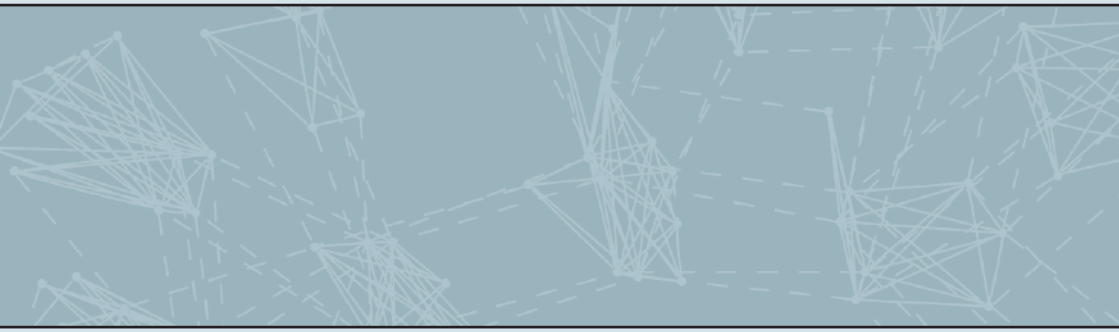
Sebastian Hack



Register Allocation for Programs in SSA Form

universitätsverlag karlsruhe

Sebastian Hack

Register Allocation for Programs in SSA Form

Register Allocation for Programs in SSA Form

by
Sebastian Hack

universitätsverlag karlsruhe

Dissertation, Universität Karlsruhe (TH)
Fakultät für Informatik, 2006

Impressum

Universitätsverlag Karlsruhe
c/o Universitätsbibliothek
Straße am Forum 2
D-76131 Karlsruhe
www.uvka.de

Universitätsverlag Karlsruhe 2007
Print on Demand

ISBN: 978-3-86644-180-4

Register Allocation for Programs in SSA Form

zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften

der Fakultät für Informatik
der Universität Fridericiana zu Karlsruhe (TH)
genehmigte

Dissertation

von

Sebastian Hack
aus Heidelberg

Tag der mündlichen Prüfung: 31.10.2006
Erster Gutachter: Prof. em. Dr. Dr. h. c. Gerhard Goos
Zweiter Gutachter: Prof. Dr. Alan Mycroft

Acknowledgements

This thesis is the result of the research I did at the chair of Prof. Goos at the Universität Karlsruhe from May 2004 until October 2006. In these two and a half years I had the chance to meet, work and be with so many great people. I here want to take the opportunity to express my gratitude to them.

I am very obliged to Gerhard Goos. He supported my thesis from the very beginning and was at any time open for discussions not only regarding the thesis. I definitely learned a lot from him. I thank him for granting me the necessary freedom to pursue my ideas, for his faith in me and all the support I received from him.

Alan Mycroft very unexpectedly invited my to Cambridge to give a talk about my early results. I enjoyed many discussions with him and am very thankful for his advice and support through the years as well as being the external examiner for this thesis.

Both referees provided many excellent comments which improved this thesis significantly.

I am very thankful to my colleagues Mamdouh Abu-Sakran, Michael Beck, Boris Boesler, Rubino Geiß, Dirk Heuzeroth, Florian Liekweg, Götz Lindenmaier, Markus Noga, Bernd Traub and Katja Weißhaupt for such a pleasant time in Karlsruhe, their friendship and many interesting and fruitful discussions and conversations from which I had the chance to learn so much; not only regarding computer science.

I had immense help from many master students in Karlsruhe to realise this project. Their scientific vitality was essential for my work. I already miss their extraordinary dedication and the high quality of their work. Thank you Veit Batz, Matthias Braun, Daniel Grund, Kimon Hoffmann, Enno Hofmann, Hannes Jakschitsch, Moritz Kroll, Christoph Mallon, Johannes Spallek, Adam M. Szalkowski and Christian Würdig.

I would also like to thank many people outside Karlsruhe who helped to improve my research with many insightful discussions. I would like to thank:

Benoit Boissinot, Florent Bouchez, Philip Brisk, Alain Darte, Jens Palsberg, Fernando Pereira, Fabrice Rastello and Simon Peyton-Jones.

I thank André Rupp for the many opportunities he opened for me and his friendship over the last years.

Nothing would have been possible without my family. I thank my parents and my brother for their support. My beloved wife Kerstin accompanied me through all the ups and downs a PhD thesis implies and always supported me. Thank you for being there.

Lyon, September 2007
Sebastian Hack

Contents

List of Symbols xv

1 Introduction 1

1.1 Graph-Coloring Register Allocation . . . 2
1.2 SSA-based Register Allocation . . . 3
1.3 Overview of this Thesis . . . 6

2 Foundations 7

2.1 Lists and Linearly Ordered Sets . . . 7
2.2 Programs . . . 8
2.3 Static Single Assignment (SSA) . . . 11
2.3.1 Semantics of Φ-operations . . . 12
2.3.2 Non-Strict Programs and the Dominance Property . . . 13
2.3.3 SSA Destruction . . . 15
2.4 Global Register Allocation . . . 16
2.4.1 Interference . . . 17
2.4.2 Coalescing and Live Range Splitting . . . 18
2.4.3 Spilling . . . 19
2.4.4 Register Targeting . . . 21

3 State of the Art 23

3.1 Graph-Coloring Register Allocation . . . 24
3.1.1 Extensions to the Chaitin-Allocator . . . 26
3.1.2 Splitting-Based Approaches . . . 28
3.1.3 Region-Based Approaches . . . 29
3.1.4 Other Graph-Coloring Approaches . . . 30
3.1.5 Practical Considerations . . . 31

3.2 Other Global Approaches . . . 33
3.2.1 Linear-Scan . . . 34
3.2.2 ILP-based Approaches . . . 34
3.3 SSA Destruction . . . 35
3.4 Conclusions . . . 35

4 SSA Register Allocation 37
4.1 Liveness, Interference and SSA . . . 37
4.1.1 Liveness and Φ-operations . . . 37
4.1.2 Interference . . . 39
4.1.3 A Colorability Criterion . . . 41
4.1.4 Directions from Here . . . 42
4.2 Spilling . . . 44
4.2.1 Spilling on SSA . . . 44
4.2.2 Generic Procedure . . . 46
4.2.3 Rematerialisation . . . 50
4.2.4 A Spilling Heuristic . . . 50
4.3 Coloring . . . 53
4.4 Implementing Φ-operations . . . 55
4.4.1 Register Operands . . . 55
4.4.2 Memory Operands . . . 58
4.5 Coalescing . . . 59
4.5.1 The Coalescing Problem . . . 59
4.5.2 A Coalescing Heuristic . . . 63
4.5.3 Optimal Coalescing by ILP . . . 71
4.6 Register Targeting . . . 75
4.6.1 Copy Insertion . . . 76
4.6.2 Modelling Register Pressure . . . 80
4.6.3 Interference Graph Precoloring . . . 81
4.6.4 Coloring . . . 82
4.6.5 An Example . . . 83

5 Implementation and Evaluation 87
5.1 Implementation . . . 87
5.1.1 The *Firm* Backend . . . 87
5.1.2 The x86 Architecture . . . 88
5.2 Measurements . . . 91
5.2.1 Quantitative Analysis of Coalescing . . . 92
5.2.2 Runtime Experiments . . . 99

6 Conclusions and Future Work 103
6.1 Conclusions . . . 103
6.2 Future Work . . . 104

A Graphs 105
A.1 Bipartite Graphs and Matchings . . . 106
A.2 Perfect Graphs . . . 106
A.2.1 Chordal Graphs . . . 107

B Integer Linear Programming 111

Bibliography 113

Index 121

List of Symbols

2^X	The powerset of X, page 7
$\&$	Spill slot assignment, page 44
$\mathcal{A}^{arg}$	Register constraints of the arguments of a label, page 75
$\mathcal{A}^{res}$	Register constraints of the results of a label, page 75
arg	List of arguments of an instruction, page 8
$\chi(G)$	Chromatic number of a graph, page 106
$\mathscr{L}(X)$	The set of all lists over X, page 7
$\ell' \xrightarrow{i} \ell$	Predicate to state that the i-th predecessor of ℓ is ℓ', page 8
$\ell' \rightarrow \ell$	Predicate to state that there is a i for which ℓ is the i-th predecessor of ℓ', page 8
$\ell_1 \rightarrow^* \ell_n$	There is a path from ℓ_1 to ℓ_n, page 9
$\ell \preceq \ell'$	ℓ dominates ℓ', page 10
L	Set of labels in the program, page 8
$\mathbf{m}$	A memory variable, page 44
$\omega(G)$	Clique number of a graph, page 105
op	Operation of an instruction, page 8
pred	Linearly ordered set of predecessor labels of a label, page 8
res	List of results of an instruction, page 8
ρ	A register allocation. Maps variables to registers, page 17
$\mathscr{S}(X)$	The set of all linearly ordered sets over X, page 7
start	The start label of a control flow graph, page 8
$\mathcal{T}_\Phi(P)$	Program transformation to express Φ-operation's semantics, page 12
`undef`	The variable with undefined content, page 14
V	Set of a variables in a program, page 8

1 Introduction

One major benefit of higher-level programming languages over machine code is, that the programmer is relieved of assigning storage locations to the values the program is processing. To store the results of computations, almost every processor provides a set of *registers* with very fast access. However, the number of registers is often very small, usually from 8 to 32. So, for complex computations there might not be enough registers available. Then, some of the computed values have to be put into memory which is, in comparison to the register bank, huge but much slower to access. Generally, one talks about a memory hierarchy where the larger a memory is, the slower it is to access. The processor's registers represent the smallest and fastest end of this hierarchy.

