $y \in Y$, we let $\overline{y} = \overrightarrow{abcd}$. And for any $z \in Z$, we let $\overline{z} = \overrightarrow{cd}$. It is straightforward to show that the interval representation meets the requirements of Definition 19.

Let us then show that G has an elementary interval representation. For each connected component $C_i, 1 \leq i \leq K$ of G, let T_i be the rooted tree that underlies its directed path representation, and let $root_i$ be its root. Build a rooted tree T as $root \cup T_i, 1 \leq i \leq K$, where $root$ is a new node not in any T_i, and let $root$ be adjacent to each $root_i \in T_i$. The directed paths on each branch of T meet the requirements in Lemma 20 and thus constitute an elementary interval representation. □

Lemma 20 has a straightforward corollary that justifies one of the inclusions in Figure 3.20.

Corollary 21 *An elementary graph is a unit interval graph.*

Proof. Let us first show that a clique substitution of P_3 is a unit interval graph. Let i be an integer. Given $P_{X,Y,Z}$, we define a unit interval graph I in the following way. For any $x \in X - Y$, let $\overline{x} = [i, i+3]$; for any $y \in (Y - (X \cup Z))$, let $\overline{y} = [i+2, i+5]$; and for any $z \in Z - Y$, let $\overline{z} = [i+4, i+7]$. Those intervals represent $P_{X,Y,Z}$ and constitute a unit interval graph.

By the definition of elementary graphs we have that every connected component of G is a clique substitution of P_3. From each connected component G we can build a unit interval graph and then assemble them all into one unit interval graph that represents G. □

A.1.4.3 An elementary program has an elementary interference graph

Elementary programs were first introduced in Section 3.2.2. In that section we described how elementary programs could be obtained from ordinary programs via live range splitting and renaming of variables; we now give a formal definition of elementary programs.

Program points and live ranges have been defined in Section 3.2.2. We denote the live range of a variable v by $LR(v)$, and we let $d(v)$ be the instruction that defines v. A program P is strict [21] if every path in the control-flow graph of P from the start node to a use of a variable v passes through one of the definitions of v. A program P is *simple* if P is a strict program in SSA-form and for any variable v of P, $LR(v)$ contains at most one program point outside the basic block that contains $d(v)$. For a variable v defined in a basic block B in a simple program, we define $k(v)$ to be either the unique instruction outside B that uses B, or, if v is used only in B, the last instruction in B that uses v. Notice that because P is simple, $LR(v)$ consists of the program points on the unique path from $d(v)$ to $k(v)$. Elementary programs are defined as follows:

Definition 22 *A program produced by the grammar in Figure A.4 is in elementary form if, and only if, it has the following properties:*

1. *P_e is a simple program;*

2. *if two variables u, v of P_e interfere, then either $d(u) = d(v)$, or $k(u) = k(v)$;*

$$
\begin{array}{rcl}
P & ::= & S\ (L\ \ \phi(m,n)\ i^{*}\ \ \pi(p,q))^{*}\ E \\
L & ::= & L_{start}, L_1, L_2, \ldots, L_{end} \\
v & ::= & v_1, v_2, \ldots \\
r & ::= & \texttt{AX, AH, AL, BX,} \ldots \\
o & ::= & \bullet \\
 & | & v \\
 & | & r \\
S & ::= & L_{start} :\ \pi(p,q) \\
E & ::= & L_{end} :\ halt \\
i & ::= & o = o \\
 & | & V(n) = V(n) \\
\pi(p,q) & ::= & M(p,q) = \pi V(q) \\
\phi(n,m) & ::= & V(n) = \phi M(m,n) \\
V(n) & ::= & (o_1, \ldots, o_n) \\
M(m,n) & ::= & V_1(n) : L_1, .., V_m(n) : L_m
\end{array}
$$

Figure A.4: The grammar of elementary programs.

and

3. *if two variables u, v of P_e interfere, then either $LR(u) \subseteq LR(v)$, or $LR(v) \subseteq LR(u)$.*

We can produce an elementary program from a strict program:

- insert ϕ-functions at the beginning of basic blocks with multiple predecessors;
- insert π-functions at the end of basic blocks with multiple successors;

- insert parallel copies between consecutive instruction in the same basic block; and
- rename variables at every opportunity given by the ϕ-functions, π-functions, and parallel copies.

An elementary program P generated by the grammar A.4 is a sequence of basic blocks. A basic block, which is named by a label L, is a sequence of instructions, starting with a ϕ-function and ending with a π-function. We assume that a program P has two special basic blocks: L_{start} and L_{end}, which are, respectively, the first and last basic blocks to be visited during P's execution. Ordinary instructions either define, or use, one operand, as in $r_1 = v_1$. An instruction such as $v_1 = \bullet$ defines one variable but does not use a variable or register. Parallel copies are represented as $(v_1, \ldots, v_n) = (v'_1, \ldots, v'_n)$.

In order to split the live range of variables, elementary programs use ϕ-functions and π-functions. ϕ-functions are an abstraction used in SSA-form to join the live ranges of variables. An assignment such as:

$$(v_1, \ldots, v_n) = \phi[(v_{11}, \ldots, v_{n1}) : L_1, \ldots (v_{1m}, \ldots, v_{nm}) : L_m]$$

contains n ϕ-functions such as $v_i \leftarrow \phi(v_{i1} : L_1, \ldots, v_{im} : L_m)$. The ϕ symbol works as a multiplexer. It will assign to each v_i the value in v_{ij}, where j is determined by L_j, the basic block last visited before reaching the ϕ assignment. Notice that these assignments happen in parallel, that is, all the variables $v_{1i}, \ldots, v_{ni}$ are simultaneously copied into the variables $v_1, \ldots, v_n$.

The π-functions were introduced in [52] with the name of *swicth nodes*. The name π-node was established in [9]. The π-nodes, or π-functions, as we will call them, are the dual of ϕ-functions. Whereas the latter has the functionality of a variable multiplexer, the former is analogous to a demultiplexer, that performs a parallel assignment depending on the execution path taken. Consider, for instance, the assignment below:

$$[(v_{11}, \ldots, v_{n1}) : L_1, \ldots (v_{1m}, \ldots, v_{nm}) : L_m] = \pi(v_1, \ldots, v_n)$$

which represents m π-nodes such as $(v_{i1} : L_1, \ldots, v_{im} : L_m) \leftarrow \pi(v_i)$. This instruction has the effect of assigning to each variable $v_{ij} : L_j$ the value in v_i if control flows into block L_j. Notice that variables alive in different branches of a basic block are given different names by the π-function that ends that basic block.