Common programming languages do not pay attention to the memory hierarchy for several reasons. First of all, the number, size and speed of the different kinds of memory differ from one machine to another. Secondly, the programmer should be relieved of considering all the details concerning the underlying hardware architecture since the program should efficiently run on as many architectures as possible. These details are covered by the compiler, which translates the program as it is written by the programmer into machine code. Since the compiler targets a single processor architecture in such a translation process, it takes care of these details in order to produce efficient code for the processor. Thus, the compiler should be concerned about assigning as many variables as possible to processor registers. In the case that the number of registers available does not suffice the compiler has to carefully decide which variables will reside in main memory. This whole task is called *register allocation*.

The principle of register allocation is simple: the compiler has to determine for each point in the program which variables are *live*, i.e. will be needed in some computation later on. If two variables being live at the same point in the program, i.e. they are still needed in some future computation, they must not

occupy the same storage location, especially not the same register. We then say the variables *interfere*. The register allocator then has to assign a register to each variable while ensuring that all interfering variables have different registers. However, if the register allocator determines that the number of registers does not suffice to meet the demand of the program, it has to modify the program by inserting explicit memory accesses for variables that could not be assigned a register. This part, called *spilling*, is crucial since memory access is comparatively slow. To quote [Hennessy and Patterson, 1997, page 92]:

> *Because of the central role that register allocation plays, both in speeding up the code and in making other optimizations useful, it is one of the most important—if not the most important—optimizations.*

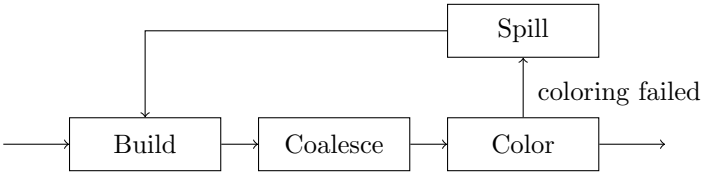
1.1 Graph-Coloring Register Allocation

The most prominent approach to register allocation probably is *graph coloring*. Thereby, interference is represented as a graph: each variable in the program corresponds to a node in the graph. Whenever two variables interfere, the respective nodes are connected by an edge. As we want to map the variables to registers, we assign registers to the nodes in the *interference graph* so that two adjacent nodes are assigned different registers.

In graph theory, such a mapping is called a coloring.[1] The major problem is that an optimal coloring, i.e. one using as few colors as possible, is generally hard to compute; it is NP-complete. Furthermore, the problems of checking a graph for its k-colorability and the problem of finding its chromatic number, i.e. the smallest number of colors needed to achieve a valid coloring of the graph, are also NP-complete. In his seminal work, Chaitin et al. [1981] showed that each graph is the interference graph of some program. Thus, register allocation is NP-complete, also.

To color the interference graph, usually a heuristic is applied. The variables which have to be spilled are determined during coloring when a node cannot be assigned a color by the heuristic. This leads to an iterative approach as shown below:

[1] The term coloring originates from the famous four color problem: Given a map, are four colors sufficient to color the countries on the map in a way that two adjacent countries have different colors? This question was positively answered in 1974 after being an open question for more than 100 years.



While being very successful in the past twenty years, this approach suffers from several inconveniences:

- Depending on the coloring heuristic used, spills may occur although the graph is colorable.
- The *coalescing* phase which is responsible for eliminating unnecessary move instructions by merging nodes in the interference graph might degrade the colorability of the interference graph. Therefore, one uses conservative coalescing methods to ensure that eliminating moves does not cause spilling. Due to the generality of the interference graphs, these algorithms have proven to be too pessimistic leaving a considerable number of uneliminated copies.
- Even if the compiler reduces the register pressure at the instructions of the program to a bearable amount, the shape of the control flow graph of the program may induce additional register demand. This additional demand is not efficiently predictable.
- Spilling is only subordinate to coloring. Since spilling is not activated unless the coloring heuristic fails, the decision which node to spill is very dependent on the way the heuristic works. This may lead to spilling decisions which improve the structure of the interference graph concerning coloring but are a bad choice regarding the program's performance.
- When spilling has taken place, the interference graph has to be re-built since the program has been modified by inserting the load and store instructions implementing the spill. Since interference graphs are commonly large and complex data structures, this is an expensive operation especially for just-in-time compilation scenarios.

1.2 SSA-based Register Allocation

Chaitin's reduction has one indispensable feature to sustain the generality of the interference graphs: a variable must be definable more than once.

This especially means that if a variable is defined multiple times, all values computed at these definitions must be written to the *same* register. While this seems to be an obvious presumption, it allows for generating arbitrary interferences and thus arbitrary interference graphs.

Having multiple definitions per variable is a serious obstacle in various compiler phases and optimizations. In many optimizations one is interested in the definition of a variable being the place where the variable is written to, i.e. defined. However, this is generally dependent on the place in the program since a variable may have multiple definitions! Think of a variable a defined in a then- and an else-clause of an if statement and asking which expression is responsible for a's value *after* the if as shown in Figure 1.1a.

(a) non-SSA (b) SSA

Figure 1.1: If-Then-Else

To remedy this problem, the *static single assignment form* (SSA form) has been invented (see Rosen et al. [1988] and Cytron et al. [1991]). The basic trick is to identify the variable with its definition. As a direct consequence, each variable is defined only once. At join points of the control flow, where the definition of a variable is not unambiguously determinable, so-called Φ-operations are placed which can be thought of as a control flow dependent move instructions. The SSA equivalent for our example is shown in Figure 1.1b. The former non-SSA variable a has been split into three SSA variables a_1, a_2, a_3. This enables the register allocator to assign different registers to a_1, a_2, a_3 instead of a single one for a. Each non-SSA program can be converted into an SSA form program by applying SSA construction algorithms such as the one by Cytron et al. [1991].

The central result of this thesis is the fact that the interference graphs of SSA form programs are *chordal.* This has a significant impact on the way register allocators can be built:

- Coloring and spilling can be completely decoupled.

- Since chordal graphs are *perfect* they inherit all properties from perfect graphs, of which the most important one is, that the chromatic number of the graph is equal to the size of the largest clique. Even stronger, this property holds for each induced subgraph of a perfect graph. In other words, chordality ensures that register pressure is not only a lower bound for the true register demand but a precise measure. Determining the instruction in the program where the most variables are live, gives the number of registers needed for a valid register allocation of the program. Unlike non-SSA programs, the structure of the control flow graph cannot cause additional register demand.

- This allows the spilling phase to exactly determine the locations in the program where variables must reside in memory. Thus, the spilling mechanism can be based on examining the instructions in the program instead of considering the nodes in the interference graph. After the spilling phase has lowered the register pressure to the given bound, it is guaranteed, that no further spill will be introduced. So spilling has to take place only *once*.

- Coloring a chordal graph can be done in $O(|V|^2)$. We will furthermore show that the order in which the nodes of the interference graph are colored is related to the order of the instructions in the program. Hence, coloring can be obtained from the program without materializing the interference graph itself.

- The major source of move instructions in a program are Φ-operations. Coalescing these copies too early might result in an unnecessarily high register demand. The coalescing phase must take care that coalescing two variables will not exceed the number of available registers. Instead of merging nodes in the interference graph, we try to assign these two nodes the same color. The graph remains chordal this way and coalescing can easily keep track of the graph's chromatic number and refuse to coalesce two variables if this would increase the chromatic number beyond the number of available registers. However, as we show in Section 4.5, this modified graph coloring problem is NP-complete.

1.3 Overview of this Thesis

Chapter 2 provides notational foundations and a more precise introduction to the register allocation problem and its components. Chapter 3 gives an overview over the state of register allocation research that is relevant and related to this thesis. Chapter 4 establishes the chordal property of the interference graphs of SSA form programs. Based on this theoretic foundation, methods and algorithms for spilling, coalescing and the treatment of register constraints are developed to exploit the benefits of SSA-based register allocation. Please note that this chapter makes intensive use of terminology of the theory of perfect graphs. (Readers not familiar with this should read Appendix A beforehand). Afterwards, an experimental evaluation of the new register allocator proposed is presented in Chapter 5. The thesis ends with final conclusions and an outlook to further research. Finally, Appendices A and B provide the terms and definitions of graph theory and integer linear programming which are needed in this thesis.

2 Foundations

In this chapter we present the formal foundations needed throughout this thesis.

2.1 Lists and Linearly Ordered Sets

In the following we often deal with lists and sets where the order of the elements matter. A *list* of length n over a carrier set X is defined as a total map $s : \{1, \ldots, n\} \to X$. The set of all lists of length n over X is denoted by $\mathscr{L}^n(X)$. Furthermore

$$\mathscr{L}(X) = \bigcup_{i \in \mathbb{N}} \mathscr{L}^i(X)$$

represents the set of all lists over X. For a list $s \in \mathscr{L}(X)$ with $s(1) = x_1, \ldots, s(n) = x_n$ we shortly write $(x_1, \ldots, x_n)$. The i-th element of a list s is denoted by $s(i)$. If there is some i for which $s(i) = x$ we also write $x \in s$.