Lemma 23 *An elementary program has an elementary interference graph.*

Proof. Let P be an elementary program, let $G = (V, E)$ be P's interference graph, and let T_P be P's dominator tree. We first prove that for any variable v, $LR(v)$ determines a directed path in T_P. Recall that $LR(v)$ consists of the vertices on the unique path from $d(v)$ to $k(v)$. Those vertices are all in the same basic block, possibly except $k(v)$. So every vertex on that path dominates the later vertices on the path, hence $LR(v)$ determines a directed path in T_P. So, G is an RDV-graph. Given a variable v, we let $s(LR(v)) = d(v)$, and we let $e(LR(v)) = k(v)$. The second and third requirements in Lemma 20 follow immediately from the second and third requirements in Definition 22. □

A.1.5 An elementary graph is the interference graph of an elementary program

In this section we show in two steps that any elementary graph is the interference graph of some elementary program.

Lemma 24 *A clique substitution of P_3 is the interference graph of an instruction sequence.*

Proof. Let $G = P_{X,Y,Z}$ be a clique substitution of P_3. Let $m = |X|$, $n = |Y|$ and $p = |Z|$. We build a sequence of $2(m + n + p)$ instructions $i_1, \ldots i_{2(m+n+p)}$ that use $m+n+p$ variables, such that each instruction either defines or uses one variable:

$$\begin{array}{lrcll}
i_j & v_j & = & \bullet & \text{for } j \in 1..n \\
i_{n+j} & v_{n+j} & = & \bullet & \text{for } j \in 1..m \\
i_{n+m+j} & \bullet & = & v_{n+j} & \text{for } j \in 1..m \\
i_{n+2m+j} & v_{n+m+j} & = & \bullet & \text{for } j \in 1..p \\
i_{n+2m+p+j} & \bullet & = & v_{n+m+j} & \text{for } j \in 1..p \\
i_{n+2m+2p+j} & \bullet & = & v_j & \text{for } j \in 1..n
\end{array}$$

Figure A.5 illustrates the instructions. It is straightforward to show that $P_{X,Y,Z}$ is the interference graph of the instruction sequence. □

Lemma 25 *An elementary graph is the interference graph of an elementary program.*

Proof. Let G be an elementary graph and let $C_1, \ldots, C_n$ be the connected components of G. Each C_i is a clique substitution of P_3 so from Lemma 24 we have that each C_i is the interference graph of an instruction sequence s_i. We build an elementary program P with $n + 2$ basic blocks: $B_{start}, B_1, \ldots, B_n, B_{end}$, such that B_{start} contains a single jump to B_1, each B_i consists of s_i followed by a single jump to B_{i+1}, for $1 \leq i \leq n - 1$, and B_n consists of s_n followed by a single jump to B_{end}. The interference graph of the constructed program is G. □

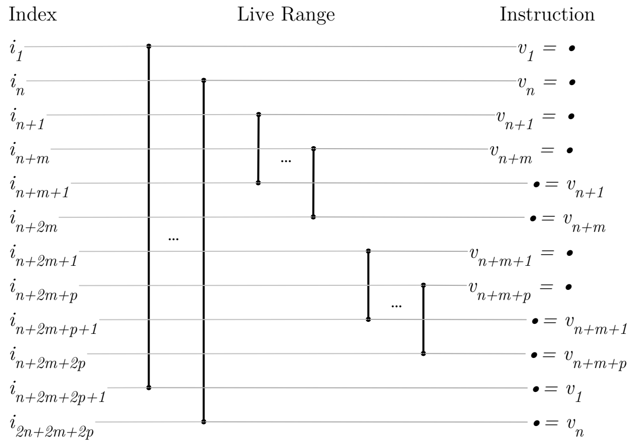


Figure A.5: An elementary program representing a clique substitution of P_3.

A.1.6 From Aligned 1-2-coloring to Puzzle Solving

We now show that aligned 1-2-coloring extension for clique substitutions of P_3 and puzzle solving are equivalent under linear-time reductions. Our proof is in two steps: first we show how to simplify the aligned 1-2-coloring extension problem by *padding* a graph, and then we show how to map a graph to a puzzle.

Padding of puzzles has been defined in Section 3.3. A similar concept applies to clique substitutions of P_3. We say that a graph $P_{X,Y,Z}$ is $2K$*-balanced*, if (1) the weight of X equals the weight of Z, and (2) the weight $X \cup Y$ is $2K$. We pad $P_{X,Y,Z}$ by letting X', Z' be sets of fresh vertices of weight one such that the padded graph $P_{(X \cup X'),Y,(Z \cup Z')}$ is $2K$-balanced. It is straightforward to see that padding executes in linear time. Figure A.6 shows an example of padding. The original graph has two maximal cliques: $K_X \cup K_Y$ with weight 5 and $K_Y \cup K_Z$ with weight 4. We use square nodes to denote vertices of weight two. After the padding, each maximal clique of the resulting graph has weight 6.

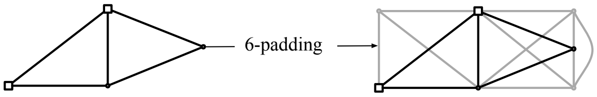


Figure A.6: Example of padding. Square nodes represent vertices of weight two, and the other nodes represent vertices of weight one.

Lemma 26 *For any partial aligned 1-2-coloring ϕ whose domain is a subset of $X \cup Y \cup Z$, we have that $(2K, P_{X,Y,Z}, \phi)$ is solvable if and only if $(2K, P_{(X \cup X'),Y,(Z \cup Z')}, \phi)$ is solvable.*

Proof.

($\Rightarrow$) Let $G = P_{X,Y,Z}$, and let $G_p = P_{(X \cup X'),Y,(Z \cup Z')}$. Let c be a function that associates vertices of G with colors, and let c_p be a function that associates vertices of G_p with colors. Let v be a vertex of G, and let v_p be the vertex of G_p that corresponds to v. Finally, let s be a short vertex of G_p created due to padding. Given c, we let $c_p(v_p) = c(v)$. After coloring vertices v_p, we greedily color the vertices s. Let M_1 and M_2 be the maximal cliques of G. Because M_1 is a comparability graph, the coloring of M_1 uses a number of colors equal to its weight w_1 [42, pp.133]. The same holds true for M_2. Let Q_1 be the maximal clique of G_p that corresponds to M_1, let s_1 be the short vertices of Q_1 created due to padding and let v_1 be the vertices of Q_1 which have corresponding vertices in M_1. We define w_2, Q_2, v_2 and s_2 in analogous way. The number of s_1 vertices in Q_1 equals $2K - w_1$, by the definition of padding, and the coloring of vertices v_1 requires exactly w_1 colors. We color the s_1 vertices with the remaining $2K - w_1$ colors. The number of s_2 vertices in Q_2 equals $2K - w_2$, and the coloring of vertices v_2 requires exactly w_2 colors. We color the s_2 vertices with the remaining $2K - w_2$ colors. The colors greedily assigned to vertices s_1 and s_2 constitute a valid coloring, because s_1 vertices are not adjacent to s_2 vertices.

($\Leftarrow$) Given c_p, we build c as $c(v) = c_p(v_p)$. □

We now define a bijection F from the aligned 1-2-coloring extension problem for $2K$-balanced clique substitutions of P_3 to puzzle solving. We will view a board with K areas as a 2-dimensional $2 \times 2K$ table, in which the i'th area consists of the squares with indices $(1, 2i), (1, 2i+1), (2, 2i)$ and $(2, 2i+1)$.

Let $(2K, G, \phi)$ be an instance of the aligned 1-2-coloring extension problem, where G is a $2K$-balanced clique substitution of P_3. We define a puzzle $F(2K, G, \phi)$ with K areas and the following pieces:

- $\forall v \in X$, weight of v is one: a size-1 X-piece. If $\phi(v)$ is defined and $\phi(v) = i$, then the piece is placed on the square $(1, i)$, otherwise the piece is off the board.

- $\forall v \in X$, weight of v is two: a size-2 X-piece. If $\phi(v)$ is defined and $\phi(v) = \{2i, 2i+1\}$, then the piece is placed on the upper row of area i, otherwise the piece is off the board.

- $\forall v \in Y$, weight of v is one: a size-2 Y-piece. If $\phi(v)$ is defined and $\phi(v) = i$, then the piece is placed on the squares $(1, i)$ and $(2, i)$, otherwise the piece is off the board.

- $\forall v \in Y$, weight of v is two: a size-4 Y-piece. If $\phi(v)$ is defined and $\phi(v) = \{2i, 2i+1\}$, then the piece is placed on area i. otherwise the piece is off the board.

- $\forall v \in Z$, weight of v is one: a size-1 Z-piece. If $\phi(v)$ is defined and $\phi(v) = i$, then the piece is placed on the square $(2, i)$, otherwise the piece is off the board.

- $\forall v \in Z$, weight of v is two: a size-2 Z-piece. If $\phi(v)$ is defined and $\phi(v) = \{2i, 2i+1\}$, then the piece is placed on the lower row of area i, otherwise the piece is off the board.

Given that ϕ is a partial aligned 1-2-coloring of G, we have that the pieces on the board don't overlap. Given that G is $2K$-balanced, we have that the pieces have a total size of $4K$ and that the total size of the X-pieces is equal to the total size of the Z-pieces.

It is straightforward to see that F is injective and surjective, so F is a bijection. It is also straightforward to see that F and F^{-1} both execute in $O(K)$ time.

Lemma 27 *Aligned 1-2-coloring extension for a clique substitution of P_3 is equivalent to puzzle solving.*

Proof. First we reduce aligned 1-2-coloring extension to puzzle solving. Let $(2K, G, \phi)$ be an instance of the aligned 1-2-coloring extension problem where G is a clique substitution of P_3. Via the linear-time operation of padding, we can assume that G is $2K$-balanced. Use the linear-time reduction F to construct a puzzle $F(2K, G, \phi)$. Suppose $(2K, G, \phi)$ has a solution. The solution extends ϕ to an aligned 1-2-coloring of G, and we can then use F to place all the pieces on the board. Conversely, suppose $F(2K, G, \phi)$ has a solution. The solution places the remaining pieces on the board, and we can then use F^{-1} to define an aligned 1-2-coloring of G which extends ϕ.

Second we reduce puzzle solving to aligned 1-2-coloring. Let Pz be a puzzle and use the linear-time reduction F^{-1} to construct an instance of the aligned 1-2-coloring extension problem $F^{-1}(Pz) = (2K, G, \phi)$, where G is a clique substitution of P_3. Suppose Pz has a solution. The solution places all pieces on the board, and we can then use F^{-1} to define an aligned 1-2-coloring of G which extends ϕ. Conversely suppose $F^{-1}(Pz)$ has a solution. The solution extends ϕ to an aligned 1-2-coloring of G, and we can then use F to place all the pieces on the board. □

A.2 Proof of Theorem 2

Theorem 2 states:

> ***(Correctness)*** *A type-1 puzzle is solvable if and only if our program succeeds on the puzzle.*

We first show that an application of a rule from the algorithm given in Figure 3.7 preserves solvability of a puzzles.

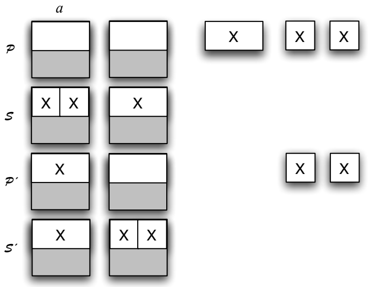
Lemma 28 ***(Preservation)*** *Let Pz be a puzzle and let $i \in \{1, \ldots, 15\}$ be the number of a statement in our program. For $i \in \{11, 12, 13, 14\}$, suppose every area of Pz is either complete, empty, or has just one square already filled in. For $i = 15$, suppose every area of Pz is either complete or empty. Let a be an area of Pz such that the pattern of statement i matches a. If Pz is solvable, then the application of statement i to a succeeds and results in a solvable puzzle.*

Proof. We begin by outlining the proof technique that we will use for each $i \in \{1, \ldots, 15\}$. Notice that statement i contains a rule for each possible strategy that can be used to complete a. Let Sl be a solution of Pz. Given that Sl completes a, it is straightforward to see that the application of statement i to a succeeds, although possibly using a different strategy than Sl. Let Pz' be the result of the application of statement i to a. To see that Pz' is a solvable puzzle, we do a case analysis on (1) the strategy used by Sl to complete a and (2) the strategy used by statement i to complete a. For each case of (1), we analyze the possible cases of (2), and we show that one can rearrange Sl into Sl' such that Sl' is a solution of Pz'. Let us now do the case analysis itself. If statement i is a conditional statement, then we will use $i.n$ to denote the n^{th} rule used in statement i.