A *linearly ordered set* is a list in which no element occurs twice, i.e. the map s is injective. The set $\mathscr{S}(X)$ of linearly ordered sets over X is thus defined as

$$\mathscr{S}(X) = \{s \mid s \in \mathscr{L}(X) \text{ and } s \text{ injective}\}$$

If a list $(x_1, \ldots, x_n)$ is also a linearly ordered set, we sometimes emphasize this property by writing $\langle x_1, \ldots, x_n \rangle$.

In the following, we often will deal with maps returning a list or a linearly ordered set. For example, let X and Y be two sets and $a : X \to \mathscr{L}(Y)$ be such a map. Instead of writing $a(x)(i)$ for accessing the i-th element of the returned list, we will write $a(x, i)$ according to the isomorphy of $A \to (B \to C)$ and $(A \times B) \to C$.

2.2 Programs

Throughout this thesis we only consider register allocation on the procedure-level. This means the input we are processing is a single procedure given by its control flow graph consisting of *labels* and instructions. To stick to established conventions we call such a unit a *program*. Since register allocation has to assign storage locations to *variables*, we do not care about the kind of computation carried out by an instruction but on which variables it reads and writes. All this is captured in a program P being a tuple

$$(V, O, L, pred, arg, res, op, start)$$

consisting of

- a set of variables V.
- a set of operations O from which the operations of the program are drawn. We will use infix notation for unary and binary operations. We require that there is a copy operation $y \leftarrow x$ which copies the value from variable x to variable y.
- a set of labels L which denote the instructions of the program.
- a function $pred : L \to \mathscr{S}(L)$ which assigns each label a linearly ordered set of *predecessor* labels. Whenever there is an $i \in \mathbb{N}$ for which $\ell' = pred(\ell, i)$, we write $\ell' \xrightarrow{i} \ell$. If i is not important in the context we write $\ell' \to \ell$. We further define $|\ell| = |pred(\ell)|$.

  The map $pred$ induces a complementary map $succ : L \to 2^L$ which is defined as $succ(\ell) = \{\ell' \mid \ell \to \ell'\}$.
- a usage function $arg : L \to \mathscr{L}(V)$ denoting which variables are read at a certain label.
- a definition function $res : L \to \mathscr{S}(V)$ expressing which variables are written by the instruction at a certain label.
- an operation assignment $op : L \to O$ reflecting the operation executed at some label.
- a distinct label $start \in L$ for which $|start| = 0$ giving the start of control flow.

For some $\ell \in L$, $op(\ell) = \tau$, $res(\ell) = (y_1, \ldots, y_m)$ and $arg(\ell) = (x_1, \ldots, x_n)$ we will write shortly

$$\ell : (y_1, \ldots, y_m) \leftarrow \tau\,(x_1, \ldots, x_n)$$

reflecting that the operation τ reads the variables $x_1, \ldots, x_n$ and writes the variables $y_1, \ldots, y_m$ at label ℓ. We say a label ℓ is a *definition* of a variable x if $x \in res(\ell)$ and ℓ is a *usage* of a variable x if $x \in arg(\ell)$, respectively.

We visualize the program by drawing its *control flow graph* (CFG). We put a label, its associated operation and variables into boxes. If there is flow of control from ℓ to ℓ' and ℓ is the i-th predecessor of ℓ', i.e. there is $\ell \xrightarrow{i} \ell'$, we draw an edge from the box of ℓ to the box of ℓ' and annotate the i at the edge's arrow if it is important in the context. Figure 2.1 shows a program in pseudo-code and its control flow graph.

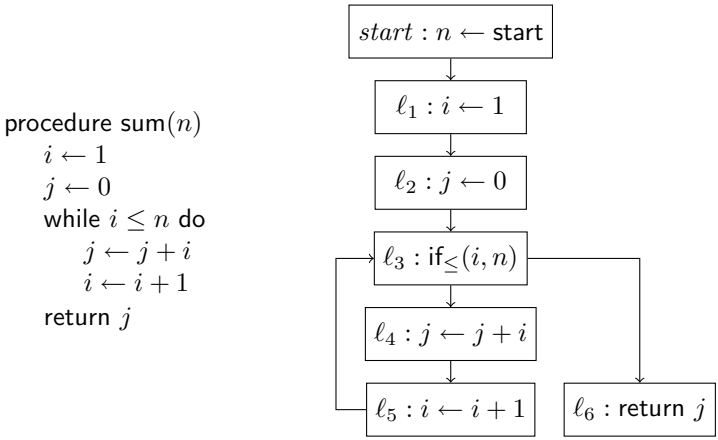


Figure 2.1: Example program P and its control flow graph

We say a linearly ordered set of labels $\langle \ell_1, \ldots, \ell_n \rangle$ is a *path* iff there is $\ell_1 \to \ell_2, \ldots, \ell_{n-1} \to \ell_n$. To indicate that there is a path from ℓ_1 to ℓ_n we write $\ell_1 \to^* \ell_n$. A path $B = \langle \ell_1, \ldots, \ell_n \rangle$ is called a *basic block* if for each ℓ_i and ℓ_j holds:

1. For each $1 \leq i < n$ holds: $succ(\ell_i) = \{\ell_{i+1}\}$ and $pred(\ell_{i+1}) = \{\ell_i\}$
2. B is maximal, i.e. it cannot be extended by further labels.

Our example program consists of four basic blocks: $B_1 = \langle start, \ell_1, \ell_2 \rangle$, $B_2 = \langle \ell_3 \rangle$, $B_3 = \langle \ell_4, \ell_5 \rangle$ and $B_4 = \langle \ell_6 \rangle$.

As control flow within basic blocks is not important for our work, we will draw all labels within the same basic block in one box and only give control flow edges leaving and entering basic blocks. The result of doing so is shown in Figure 2.2.

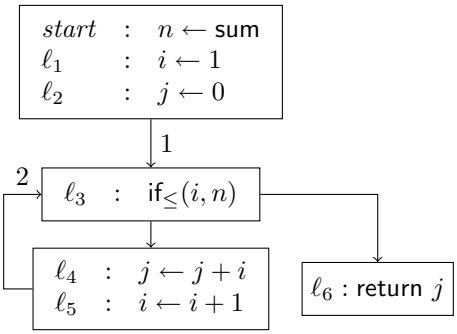


Figure 2.2: Simplified drawing using basic blocks

A control flow edge $\ell \to \ell'$ is *critical* if $|\ell'| > 1$ and there exists some $\ell'' \neq \ell'$ with $\ell \to \ell''$, i.e. the edges connects a label with multiple successors to a label with multiple predecessors.

We say a label ℓ *dominates* another label ℓ' iff each path from *start* to ℓ' also contains ℓ. We then write $\ell \preceq \ell'$. In our example in Figure 2.1, ℓ_1 dominates $\ell_2, \ldots, \ell_6$. It is clear, that *start* dominates all other labels in the CFG. Dominance is an order relation (i.e. it is reflexive, transitive and anti-symmetric), cf. Lengauer and Tarjan [1979] for example. Furthermore, dominance induces a tree order on the labels, since for $\ell_1 \preceq \ell_3$ and $\ell_2 \preceq \ell_3$ there must always be $\ell_1 \preceq \ell_2 \preceq \ell_3$ or $\ell_2 \preceq \ell_1 \preceq \ell_3$. Thus, each label ℓ has a unique label $idom(\ell) \neq \ell$, called its *immediate dominator*, for which holds: $\forall \ell' \prec \ell : \ell' \preceq idom(\ell)$. The immediate dominator of ℓ is the parent of ℓ in the CFG's dominance tree.

Following the nomenclature of Budimlić et al. [2002], we say a program is *strict* if for each variable x and each usage ℓ of x holds: Each path from *start* to ℓ contains a definition of x. If a program is non-strict due to the violation of the strictness property by a variable x we also say that the variable x is not strict.

2.3 Static Single Assignment (SSA)

We say, a program fulfils the static single assignment property (SSA-property) if each variable is statically defined once. In our setting this means that for each variable there is exactly one label where the variable is written to. We will write $\mathcal{D}_x$ for the label where x is defined. Obviously, the program in Figure 2.1 does not have the SSA-property since e.g., the variable i is defined at labels ℓ_1 and ℓ_5. If a program possesses the SSA-property, we also say the program is in SSA-form.

Programmers often use the same variable to refer to different computations which are dependent on the control flow of the program. Consider the example program in Figure 2.2. The variable i is defined twice: inside the loop and before the loop. If the example program were in SSA-form, only one definition of i would be allowed. Thus, one of the definitions has to write to another variable. Let us say that the definition inside the loop shall write to a variable i'. Now, the question is, to which definition (i or i') the usage of i inside the loop shall refer? Obviously, this is dependent on from where the loop was entered: if we came from outside we should use i, otherwise we should use i'.