$i = 1$. The area a can be completed in just one way. So, Sl uses the same strategy as statement 1 to complete a, hence Sl is a solution of Pz'.

$i \in \{2, 3, 4, 5\}$. The proof is similar to the proof for $i = 1$, we omit the details.

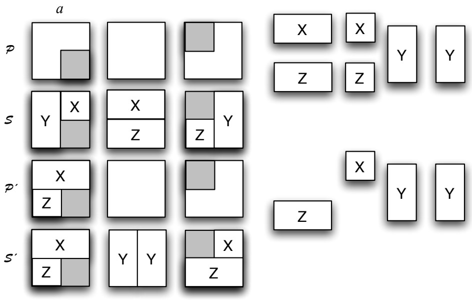
$i = 7$. The area a can be completed in two ways. If Sl uses the strategy of rule 7.1 to complete a, then statement 7 uses that strategy, too, hence Sl is a solution of the resulting puzzle. If Sl uses the strategy of rule 7.2 to complete a, we have two cases. Either statement 7 uses the strategy of rule 7.2, too, in which case Sl is a solution of Pz'. Otherwise, statement 7 uses the strategy of rule 7.1, in which case we can create Sl' from Sl in the following way. We swap the two size-2 X-pieces used by Sl to complete a, with the size-2 X-piece used by statement 7 to complete a. To illustrate the swap, here are excerpts of Pz, Sl, Pz', Sl' for a representative Pz.



It is straightforward to see that Sl' is a solution of Pz'.

$i \in \{8, 9, 10\}$. The proof is similar to the proof for $i = 7$, we omit the details.

$i = 11$. The area a can be completed in three ways. If Sl uses the strategy of rule 11.1 or of rule 11.3 to complete a, the proof proceeds in a manner similar to the proof for $i = 7$, we omit the details. If Sl uses the strategy of rule 11.2 to complete a, we have two cases. Either statement 11 uses the strategy of rule 11.2, too, in which case Sl is a solution of Pz'. Otherwise, statement 11 uses the strategy of rule 11.1, and now we have several of subcases of Sl. Because of the assumption that all areas of Pz are either complete, empty, or has just one square already filled in, the following subcase is the most difficult; the other subcases are easier and omitted. Here are excerpts of Pz, Sl, Pz', Sl'.



It is straightforward to see that Sl' is a solution of Pz'.

$i \in \{12, 13, 14\}$. The proof is similar to the proof for $i = 11$, we omit the details.

$i = 15$. The proof is similar to the proof for $i = 11$, with a total of 28 subcases. All the subcases turn out to be easy because of the assumption that all areas of Pz are either complete or empty. We omit the details. □

We can now prove Theorem 2 (**Correctness**).

Proof. Suppose first that Pz is a solvable puzzle. We must show that our program succeeds on Pz, that is, all the 15 statements succeed. From Lemma 28 and induction on the statement number we have that indeed all 15 statements succeed.

Conversely, suppose Pz is a puzzle and that our program succeeds on Pz. Statements 1–4 complete all areas with three squares already filled in. Statements 5–10 complete all areas with two squares already filled in. Statements 11–14 complete all areas with one square already filled in. Statement 15 completes all areas with no squares already filled in. So, when our program succeeds on Pz, the result is a solution to the puzzle. □

As a collary we get the following complexity result.

Lemma 29 *The aligned 1-2-coloring extension problem for an elementary graph G is solvable in $O(C \times K)$, where C is the number of connected components of G, and $2K$ is the number of colors.*

Proof. Let $(2K, G, \phi)$ be an instance of the aligned 1-2-coloring extension problem for which G is an elementary graph. We first list the connected components of G in linear time [26]. All the connected components of G are clique substitutions of P_3. Next, for each connected component, we have from Lemma 27 that we can reduce the aligned 1-2-coloring extension problem to a puzzle solving problem in linear time. Finally, we run our linear-time puzzle solving program on each of those puzzles (Theorem 2). The aligned 1-2-coloring extension problem is solvable if and only if all those puzzles are solvable. The total running time is $O(C \times K)$. □

A.3 Proof of Theorem 4

Theorem 4 states:

> ***(Hardness)*** *Register allocation with pre-coloring and spilling of families of variables for an elementary program is NP-complete.*

We reduce this problem to the maximal K-colorable subgraph of a chordal graph, which was proved to be NP-complete by Yannakakis and Gavril [91]. The key step is to show that any chordal graph is the interference graph of a program in SSA form. We first define a convenient representation of chordal graphs. Suppose we have a tree T and a family V of subtrees of T. We say that (T, V) is a *program-like decomposition* if for for all $\sigma \in V$ we have that:

1. the root of σ has one successor.
2. Each leaf of σ has zero or one predecessor.
3. Each vertex of T is the root of at most one element of V.
4. A vertex of T is the leaf of at most one element of V, in which case it is not the root of any subtree.
5. Each element of V contains at least one edge.

For each subtree $\sigma \in V$, we identify root_σ as the vertex of σ that is the closest to the root of T.

In order to prove that any chordal graph has a program like decomposition, we rely on the concept of *nice tree decomposition* [61]. Given a nice tree T, for each vertex $x \in T$ we denote by K_x the union of all the subtrees that touch x. T satisfies the following properties:

1. every node x has at most two children.
2. If $x \in T$ has two children, $y, z \in T$, then $K_x = K_y = K_z$. In this case, x is called a *joint* vertex.
3. If $x \in T$ has only one child, $y \in T$, then $K_x = K_y \cup \{y\}$, or $K_x = K_y \setminus \{y\}$.

4. If $x \in T$ has no children, then K_x is reached by at most one subtree, and x is called a *leaf* node.

Figure A.7 (b) shows a nice tree decomposition produced for the graph in Figure A.7 (a). The program like decomposition is given in Figure A.7 (c).

Lemma 30 *A graph is chordal if and only if it has a program like tree decomposition.*

Proof. $\Leftarrow$: immediate.

$\Rightarrow$: A graph is chordal if and only if it has a *nice* tree decomposition [61]. Given a chordal graph, and its nice tree decomposition, we build a *program like* decomposition as follows:

1. the only nodes that have more than one successor are the joint nodes. If a joint node v is the root of a subtree, replicate v. Let v' be the replicated node. Add the predecessor of v as the predecessor of v', and let the unique predecessor of v be v'. Now, v' is the root of any subtree that contains v.

2. this is in accordance to the definition of nice tree, for joint nodes are never leaves of subtrees.

3. If there is $v \in T$ such that v is the root of $\sigma_x, \sigma_y \in V$, then replicate v. Let v' be the replicated node in such a way that $K_{v'} = K_v \setminus \{x\}$. Add the predecessor of v as the predecessor of v', and let the unique predecessor of v be v'. Now, v' is the root of any subtree that reaches v, other than σ_y.

4. If there is $v \in T$ such that v is the leaf of $\sigma_x, \sigma_y \in V$, then replicate v. Let v' be the replicated node in such a way that $K_{v'} = K_v \setminus \{x\}$. Add the sucessor of v as the successor of v', and let the unique successor of v be v'. Now, v' is the leaf of any subtree that reaches v, except σ_y.

5. If there is a subtree that only spans one node, replicate that node as was done in (1).

$\square$

We next define simple notions of *statement* and *program* that are suitable for this paper. We use v to range over program variables. A statement is defined by the grammar:

$$\begin{array}{rcll} (\text{Statement})\ s & ::= & v\ \texttt{=} & (\text{definition of } v) \\ & | & \texttt{=}\ v & (\text{use of } v) \\ & | & \texttt{skip} & \end{array}$$

A program is a tree-structured flow chart of a particular simple form: a program is a pair (T, ℓ) where T is a finite tree, ℓ maps each vertex of T with zero or one successor to a statement, and each variable v is defined exactly once and the definition of v dominates all uses of v. Notice that a program is in strict SSA form.

The *interference graph* of a program (T, ℓ) is an intersection graph of a family of subtrees V of T. The family of subtrees consists of one subtree, called the *live range*, per variable v in the program; the live range is the subtree of the finite tree induced by the set of paths from each use of v to the definition of v. Notice that a live range consists of both vertices and edges (and not, as is more standard, edges only). That causes no problem here because we don't allow a live range to end in the same node as another live range begins.

From a chordal graph G presented as a finite tree T and a program-like family of subtrees V, we construct a program $P_G = (T, \ell)$, where for each subtree $\sigma \in V$, we define $\ell(\text{root}_\sigma)$ to be "v_σ $\texttt{=}$", and for each subtree $\sigma \in V$, and a leaf n of σ, we define $\ell(n)$ to be "$\texttt{=}$ v_σ". Figure A.7(d) shows the program that corresponds to the tree in Figure A.7 (c).

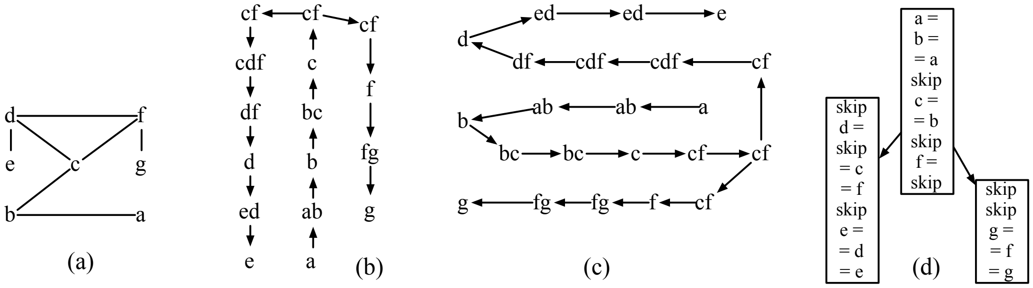


Figure A.7: A chordal graph represented as a program.

Lemma 31 *G is the interference graph of P_G.*

Proof. For all $\sigma \in V$, the live range of v_σ in P is σ. □

In Section 3.4 we introduced families of variables in an elementary program. This concept is formally defined as:

Definition 32 *Let P_s to be a strict program, and let P_e to be the corresponding elementary program. Given a variable $v \in P_s$, the set Q_v of all the variables in P_e produced from the renaming of v is called the family of variables v.*

We emphasize that the union of the live ranges of all the variables in a family Q_v is topologically equivalent to the live range of v. We state this fact as Lemma 33.

Lemma 33 *Let P_s be a strict program, and let P_e be the elementary program derived from P_s. Let v and u be two variables of P_s, and let Q_v and Q_u be the corresponding families of variables in P_e. The variables v and u interfere if, and only if, there exists $v' \in Q_v$ and $u' \in Q_u$ such that v' and u' interfere.*

Proof. Follows from definition 32. □

Theorem 34 *The maximal aligned 1-2-coloring extension problem for elementary graphs is NP-complete.*

Proof. The problem of finding the maximum induced subgraph of a chordal graph that is K colorable is NP-complete [91]. We combine this result with Lemmas 31 and 33 for the proof of this theorem. □

The proof of Theorem 4 is a corollary of Theorem 34:

Proof. Follows from Theorem 34. □

A.4 Proof of Theorem 13

Theorem 13 was stated as follows:

> ***(Correctness)*** *For a spartan location transfer graph G,* ***ImplementSpartan***(G) *is a correct implementation of G.*

By Theorem 12, G must be either a cycle or a path; thus, we divide this proof into two parts: Lemma 35 and Lemma 36. The semantics of parallel copies are defined in the obvious way:

$$\langle I, (l_1, \ldots, l_n) := (l'_1, \ldots, l'_n)\rangle \rightarrow \rho[l_1 \leftarrow \rho(l'_1), \ldots l_n \leftarrow \rho(l'_n)]. \tag{A.1}$$

Lemma 35 *If μ is a spartan parallel copy and its location transfer graph is a cycle, then there is a sequence of $n-1$ swaps in the language* ***`Seq`*** *that is semantically equivalent to μ.*

Proof. The proof is by induction on the length of μ. By Theorem 12 all the locations in μ are registers, because μ is a cycle by hypothesis.

Base case: if μ has lenght two, then by Equation A.1 we have that $(r_1, r_2) := (r_2, r_1) \equiv r_1 \oplus r_2$.

Induction hypothesis: the theorem is true for parallel copies with up to $n - 1$ variables on each side.