This disambiguation is provided by a new operation (denoted by Φ) that selects variables dependent on the edge a basic block was reached:

$$\ell : (y_1, \ldots, y_m) \leftarrow \Phi\left(x'_1, \ldots, x'_{m \times n}\right) \quad \text{with } n = |\ell|$$

If ℓ was reached via $pred(\ell, i)$, the variables $x_{(i-1)n+1}, \ldots, x_{(i-1)n+m}$ are copied to $y_1, \ldots, y_m$ in parallel. For better comprehension, we will consider the Φ-operation as given in a matrix style notation

$$\ell : (y_1, \ldots, y_m) \leftarrow \Phi \begin{bmatrix} x_{11} & \cdots & x_{1m} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nm} \end{bmatrix}$$

where x_{ij} corresponds to $x'_{(i-1)n+j}$. Each column of the matrix corresponds to a result variable and each row of the matrix associates the variables in that column with a particular predecessor.

As the arguments of a Φ-operation are related to control flow, we reflect this in a comfortable way to describe the arguments of a Φ-operation.

$$arg(\ell, i) = \begin{cases} (x_{i1}, \ldots, x_{im}) & \text{if } op(\ell) = \Phi \\ arg(\ell) & \text{otherwise} \end{cases}$$

2.3.1 Semantics of Φ-operations

To express the semantics of the Φ-operations, we split each Φ-operation into two parts. A read part called Φ^w and as many write parts Φ^r as there are predecessors of the Φ's label. The semantics of the Φ-operation is then given by considering the program after applying the program transformation $\mathcal{T}_\Phi$.

Definition 2.1 (Transformation $\mathcal{T}_\Phi$): *Consider a label*

$$\ell : (y_1, \ldots, y_m) \leftarrow \Phi \begin{bmatrix} x_{11} & \cdots & x_{1m} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nm} \end{bmatrix}$$

For each predecessor $pred(\ell, i)$ *insert a label*

$$\ell_i : \Phi^r(x_{i1}, \ldots, x_{im})$$

splitting the edge from $pred(\ell, i)$ *to* ℓ*. Replace the* Φ*-operation in* ℓ *by*

$$\ell : (y_1, \ldots, y_m) \leftarrow \Phi^w$$

The semantics of Φ^w and Φ^r is defined as follows:

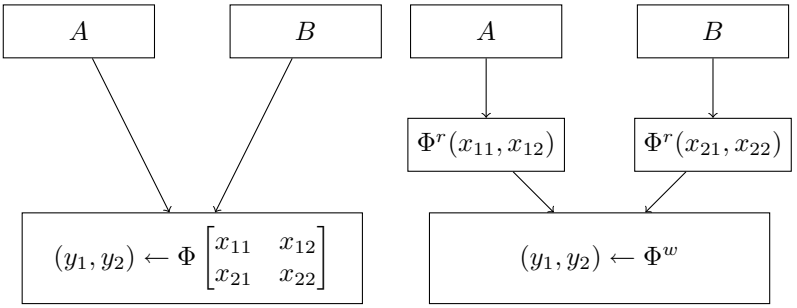
$$\begin{aligned} [\![(y_1, \ldots, y_m) \leftarrow \Phi^w]\!] &:= [\![y_1 \leftarrow \$_1; \cdots; y_n \leftarrow \$_n]\!] \\ [\![\Phi^r(x_1, \ldots, x_m)]\!] &:= [\![\$_1 \leftarrow x_1; \cdots; \$_n \leftarrow x_n]\!] \end{aligned}$$

where $\$_1, \ldots, \$_m$ are variables that did not occur in P. Note that this implies that the $\$_i$ are multiply defined and thus violate the single assignment property. However, this is no problem as they serve only for modelling the *semantics* of Φ-operations. In other words: we use a transformation of an SSA program to a non-SSA program to model the semantics of a Φ-operation.

Figure 2.3 illustrates the transformation $\mathcal{T}_\Phi$. Intuitively, Φ-operations can be seen as control-flow dependent *parallel* copies . The parallel character of the Φ evaluation is most important, since assignments can be cyclic as in

$$(y, x) \leftarrow \Phi \begin{bmatrix} x & y \\ y & x \end{bmatrix}$$

which means, that x and y are swapped when the label is reached along the edge from its first predecessor. Figure 2.4 shows the example program in SSA-form.



(a) A Φ-operation in some program P (b) $\mathcal{T}_\Phi(P)$

Figure 2.3: The program transformation $\mathcal{T}_\Phi$

Remark 2.1: The matrix-style notation of Φ-instructions used in this thesis is not common in literature. Instead of a single Φ with a vector result and a matrix of operands one finds a ϕ for each column of the matrix with the arguments separated by commas. We find this misleading for two reasons:

1. it suggests that these ϕ-instructions are executed serially which is *not* the case.

2. The commas make ϕ-instructions seem like each operand is needed to "compute" the results of the ϕ-instruction. This is also not the case since only the i-th operand is needed when the ϕ's label ℓ is reached along $pred(\ell, i)$

Figure 2.5 juxtaposes the style used in this thesis and the classical notation.

2.3.2 Non-Strict Programs and the Dominance Property

In principle, SSA-form programs can be non-strict (contain variables which are not properly initialised). That is, there is a usage ℓ of some variable x for which there is a path from *start* to ℓ which does not contain $\mathcal{D}_x$. However, almost every SSA-based algorithm only considers strict programs by relying on the dominance property:

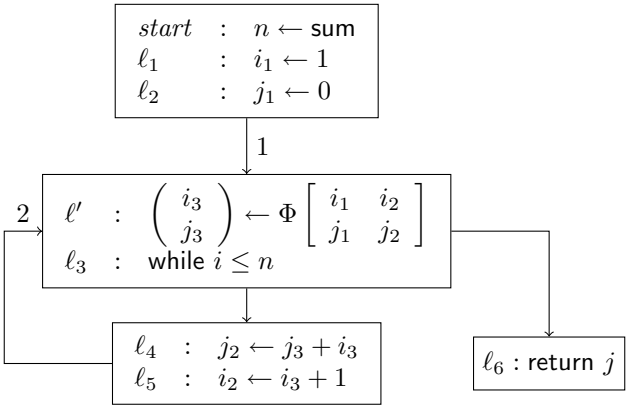


Figure 2.4: Example program P in SSA-form

$$\begin{array}{c} y_1 \leftarrow \phi(x_{11}, \ldots, x_{n1}) \\ \vdots \\ y_m \leftarrow \phi(x_{1m}, \ldots, x_{nm}) \end{array}$$

(a) Classical

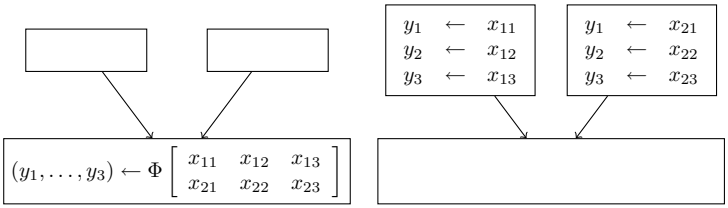
$$(y_1, \ldots, y_m) \leftarrow \Phi \begin{bmatrix} x_{11} & \cdots & x_{1m} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nm} \end{bmatrix}$$

(b) Matrix

Figure 2.5: Φ-Notations

Definition 2.2 (SSA Dominance Property): *Let x be a variable and ℓ be a label for which there is some i such that $x \in arg(\ell, i)$. If $op(\ell) = \Phi$ then $\mathcal{D}_x \preceq pred(\ell, i)$, else $\mathcal{D}_x \preceq \ell$.*

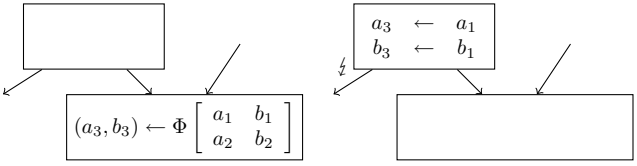
Mostly, SSA is constructed in a strict way, also if the corresponding non-SSA program was not strict. Each SSA construction algorithm needs to rewrite the uses of the labels to the correct SSA variable created for a non-SSA variable. In a strict program this poses no problem. In a non-strict program however, for every non-strict variable, there will be at least one use which is not correctly mappable to a definition. Many construction algorithms then create a new variable which is initialized to zero (or any other value) right in front of the use. While this results in a correct program, the initialization to zero will probably result superfluous code being generated. A better solution is to introduce a dummy variable `undef` which is defined right after *start* so that it dominates all the labels in the program. If a proper definition is not



(a) Program fragment

(b) Program fragment with copies implementing a Φ

Figure 2.6: SSA Destruction



(a) SSA

(b) Post-SSA

Figure 2.7: Lost Copy Problem

found for some (non-strict) use, the use is rewritten to `undef`. Of course, no code will ever be generated for the initialization of `undef`.

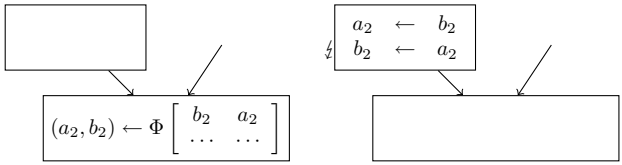
2.3.3 SSA Destruction

Since no currently available processor provides Φ-operations, a compiler must destruct the SSA-form before final machine code is emitted. Usually, Φ-operations are *implemented* by inserting a sequence of copy instructions in each predecessor block of the Φ's block. Figure 2.6 illustrates the basic principle.