Induction step: we consider the parallel copy $(r_1, r_2, \ldots, r_{n-1}, r_n) := (r_2, r_3, \ldots, r_n, r_1)$ applied on the environment ρ, where $\rho(r_i) = v_i$. If we apply $r_1 \oplus r_n$ on ρ, we get the environment $\rho' = \rho[r_n \leftarrow v_1][r_n \leftarrow v_1]$. Register r_n has the location that would be assigned to it by μ. Consider now the parallel copy $\mu' = (r_1, r_2, \ldots, r_{n-2}, r_{n-1}) := (r_2, r_3, \ldots, r_{n-1}, r_1)$. The parallel copy μ' is similar to μ, except that r_1 sends its value to r_{n-1}, and r_n is no longer present. But r_1 contains now v_n, the value that should be transfered to r_{n-1}. The result follows by applying induction on μ', which has size n. □

Lemma 36 *If μ is a spartan parallel copy and its location transfer graph is a path, then there is a sequence of $n - 1$ swaps in the language* `Seq` *that is semantically equivalent to μ.*

Proof. The proof is by induction on the length of μ, and it is similar to the proof of Lemma 35. □

The proof of Theorem 13 follows by combining the two previous lemmas, plus the fact that any component of a lcoation transfer graph is either a cycle or a path.

A.5 Proof of Theorem 14

In this section we prove Theorem 14, which we re-state as follows:

(Register Pressure) *Let P be a program whose control flow graph does not contain critical edges. The SSA-to-CSSA conversion does not increase the global register pressure in P.*

We will assume that P is in strict, pruned SSA-form. A program is in strict SSA form [21] if it is in SSA form and for each variable x, the single definition of x dominates all its uses. A program is in pruned SSA-form if none of the variables defined by a ϕ-function is a dead-definition [25]. We recall the definition of liveness analysis, as given by Appel and Palsberg [4, p.206], where l is a statement in the program, $in[l]$ is the set of variables live before l, $out[l]$ is the set of variables live after l, $def[l]$ is the set of variables defined at l, $use[l]$ is the set of variables used at l, and $succ[l]$ is the set of statements that succeed l.

$$\begin{aligned} in[l] &= use[l] \cup (out[l] - def[l]) \\ out[l] &= \bigcup_{s \in succ[l]} in[s] \end{aligned} \tag{A.2}$$

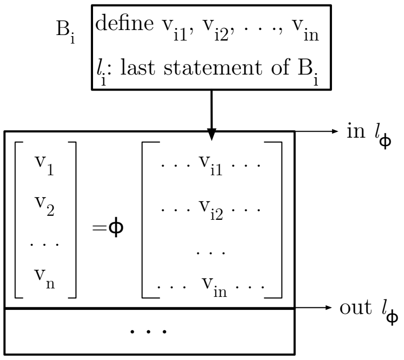


Figure A.8: The parallel copy $l_\phi : (v_1, v_2, \ldots, v_n) := (v_{i1}, v_{i2}, \ldots, v_{in})$.

Lemma 37 *Let P be a strict program in pruned-CSSA-form with no critical edges. If l_i and l_ϕ are defined as in Figure A.8, then $|out(l_i)| = |out(l_\phi)|$.*

Proof. In order to prove this lemma, we will use the claims listed below, where $X_i = \{v_{i1}, v_{i2}, \ldots, v_{in}\}$, and $X_\phi = \{v_1, v_2, \ldots, v_n\}$:

1. $out[l_i] = in[l_\phi]$;
2. $v_{ij} \notin out[l_\phi], 1 \leq j \leq n$;
3. $X_i \cap out[l_\phi] = \emptyset$;
4. $out[l_i] - X_i = out[l_\phi] - X_\phi$;
5. $v_{ij} \neq v_{ik}, 1 \leq j, k \leq n$ and $j \neq k$;
6. $|X_i| = |X_\phi|$;
7. $X_i \subseteq out[l_i]$;
8. $X_\phi \subseteq out[l_\phi]$;

We proof these claims as follows:

- **proof of claim 1** This follows from Equation A.2, plus the fact that P has no critical edges, so $\bigcup\limits_{s \in succ[l_i]} = \{l_\phi\}$.
- **proof of claim 2** If we assume otherwise, v_{ij} would interfere with all v_j. We have that $v_j \in out[l_\phi]$ because P is pruned. However, v_j and v_{ij} cannot interfere because P is in CSSA-form, and v_{ij} and v_j are ϕ-related.
- **proof of claim 3** Follows as a simple corollary of claim 2.
- **proof of claim 4** According to Equation A.2:
  $in[l_\phi] = use[l_\phi] \cup (out[l_\phi] - def[l_\phi]) = X_i \cup (out[l_\phi] - X_\phi)$
  From claim 1:
  $out[l_i] = X_i \cup (out[l_\phi] - X_\phi)$

From claim 3:

$out[l_i] - X_i = out[l_\phi] - X_\phi$

- **proof of claim 5** by the definition of CSSA-form program.

- **proof of claim 6** Follows as a simple corollary of claim 5.

- **proof of claim 7** Follows from Equation A.2, plus claim 1, e.g: $out[l_i] = in[l_\phi] = X_i \cup (\ldots)$.

- **proof of claim 8** This claim follows from the fact that we are dealing with a program in pruned-SSA-form. In this case, none of the variables defined by ϕ-functions are dead at the definition point.

Finally, to prove our final result, e.g $|out(l_i)| = |out(l_\phi)|$, we combine claims 4, 6, 7 and 8. □

The *global register pressure* of a program is bounded by the maximum number of variables alive at any point of the program. A program in SSA-form never requires more registers than its global registerpressure.Theorem 14 shows that the procedure **PhiLifting** preserves the global register pressure of the target program, that is, if P is a program in pruned-SSA-form that could be compiled with K registers before being transformed by **PhiLifting**, it still can be compiled with K registers after it.

We now prove Theorem 14:

Proof. Given a ϕ-function such as $a_i : B = \phi(a_{i1} : B_1, a_{i2} : B_2, \ldots, a_{im} : B_m)$, the procedure **PhiLifting** changes it in two ways:

1. it splits the live range of the variable defined by the ϕ-function with an instruction $I = \langle a_i := v_i \rangle$.

2. it splits the live ranges of the variables used in the ϕ-function with m instructions like $I_j = \langle v_{ij} := a_{ij} \rangle$.

We will show that each transformation preserves the global register pressure of the source program.

1. Because P is a program in pruned-SSA-form, each variable defined by a ϕ-function is alive past its definition point. The variable v_i inserted by **PhiLifting** is alive from the ϕ-function until instruction I. Variable a_i is alive thereafter.Thus, variable v_i does not increase the register pressure in P, because v_i and a_i are never simultaneously alive.

2. From Lemma 37 we know that the register pressure at the end of a basic block that feeds a ϕ-equation $V = \phi M$ is bounded by the register pressure at program point l_ϕ, past the definition point of V, and, from the proof of (1) above, we know that the register pressure at l_ϕ remains constant after the source program is modified by ϕ-lifting.

□

References

[1] Scott Ananian. The static single information form. Master's thesis, MIT, September 1999.

[2] Andrew W. Appel. SSA is functional programming. *SIGPLAN Notices*, 33(4):17–20, 1998.

[3] Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. In *PLDI*, pages 243–253. ACM Press, 2001.

[4] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition, 2002.

[5] Rajkishore Barik, Christian Grothoff, Rahul Gupta, Vinayaka Pandit, and Raghavendra Udupa. Optimal bitwise register allocation using integer linear programming. In *LCPC*, volume 4382 of *Lecture Notes in Computer Science*, pages 267–282. Springer, 2006.

[6] L. Belady. A study of the replacement of algorithms of a virtual storage computer. *IBM System Journal*, 5:78–101, 1966.

[7] Anne Berry, Jean Blair, Pinar Heggernes, and Barry Peyton. Maximum cardinality search for computing minimal triangulations of graphs. *Algorithmica*, 39(4):287 – 298, 2004.

[8] M Biró, M Hujter, and Zs Tuza. Precoloring extension. I:interval graphs. In *Discrete Mathematics*, pages 267 – 279. ACM Press, 1992. Special volume (part 1) to mark the centennial of Julius Petersen's "Die theorie der regularen graphs".

[9] Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *Conference on Programming Language Design and Implementation*, pages 321–333, 2000.

[10] Florent Bouchez. Allocation de registres et vidage en mémoire. Master's thesis, ENS Lyon, October 2005.

[11] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register allocation: What does the NP-completeness proof of chaitin et al. really prove? In *5th Annual Workshop in Duplicating, Deconstructing, and Debunking*, 2006.

[12] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of register coalescing. In *CGO*, pages 102 – 104. IEEE, 2007.

[13] Robert S. Boyer and J. Strother Moore. Mjrty: A fast majority vote algorithm. In *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 105–118, 1991.

[14] Preston Briggs, Keith D. Cooper, Timothy 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.

[15] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *PLDI*, pages 311–321, 1992.

[16] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):428–455, 1994.

[17] Philip Brisk. *Advances in Static Single Assignment Form and Register Allocation.* PhD thesis, UCLA - University of California, Los Angeles, 2006.

[18] Philip Brisk, Foad Dabiri, Roozbeh Jafari, and Majid Sarrafzadeh. Optimal register sharing for high-level synthesis of ssa form programs. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(5):772–779, 2006.

[19] Philip Brisk, Foad Dabiri, Jamie Macbeth, and Majid Sarrafzadeh. Polynomial-time graph coloring register allocation. In *14th International Workshop on Logic and Synthesis.* ACM Press, 2005.

[20] Philip Brisk and Majid Sarrafzadeh. Interference graphs for procedures in static single information form are interval graphs. In *10th international workshop on Software and compilers for embedded systems*, pages 101–110. ACM Press, 2007.

[21] Zoran Budimlic, Keith D. Cooper, Timothy J. Harvey, Ken Kennedy, Timothy S. Oberg, and Steven W. Reeves. Fast copy coalescing and live-range identification. In *PLDI*, pages 25–32. ACM Press, 2002.

[22] Rainer E. Burkard, Eranda Çela, Panos M. Pardalos, and Leonidas S. Pitsoulis. Quadratic assignment problems. *European Journal of Operational Research*, 15:283–289, 1984.

[23] G. J. Chaitin. Register allocation and spilling via graph coloring. *Symposium on Compiler Construction*, 17(6):98–105, 1982.

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

[25] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *POPL*, pages 55–66, 1991.

[26] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Cliff Stein. *Introduction to Algorithms*. McGraw-Hill, 2nd edition, 2001.

[27] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.

[28] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw-Hill Science/Engineering/Math, 2006.

[29] François de Ferriére, Christophe Guillon, and Fabrice Rastello. Optimizing the translation out-of-SSA with renaming constraints. *ST Journal of Research Processor Architecture and Compilation for Embedded Systems*, 1(2):81–96, 2004.

[30] Dietmar Ebner, Florian Brandner, Bernhard Scholz, Andreas Krall, Peter Wiedermann, and Albrecht Kadlec. Generalized instruction selection using SSA-graphs. In *LCTES*, pages 31–40, 2008.

[31] Erik Eckstein, Oliver König, and Bernhard Scholz. Code instruction selection based on ssa-graphs. In *SCOPES*, pages 49–65, 2003.

[32] Erik Eckstein and Bernhard Scholz. Addressing mode selection. In *CGO*, pages 337–346, 2003.

[33] S Even, A Itai, and A Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM J. Computing*, 5(4):691 – 703, 1976.

[34] Alkis Evlogimenos. Improvements to linear scan register allocation. Technical report, University of Illinois, Urbana-Champaign, 2004.

[35] Martin Farach and Vincenzo Liberatore. On local register allocation. In *9th ACM-SIAM symposium on Discrete Algorithms*, pages 564–573. ACM Press, 1998.

[36] Changqing Fu and Ken D. Wilken. A faster optimal register allocator. In *Internation Symposium on Microarchitecture*, pages 245–256. ACM, 2002.

[37] Fanica Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SICOMP*, 1(2):180 – 187, 1972.

[38] Fanica Gavril. The intersection graphs of subtrees of a tree are exactly the chordal graphs. *Journal of Combinatoric*, B(16):46 – 56, 1974.

[39] Fanica Gavril. A recognition algorithm for the intersection graphs of directed paths in directed trees. *Discrete Mathematics*, 13:237 – 249, 1975.

[40] Lal George and Andrew W. Appel. Iterated register coalescing. *Transactions on Programming Languages and Systems (TOPLAS)*, 18(3):300–324, 1996.

[41] Martin Charles Golumbic. Trivially perfect graphs. *Discrete Mathematics*, 24:105 – 107, 1978.

[42] Martin Charles Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Elsevier, 1st edition, 2004.

[43] David W. Goodwin and Ken D. Wilken. Optimal and near-optimal global register allocations using 0-1 integer programming. *SPE*, 26(8):929–965, 1996.

[44] Brian J. Gough. *An Introduction to GCC*. Network Theory Ltd, 1st edition, 2005.