However, there are two subtleties arising from the definition of the Φ-operation's semantics as stated in Definition 2.1.

Lost Copies

Consider a Φ-operation in a block B having n predecessors $B_1, \ldots, B_n$. Since the transfer action from the arguments in the Φ-operation's i-th row to the results is done *on the way* from the i-th predecessor to the Φ-operation's block, there might not be a block to which the copy instructions can be appended to. If the i-th predecessor has multiple



(a) SSA (b) Post-SSA

Figure 2.8: Φ-Swap Problem

successors, the copies related to the Φ-operation would be also executed if control is not transferred to the Φ-operation's block, i.e. the edge from B_i to B is *critical*. As the copies must be done "on the edge" from B_i to B, the critical edge must be split by inserting a block B_i' between B_i and B.

Φ-Swap

The arguments and results of Φ-operations may be cyclically interdependent as shown in Figure 2.8a. Then, implementing a Φ-operation straightforwardly using a sequence of copies leads to wrong code as shown in Figure 2.8b. The value of a_2 is destroyed after the first copy but is needed in the second one to be transferred to b_2. In this case, one must either use an instruction swapping a_2 and b_2 or introduce a third copy and rewrite the sequence to

$$\begin{array}{lcl} t & \leftarrow & a_2 \\ a_2 & \leftarrow & b_2 \\ b_2 & \leftarrow & t \end{array}$$

2.4 Global Register Allocation

Let P be a program as defined in Section 2.2. A *register allocation* $\rho : V_P \to R$ of the program P assigns elements of some (finite) set R, called registers, to variables of P. Note that most processors have multiple register classes. Commonly one performs register allocation for each class separately. A *valid* register allocation must take the *structure* of the program into account: it is intuitively clear that some variables x and y cannot be assigned the same register if a definition of x would overwrite the value of y from the register although y's value might still be needed "later on". We then say x and y

interfere. As interference is a binary relation on the program's variables, we will often talk about the *interference graph (IG)* $\mathcal{I}_P = (V_P, E)$ where $E = \{xy \mid x \text{ and } y \text{ interfere}\}$. By requiring that a valid register allocation ρ never assigns two interfering variables to the same register, ρ is a *coloring* of the IG. Figure 2.9 shows the IG of the program shown in Figure 2.1.

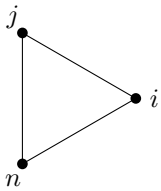


Figure 2.9: Interference graph of the example program

2.4.1 Interference

Interference is commonly (but not necessarily only) defined using the notion of *liveness*:

Definition 2.3 (Liveness): *A variable x is live at some label ℓ if there is a path from ℓ to some usage ℓ' of x and the path does not contain a definition of x.*

We call the set of labels where a variable x is live the *live range* of x. The number of variables live at some label ℓ is called the *register pressure* at ℓ. Liveness is usually computed by solving a pair of dataflow equations

$$\begin{aligned} \mathit{liveout}(\ell) &= \bigcup_{\ell' \in \mathit{succ}(\ell)} \mathit{livein}(\ell') \\ \mathit{livein}(\ell) &= [\mathit{liveout}(\ell) \setminus \mathit{res}(\ell)] \cup \mathit{arg}(\ell) \end{aligned}$$

using a standard worklist algorithm (cf. Nielson et al. [1999] for example). If we say x is live at ℓ we mean that $x \in \mathit{livein}(\ell)$.

Straightforwardly, this gives the standard definition of *interference*:

Definition 2.4 (Interference): *Two variables x and y interfere if and only if there is a label where both are live.*

However, this definition only regards the structure of the program. Structural interference of two variables can be disregarded if x and y contain the

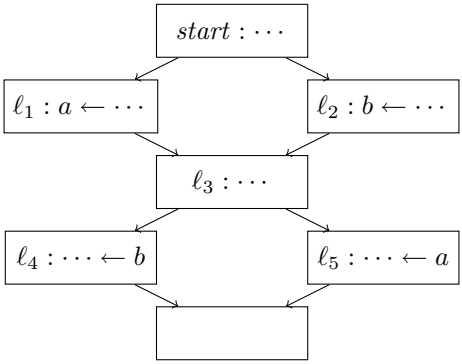


Figure 2.10: Non-strict program

same value at all labels where they are simultaneously live, since as they contain the same value, they can share their storage location. This becomes important if one considers non-strict programs. If a program is not strict, there may be a path from *start* to the usage of some variable x which does not contain a definition of x. The content of x on this path is arbitrary, so it can be considered equal to any other value. Thus, x will not interfere with any other variable y on the path to ℓ although x and y may be simultaneously live on the path. Consider the program fragment in Figure 2.10. a and b are both non-strict variables and are both live at the label ℓ_3. According to Definition 2.4 they interfere. However, if we reach ℓ_3 either the value of a or b is arbitrary depending on how we reached ℓ_3 (through ℓ_1 or ℓ_2). Thus we could assign the same register to a and b without compromising validity. Conclusion: defining interference by liveness is only an approximation and may be refined through a more precise analysis of the program.

2.4.2 Coalescing and Live Range Splitting

The shape of the IG can be modified by *splitting* live ranges. To split the live range of a variable x, a copy $x' \leftarrow x$ is inserted at an appropriate label ℓ in the program where x is live. All successive usages of x then get rewritten to use x' instead of x. Accordingly, the IG of the program obtains a new node x' and a part of the edges incident to the node of x are moved to the node of x'. Thus, live range splitting can lower the degree of the node whose live range is split.

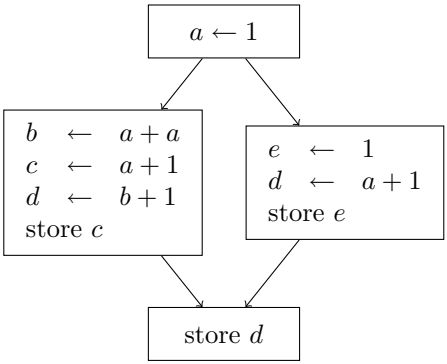
Figure 2.11 shows an example program P', its IG and a program P'' which resulted from P' by splitting the live range of variable d before the last basic block. $\mathcal{I}_{P''}$ has a lower chromatic number (2) than $\mathcal{I}_{P'}$ whose chromatic number is 3. Thus, live range splitting can have a positive effect on the colorability of the interference graph. Consider the extreme case where the live ranges of all live variables are split in front and behind each label. Then, the IG degenerates into a set of independent cliques where each clique consists of the nodes corresponding to the variables live at that label. Each of these cliques is trivially colorable using exactly as many colors as the size of the clique.

Note that constructing the SSA form breaks live ranges since Φ-operations represent (parallel) copy instructions. The effect of this particular splitting on the IG of the program will be thoroughly discussed in Chapter 4. Live range splitting obviously can improve the colorability of the IG, but it introduces an extensive amount of shuffle code, mainly copy instructions that transport the contents of the split variable to new variables. As one is usually not willing to accept an arbitrary amount of inserted shuffle code, the register allocator tries to remove as many of the inserted copies as possible. This technique is known as *coalescing*. The art of coalescing is to remove as many moves as possible without pushing the chromatic number of the IG over the number of register available.

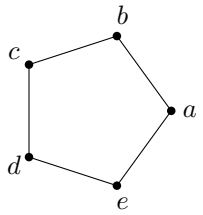
To express the information of variables emerging from split live ranges, we extend the IG of a program P with a set $A \subseteq [V_P]^2$ of *affinity edges* (see Appendix A for notation): $xy \in A$ indicates that x and y are involved in a copy instruction and assigning x and y the same color or merging them into one node will save a copy instruction. Furthermore, we also equip the IG with a cost function $c : A \to \mathbb{N}$ to weight the affinity edges arbitrarily . Thus, the IG will from now on be a quadruple $\mathcal{I}_P = (V_P, E, A, c)$. In drawings of the IG such as in Figure 2.11d, we indicate affinity edges with dotted edges and costs superscripted.

2.4.3 Spilling

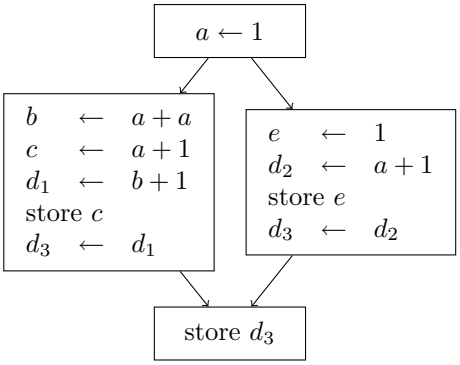
Not every program P has a valid register allocation. This is exactly the case if the chromatic number of the interference graph is larger than the number of available registers: $\chi(\mathcal{I}_P) > |R| = k$. In order to make $\mathcal{I}_P$ k-colorable, the program must be transformed. This is achieved by writing the values of some set of variables to memory at some points of the program. By inserting load and store instructions for a variable x, the live range of x is fragmented. This lowers the register pressure at several other labels. The quality of the



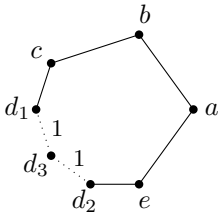
(a) Example Program P'



(b) IG of P'



(c) P'': splitting live ranges in P'



(d) IG of P''

Figure 2.11: Effects of live range splitting on the IG

spilling process is crucial for the performance of the program, since the time for accessing a memory location can exceed the time for accessing a register by at least one order of magnitude in today's processor architectures. An alternative to storing and loading a variable is to recompute it. This is commonly known as *rematerialisation*. Of course, all operands of the instruction writing to the variable must be live at the point of rematerialisation. Thus, this technique is useful for operations with few or even no operands, such as constants.