[45] Daniel Grund and Sebastian Hack. A fast cutting-plane algorithm for optimal coalescing. In *Compiler Construction*, volume 4420, pages 111–115. Springer, 2007.

[46] Sebastian Hack and Gerhard Goos. Optimal register allocation for SSA-form programs in polynomial time. *Information Processing Letters*, 98(4):150–155, 2006.

[47] Sebastian Hack and Gerhard Goos. Copy coalescing by graph recoloring. In *PLDI*, pages 227–237, 2008.

[48] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in SSA-form. In *15th Conference on Compiler Construction*, pages 247–262. Springer-Verlag, 2006.

[49] Lang Hames and Bernhard Scholz. Nearly optimal register allocation with PBQP. In *JMLC*, pages 346–361. Springer, 2006.

[50] Ulrich Hirnschrott, Andreas Krall, and Bernhard Scholz. Graph coloring vs. optimal register allocation for optimizing compilers. In *JMLC*, pages 202–213, 2003.

[51] Corporate SPARC International Inc. *The SPARC Architecture Manual, Version 8*. Prentice Hall, 1st edition, 1992.

[52] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *Conference on Programming Language Design and Implementation*, pages 78–89, 1993.

[53] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Plenum, 1972.

[54] A. B. Kempe. On the geographical problem of the four colours. *American Journal of Mathematics*, 2(3):193 – 200, 1879.

[55] David Koes and Seth Copen Goldstein. A progressive register allocator for irregular architectures. In *CGO*, pages 269–280, 2005.

[56] David Ryan Koes and Seth Copen Goldstein. A global progressive register allocator. In *PLDI*, pages 204–215, 2006.

[57] Timothy Kong and Kent D Wilken. Precise register allocation for irregular architectures. In *Internation Symposium on Microarchitecture*, pages 297–307. ACM, 1998.

[58] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.

[59] Jonathan K. Lee, Jens Palsberg, and Fernando M. Q. Pereira. Aliased register allocation. In *ICALP*, 2007.

[60] Daniel Marx. A short proof of the NP-completeness of circular arc coloring, 2003.

[61] Dániel Marx. Parameterized coloring problems on chordal graphs. *Theoretical Computer Science*, 351(3):407–424, 2006.

[62] Daniel Marx. Precoloring extension on unit interval graphs. *Discrete Applied Mathematics*, 154(6):995 – 1002, 2006.

[63] Clyde L. Monma and V. K. Wei. Intersection graphs of paths in a tree. *Journal of Combinatorial Theory Series B*, 41(2):141 – 181, 1986.

[64] Hanspeter Mossenbock and Michael Pfeiffer. Linear scan register allocation in the context of SSA form and register constraints. In *CC*, pages 229–246. LNCS, 2002.

[65] Mayur Naik and Jens Palsberg. Compiling with code size constraints. *Transactions on Embedded Computing Systems*, 3(1):163–181, 2004.

[66] V. Krishna Nandivada, Fernando Pereira, and Jens Palsberg. A framework for end-to-end verification and evaluation of register allocators. In *Proceedings of SAS'07, International Static Analysis Symposium*, pages 153–169, Kongens Lyngby, Denmark, August 2007.

[67] Venkata Krishna Nandivada and Jens Palsberg. SARA: Combining stack allocation and register allocation. In *15th International Conference on Compiler Construction*. Springer-Verlag, 2006.

[68] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. In *IEEE PACT*, pages 196–204, 1998.

[69] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. *ACM Trans. Program. Lang. Syst.*, 26(4):735–765, 2004.

[70] Sriram V. Pemmaraju and Rajiv Raman. Approximation algorithms for the max-coloring problem. In *ICALP*, pages 1064 – 1075, 2005.

[71] Fernando Magno Quintao Pereira. The minimum register bank problem. Technical report, University of California, Los Angeles, 2008.

[72] Fernando Magno Quintao Pereira. Ssa elimination after register allocation. Technical report, University of California, Los Angeles, 2008.

[73] Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In *APLAS*, pages 315–329. Springer, 2005.

[74] Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation after classic SSA elimination is np-complete. In *Foundations of Software Science and Computation Structures*. Springer, 2006.

[75] Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation by puzzle solving, 2007. http://compilers.cs.ucla.edu/fernando/ projects/ puzzles/.

[76] Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation by puzzle solving. In *PLDI*, pages 216–226, 2008.

[77] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):895–913, 1999.

[78] Laurence Rideau, Bernard P. Serpette, and Xavier Leroy. Tilting at windmills with coq: formal verification of a compilation algorithm for parallel moves, 2008. To appear.

[79] B. K. Rosen, F. K. Zadeck, and M. N. Wegman. Global value numbers and redundant computations. In *POPL*, pages 12–27. ACM Press, 1988.

[80] Konstantinos Sagonas and Erik Stenman. Experimental evaluation and improvements to linear scan register allocation. *Software, Practice and Experience*, 33:1003–1034, 2003.

[81] Vivek Sarkar and Rajkishore Barik. Extended linear scan: an alternate foundation for global register allocation. In *LCTES/CC*, pages 141–155. ACM, 2007.

[82] Bernhard Scholz and Erik Eckstein. Register allocation for irregular architectures. In *LCTES/SCOPES*, pages 139–148. ACM, 2002.

[83] Michael D. Smith, Norman Ramsey, and Glenn Holloway. A generalized algorithm for graph-coloring register allocation. In *PLDI*, pages 277–288, 2004.

[84] Vugranam C. Sreedhar, Roy Dz ching Ju, David M. Gillies, and Vatsa Santhanam. Translating out of static single assignment form. In *SAS*, pages 194–210. Springer-Verlag, 1999.

[85] JVM Team. The java HotSpot virtual machine. Technical Report Technical White Paper, Sun Microsystems, 2006.

[86] The Jikes Team. Jikes RVM home page, 2007. http://jikesrvm.sourceforge.net/.

[87] Omri Traub, Glenn H. Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 142–151, 1998.

[88] David L. Weaver and Tom Germond. *The SPARC Architecture Manual, Version 9*. Prentice Hall, 1st edition, 1994.

[89] John Whaley. Joeq: a virtual machine and compiler infrastructure. In *Workshop on Interpreters, virtual machines and emulators*, pages 58–66. ACM Press, 2003.

[90] Christian Wimmer and Hanspeter Mossenbock. Optimized interval splitting in a linear scan register allocator. In *VEE*, pages 132–141. ACM, 2005.

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