2.4.4 Register Targeting

Up to now we assumed that each register is assignable to every variable. When compiling for real processor architectures this is rarely the case. Most processors impose register constraints on several of their instructions. This may even lead to unallocatable programs in the first place. Assume two variables x and y being defined by some instruction i at labels ℓ_1 and ℓ_2 respectively. The instruction i has the constraint that its result is always written to a specific register. If x and y interfere, the interference graph of the program does not have a valid coloring since two interfering variables must be assigned the same register. Generally, one splits the live ranges of constrained variables at the labels where the instruction imposing the constraint is acting on the variable. This allows for moving the variable from/to the constrained register and place it in an arbitrary register. Common constraints of current processor architectures *and* runtime systems are:

- An instruction requires an operand or a result to reside in a specific subset of registers. Mostly, this subset is of size 1. We write $x|_S$ to express that x's value has to be in one of the registers of the set S at its occurrence.
- The instruction requires an operand to have the same register as a result. This is commonly the case for two-address-code machines like x86.
- An instruction needs a result register to be different from some operand register. This situation also occurs with two-address-code machines and some RISC architectures like the ARM.

3 State of the Art

The major distinction between register allocation approaches is *global* versus *local* register allocation. Global register allocation works at the procedure level, whereas local register allocation only considers basic blocks or only single statements and their expression trees. Local register allocation is far less complex since it avoids all subtleties which arise from control flow split- and merge points. The drawback however is, that the local allocations have to be combined across basic block borders. Very simple approaches store all variables live at the end of a basic block to memory and reload all variables live at the entrance to the successor. This approach is nowadays considered unacceptable due to significant performance drawbacks arising from the huge amount of memory traffic on block borders.

This problem becomes even worse when register allocation only considers a single expression tree. Although an optimal allocation for the tree can be obtained efficiently (cf. Sethi and Ullman [1970]) for standard register files, combining the allocation for each expression to acceptable fast code is difficult. This problem is attacked by *global register allocation* which performs the allocation on the procedure level. Some compilers rely on a mixed local/global allocation strategy reserving some registers for a local allocator inside the instruction selector and using a global allocator to assign registers to variables living over multiple basic blocks.

Although global register allocation eliminates inter-block fixup-code, the problem is algorithmically complex. Furthermore, common global register allocators as presented in this chapter rely on heavy-weight data structures that need careful engineering.

The register allocation approach presented in this thesis certainly belongs to the "global" category. Thus, we will focus on global register allocation techniques and mention other approaches along the way. The most prominent technique for global register allocation is *graph coloring* where the interference graph is materialized as a data structure and processed using coloring algorithms.

3.1 Graph-Coloring Register Allocation

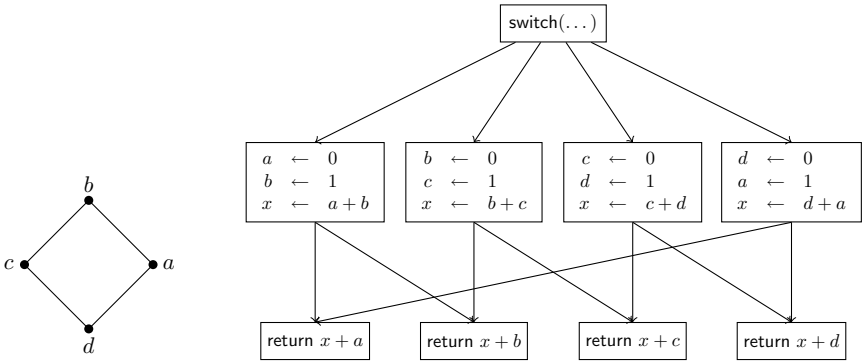
Graph coloring register allocation (GCRA) goes back to the 1960s where Ershov [1965] used "conflict matrices" to record interferences between variables in a basic block. These matrices are basically the adjacency matrices of the interference graph. The breakthrough of GCRA is marked by the seminal work of Chaitin et al. [1981] rediscovering a coloring scheme by Kempe [1879]. Chaitin et al. proved that each undirected graph is the interference graph of some program. This assertion is very strong since it implies the NP-completeness of GCRA via the NP-completeness of general graph coloring (cf. [Garey and Johnson, 1979, Problem GT4]).

Let us revisit Chaitin's proof since we will use a similar technique for our purposes in Section 4.5. Consider an arbitrary, undirected graph $G = (V, E)$. Firstly, we augment V with an additional node, say x, giving V'. We insert edges from x to all nodes in V resulting in a new edge set E'. An optimal coloring for $G' = (V', E')$ can be easily turned into a coloring for G by simply deleting x. Since x is adjacent to all nodes in V, its color is different from the colors assigned to the nodes in V. Now, a program is constructed from G' in the following way:

1. Add a start basic block S.
2. For each $v \in V'$ add a basic block B_v containing the instruction return $x+v$
3. For each edge $e = ab \in E'$ add a basic block B_{ab} containing the following instruction sequence:
$$\begin{array}{lcl} a & \leftarrow & 1 \\ b & \leftarrow & 0 \\ x & \leftarrow & a + b \end{array}$$
4. Add control flow edges from S to B_{ab}, from B_{ab} to B_a and B_b.

Consider figure 3.1. It shows the graph C^4 and the program which is constructed following the rules above. A valid register allocation for the program in figure 3.1b can be transformed into a valid coloring of the graph in figure 3.1a simply by deleting the node for x and all its incident edges from the interference graph of the program. Thus, not only the allocation problem but also the problems

1. How many registers will be needed for a particular program?
2. Are k registers sufficient for a register allocation of a program?



(a) C^4 (b) Program for C^4

Figure 3.1: A graph and the program generated from it

are NP-complete in the number of variables in the program.

Hence, a heuristic which mostly all graph coloring register allocators have in common is applied: Firstly, unnecessary copies are removed in the coalescing phase. This is done by merging the nodes of the argument and the result of a copy instruction into a single node if they do not interfere. This is performed aggressively, i.e. for as many copy instructions as possible. Then coloring is attempted in two phases:

simplify
: searches for a node that has fewer than k neighbors. If such a node is found, it is removed from the graph and pushed onto a coloring stack. If no such node is found, a node is selected for spilling, spill code is inserted, the IG is rebuilt and coloring starts over again.

select
: If the simplify-phase managed to remove all nodes from the graph, the nodes are popped from the coloring stack one by one and re-inserted into the graph. This produces a valid coloring, since each node has no more than $k-1$ neighbors as it is re-inserted into the graph. Thus there will be always a free color to color the node.

This iterative approach, as shown in Figure 3.1, is typical for more or less all GCRA approaches.

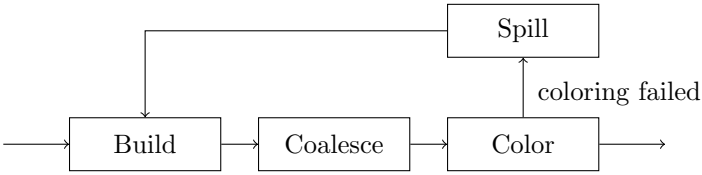


Figure 3.2: General scheme of Chaitin-style GCRA

3.1.1 Extensions to the Chaitin-Allocator

Chaitin's approach has been subject to many improvements in the last decades. However, the basic scheme, as depicted in Figure 3.1, always stays the same. The main improvements deal with better coalescing heuristics as the removal of copy instructions became more and more important as compilers started using SSA and register allocators incorporated live range splitting.

3.1.1.1 The Briggs Allocator

A drawback in Chaitin's approach is the *pessimism* in coloring. The nodes which have to be spilled are selected while removing the nodes from the graph (this phase is called simplify above). Thereby it is implicitly assumed that a node with k or more neighbors upon removal cannot be colored when the variables are re-inserted into the graph. It is *pessimistically* assumed that all neighbors of the node have different colors. However, for many graphs this leads to unnecessary spills like the graph C^4 which is shown figure 3.1a. Chaitin's heuristic is not able to color C^4 with two colors.

This inability was corrected by Briggs et al. [1994]. They delay the spilling decision to the reinsertion phase and take the coloring of the already re-inserted neighbors of a node into account. Thus, their approach is called *optimistic coloring*. In fact, their approach is able to color C^4 with two colors.

A further improvement by Briggs et al. is on coalescing: Chaitin's original method aggressively merged each (non-interfering) copy-related node pair into a single node. Since that merged node likely has a higher degree, the colorability of the graph is probably affected which might cause additional spills as Chaitin's method is not able to revoke coalescing. Briggs noticed that the number of additional spills introduced by aggressive coalescing is significant. Therefore, he proposed *conservative coalescing* which only coalesces

two nodes if the colorability of the graph is not affected. Thus, coalescing two nodes conservatively will never cause an additional spill.

More precisely, two nodes a and b are conservatively coalesceable if the node c, resulting from merging a and b, will have fewer than k neighbors of significant degree. The degree of a node is *significant* if it is larger than or equal to k. The reason for this is that all neighbors of c *not* of significant degree can be removed before c is removed. Then, c surely can be removed since there are at most $k-1$ nodes which could not have been removed yet.

3.1.1.2 Iterated Register Coalescing

George and Appel [1996] extended Briggs' approach by firstly sharpening the conservative coalescing criterion and secondly interleaving simplification with coalescing. Consider a pair of copy-related nodes a, b in the IG of some program. The nodes a and b can be coalesced, if each neighbor t of a either already interferes with b, or t is of insignificant degree. For a proof consult Appel and Palsberg [2002] for example.

Using this criterion, George and Appel [1996] modified the coloring scheme of Briggs by interleaving coalescing and simplification in the following way:

Simplify
: Remove non-copy-related nodes with insignificant degree from the graph

Coalesce
: Perform conservative coalescing on the graph resulting from simplification. The hope is that simplification already decreased the degrees of many nodes so that it is more likely that the criterion discussed above matches. When there remains no pair of nodes to coalesce, simplification is tried again.

Freeze
: If the simplification/coalesce cycle could not eliminate/coalesce nodes from the graph, all nodes of insignificant degree which are still involved in copies are marked as *not* copy-related which excludes them from further coalescing attempts. These copies will be present in the resulting program. If a node was frozen, simplify/coalesce is resumed. Else, the allocator continues as the one of Briggs.

3.1.1.3 Optimistic Coalescing

Recognizing that conservative coalescing removes too few copies, Park and Moon [2004] revived the aggressive coalescing scheme as used in Chaitin's

original allocator. However, their allocator is able to undo the coalescing decision during the select phase. Assume a node a resulted from coalescing several nodes $a_1, \ldots, a_n$. If a cannot be assigned a color in the select phase, the coalescing is undone. Then each of the a_i is inspected if it can be colored. If not, it is selected for spilling. For the remaining set A of not spilled nodes, each combination of subsets of A is checked if it can be colored with the *same* color. Each such subset S is rated by the spill costs all nodes in $A \setminus S$ would cause. The subset causing the least spill costs is then chosen and colored. The remaining nodes are merged again and placed at the bottom of the coloring stack.

3.1.2 Splitting-Based Approaches

As shown in Section 2.4.2, splitting live ranges can have a positive impact on the colorability of the IG. This was firstly noticed by Fabri [1979] in a more general paper about storage optimization. Since then, there have been many attempts to integrate live range splitting in existing register allocators or build new algorithms having a live range splitting mechanism built-in.

3.1.2.1 Priority-Based Register Allocation

Based on Fabri's observations, Chow and Hennessy [1984, 1990] presented a graph coloring approach that differs from the Chaitin-style allocators in various ways:

1. Register allocation is performed only on a subset of the registers. A part of the register file are reserved for the allocation process inside the instruction selector. Thus, this approach can be considered as a mixed local/global method.

2. Instead of considering the definitions and usages of variables at the instruction level, basic blocks are used as the basic unit of allocation. A basic block is considered a single instruction and each variable defined/used inside the block is automatically used/defined *by* the block. This leads to significant smaller IGs, but it disallows a register to hold different quantities inside a single basic blocks which restricts the freedom of allocation. To ease this problem, basic blocks are split after a certain number of instructions.

3. Live ranges whose nodes are of insignificant degree are removed from the graph before starting. Thus, the algorithm only works on the critical part of the graph.

4. Coloring does not follow the simplify/select-scheme but a priority-based scheme. Live ranges with higher priorities get colored earlier. The priority represents the savings from assigning a live range to a register instead of keeping it in memory. If a live range cannot be colored it is attempted to split it. The basic block containing a definition of the live range is inspected for a free register to contain a portion of the live range. If there is a free register to contain the split live range, the successor blocks are tried in breadth-first order.

3.1.2.2 Live Range Splitting in Chaitin-style Allocators

For Chaitin-style allocators live range splitting has a significant impact. As the coloring heuristic is very dependent on the degrees of the nodes in the IG, reducing the degree of a node generally results in better colorability. Reconsider the extreme example of Section 2.4.2. If the IG is decomposed into cliques *and* is k-colorable, i.e. there is no clique larger than k, each node has at most $k-1$ neighbors. Thus, all nodes have insignificant degree and can be eliminated by the simplify-phase.

Briggs [1992] intensively experimented with splitting live ranges before starting the allocation. In his PhD thesis, he mentions several split paradigms. Amongst others, he used the split points caused by Φ-operations which were left over after destructing SSA. His results were mixed. As colorability of the IG improved, his coalescing method was less able to remove a satisfactory number of copies. Recall that in his conservative coalescing scheme, two nodes were only coalesced if the resulting node will have fewer than k neighbors of significant degree.

3.1.3 Region-Based Approaches

Region-based register allocation performs the allocation on smaller parts, so-called regions of the program and combines the partial results to a single one for the program. Considering regions enables the allocator to invest more efforts in special parts of the program such as loops. A major drawback in the general graph-coloring approaches is that the structure of the program is not well represented by the interference graph: a node in the IG representing a "hot" variable inside a loop is hardly discriminable from some temporary which is used only once outside all the loops. Region based approaches mainly differ in the way they define a region.

Callahan and Koblenz [1991] compute a tile tree over the CFG that is similar to the CFG part of the abstract syntax tree. The tiles are colored

separately using the standard simplify/select scheme in a bottom-up pass over the tile tree. A second pass combines the results of the tile colorings into a valid coloring for the whole program. They are furthermore able to select good locations for spills. By moving them up/down the tile tree, they are able to move them out of loops or inside branches. Norris and Pollock [1994] use the program dependence graph (PDG) mainly because their compiler infrastructure is based on the PDG.

Lueh et al. [2000] provide a more general approach to region based register allocation. First, each region is colored separately. If the IG for a region is not simplifiable, a set of *transparent* live ranges (i.e. live ranges whose variables do not have any definition or use inside the region) is selected for spilling. The spill is however delayed and performed in a later phase called *graph fusion*: The IGs of regions are fused along the control flow edges connecting the regions. If the fused IG is not simplifiable live range splitting is attempted on the live ranges spanning the control flow edge on which fusion is attempted. In the worst case, all live ranges are split which corresponds to not fusing both IGs. As a nice side-effect, delayed spilling regards the split live ranges and is able to spill portions of live ranges that do not contain definitions or usages of the live range's variable.

3.1.4 Other Graph-Coloring Approaches

In the last years, the examination of the graph-theoretical structure of interference graphs has become of more interest in research. Andersson [2003] investigated more than 20000 IGs from existing compilers especially from the *Optimal Coalescing Challenge* (see Appel and George [2000] for 1-perfectness (see Section A.2) and found that all graphs he checked were 1-perfect.

Pereira and Palsberg [2005] went further and examined the interference graphs of the Java 1.4 standard library compiled by the JoeQ compiler after SSA destruction and found that 95% of them were chordal. Based on this observation, they applied standard *optimal* coloring techniques for chordal graphs like *maximum cardinality search* as described in Golumbic [1980]. Since these coloring methods also work on non-chordal graph, although non-optimally, their allocator does not have to use the simplify/select mechanism. They furthermore present graph-based spilling and coalescing heuristics which allow them to get rid of the iterative approach commonly found in GCRA: after spilling, coloring and coalescing the allocation is finished. This allows for very fast allocation, since the IG has to be built only once.

Independently and simultaneously, Brisk et al. [2005], Bouchez et al. [2005] and ourselves (see Hack [2005]) proved that the interference graphs of SSA-

form programs are chordal. Brisk et al. are pursuing the application of SSA-based register allocation to hardware synthesis where SSA never has to be destructed since Φ-operations are implementable with multiplexers. Bouchez et al. give a thorough complexity analysis of the spilling problem for SSA-form programs, showing the NP-completeness of several problem variants. Our consequence of the fact that SSA IGs are chordal is presented in this thesis. Parts of this thesis have been published by the author and others in Hack et al. [2005], Hack et al. [2006] and Hack and Goos [2006].

3.1.5 Practical Considerations

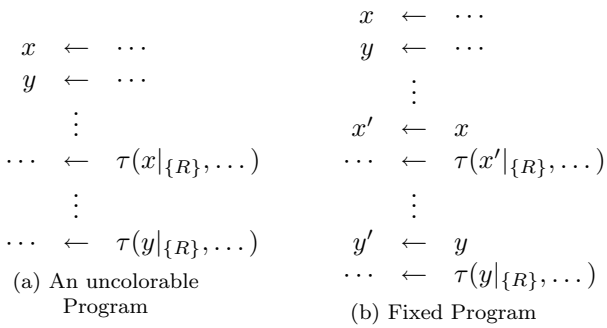
3.1.5.1 Register Targeting in Chaitin-style Allocators

A common practice in research is to assume that each variable can be assigned every register. For real-life processor architectures and runtime systems this assumption is not true. Both constrain the set of assignable registers available to the variables of the program. It might even occur that these constraints lead to an unallocatable program if a standard GCRA is applied without special preparation. This is the case if two interfering variables must reside in the same register due to targeting constraints. Consider Figure 3.3a. The variables x and y must be both assigned to register R to fulfill the constraints imposed by τ. Since both interfere, they cannot reside in the same register and the constraints are not fulfillable. Instead of constraining x and y one inserts copies of both and rewrites the usages accordingly. Then, the constrained live range is reduced to its minimum size and the interference of constrained live ranges is removed. A similar problem arises if the same variable is used at least twice in the presence of disjunct constraints. Commonly GCRAs insert copies behind/before each constrained definition/usage of a variable.

To express the register restriction, the IG is commonly enriched with pre-colored nodes, one for each register. If a variable cannot be assigned to register R_i, an edge from v's node to R_i's node is inserted. As a lot of constraints are of the type: instruction I writes its result always in register R_i thus variable v, as being the result of I, is pinned to register R_i. This introduces edges from v's node to all register nodes except the one of R_i.

Smith et al. [2004] propose an approach modifying the simplification criterion of the Chaitin/Briggs-allocators to handle aliased registers[1] as they occur in several architectures.

[1] Some parts of a register can be accessed under different names. For example, bits 0-7 and 8-15 of the 32-bit register `eax` on the x86-Architecture can be accessed with registers `al` and `ah` respectively.



(a) An uncolorable Program

(b) Fixed Program

Figure 3.3: A program uncolorable due to register constraints

3.1.5.2 Pre-Allocation Spilling

The spilling approach of Chaitin-style allocators is very crude since the spilling decisions are only taken based on the coloring heuristic becoming stuck. As a node in the IG corresponds to a live range and the whole live range is selected to be spilled, *all* uses and definitions are rewritten using loads and stores. However, it is not acceptable in practice to have a whole live range with five uses inside a loop spilled because of one use in a non-loop region of a program where the register pressure is too high. Region-based approaches, as presented in the last paragraphs, try to attack this problem. In practice however, one often uses a spilling phase *before* starting the allocation (see Paleczny et al. [2001] or Morgan [1998]).

This re-allocation spilling phase lowers the register pressure at each label to the number of available registers. This will *not* guarantee the allocation to succeed without spilling but does the major amount of spilling in advance. All spills in the allocator are then due to control flow effects or the imperfection of the coloring heuristic. The pre-allocation spiller can be program sensitive. Morgan [1998] depicts the following procedure:

1. Before spilling some variable, determine all loops where register pressure is too high and select variables live through these loops without being used inside of them to be spilled around the loop.

2. If this did not lower the register pressure to k everywhere, perform Belady's algorithm (see Belady [1966]) on the basic blocks where the

register pressure is still too high. This algorithm was originally developed for paging strategies in operating systems, selecting those pages to be removed from main memory whose next use was furthest in the future. In terms of register allocation this means that if all registers are occupied and a register is needed for a result or argument of a label, all variables currently in registers are examined and the one with the furthest use is selected to be spilled.

This has been transferred to register allocation: if a variable has to be spilled, the one whose next use is furthest away is taken.

3. Belady's algorithm will incur fixup loads and stores on the incoming control flow edges of the block it is performed on. Attempts are made to move these loads and stores to better places using techniques similar to partial redundancy elimination.

3.1.5.3 Storing the Interference Graph

All graph coloring approaches presented above need the IG as a data structure materialized in memory. As interference graphs can have more than 10000 nodes and are *not* sparse, they consume a significant amount of memory and are time-consuming to build. Unfortunately, GCRA approaches need to perform neighbor checks (is a a neighbor of b) *and* the iteration over all neighbors of a node. So, an adjacency matrix or adjacency lists would be desirable to perform both operations in acceptable time. Cooper et al. [1998] discuss engineering aspects of IGs in greater detail.

3.2 Other Global Approaches

Graph coloring has not been the only attempt to global register allocation. The costly iterative algorithm of the Chaitin-style allocators with the huge IG as a data structure make graph coloring a technique which is not trivial to implement, requires careful engineering and consumes a lot of compile time. Especially for the field of Just-In-Time (JIT) compilation where compile time matters, register allocation algorithms with emphasis on fast compile time have been developed. They mostly work linearly in the size of the program and usually do not achieve the same quality as graph-coloring register allocators. Besides these fast algorithms, there are approaches which focus on obtaining optimal solutions for the register allcoation problem using integer

linear programming. Here, we shortly outline representative approaches for both fields.

3.2.1 Linear-Scan

The linear-scan approach (cf. Poletto and Sarkar [1999]) aims at fast compile times which are of great importance in the field of JIT compilation. Unlike the other approaches presented on the last pages, the linear-scan algorithm does not consider the control flow graph and precise liveness information during register allocation. It assumes that there exists a *total* order $<$ of the instructions. Using this total order, each variable has exactly one live interval $[i, j]$ with $i < j$. These intervals can be computed by a single pass over program. However, since the instructions must be ordered linearly the live intervals may cover instructions where the variable is *not* live according to liveness analysis (as presented in Definition 2.3).

Then, the intervals are sorted increasingly in order of their start points. In a second pass, registers are assigned first come first served to the intervals. When there is no register left to assign, one of the currently "active" intervals has to be spilled. Different heuristics are applicable there. The one chosen by Poletto and Sarkar [1999] is to spill the interval whose end point is furthest away which is similar to the heuristic presented in Section 3.1.5.2.

Linear-scan is very popular in JIT-compilers mainly because it runs fast and is easy to implement. Due to its simplicity and the imprecise liveness information the quality of the allocation is considered inferior to the one produced by GCRA. In the last years, several improvements were proposed to cope with the deficiencies of imprecise liveness information and incorporating other register allocation tasks such as constraint handling and coalescing. For more details, see Traub et al. [1998], Wimmer and Mössenböck [2005] or Mössenböck and Pfeiffer [2002].

3.2.2 ILP-based Approaches

Goodwin and Wilken [1996], Fu and Wilken [2002] were the first to give an ILP (integer linear programming) formulation of the whole register allocation problem including spilling, rematerialisation, live range splitting, coalescing and register targeting. Their measurements show that it takes about five seconds for 97% of the SPEC92 benchmark suite to be within 1% of the optimal solution. They furthermore present a hybrid framework where critical control flow paths can be allocated with the ILP mechanism and the rest is processed with a standard graph coloring approach.

Appel and George [2001] focus on spilling and separate spilling and register assignment. First, ILP-based spilling inserts spills and reloads and lowers the register pressure to the number of available registers. To obtain a valid allocation, live ranges are split in front of each label as discussed in Section 2.4.2 by inserting parallel copy instructions. Thus, the IG degenerates into a set of cliques which is trivially colorable. Now, it is up to the coalescing phase to remove as many copies as necessary. Appel and George state this problem as finding a coloring of the IG which has minimal copy costs and name it *Optimal Register Coalescing*. In their experiments, they first tried iterated register coalescing which failed to remove a satisfactory number of copies due to its conservatism. Second, they applied optimistic register coalescing which produced satisfactory results concerning the runtime of the program.

3.3 SSA Destruction

Although not being a register allocation approach SSA destruction has a significant influence on register allocation. It is common sense to destruct SSA before performing register allocation. This is mostly due to a historical reason: graph coloring register allocation was invented earlier than SSA. To make SSA-based compilers "compatible" with existent (non-SSA) backends, SSA has to be destructed before the code generation process starts.

As discussed in Section 2.4.2, Φ-operations induce a live range splitting on the program. Recent SSA destruction approaches which go beyond the simple copy insertion scheme as presented in Section 2.3.3, try to subsume as many variables of the same Φ-congruence class in the same non-SSA variable (see Sreedhar et al. [1999] and Rastello et al. [2004]). Thereby, these approaches perform a *coalescing* on SSA variables before register allocation and might accidentally increase the chromatic number of the graph as shown in Figure 2.11. The fact that many variables were merged into one is invisible to the register allocator afterwards. It thus will be unable to undo this *pre-allocation coalescing*. To our knowledge, there is no register pressure sensitive SSA destruction approach.

3.4 Conclusions

We presented an overview over existing approaches in global register allocation which are relevant to this thesis. The salient approach is graph coloring as invented by Chaitin and improved by Briggs. The algorithm is dominated

by the generality of the occurring interference graphs; as shown in Section 3.1, each undirected graph can occur as an IG. Thus, the coloring scheme is designed to handle arbitrary graphs. However, the simplify/select-scheme favors graphs which contain some nodes of insignificant degree and eliminating a node causes more and more nodes to become eliminable.

We have seen that live range splitting is important since it renders the IG better colorable by the simplify/select-scheme at the costs of introducing copy instructions which have to be eliminated using coalescing. As many compilers use the SSA-form at least in some optimizations, the copy instructions introduced by the SSA destruction are still present in the register allocation phase. This led to an increasing importance of coalescing which culminated in the optimistic approach by Park and Moon.

4 SSA Register Allocation

4.1 Liveness, Interference and SSA

In this section, we investigate the properties of interference in *strict* SSA-form programs. We show that, using liveness-based interference (as given in Definition 2.4), the interference graphs of SSA-form programs are chordal. Before turning to the discussion of liveness and interference, let us start with a remark on the special variable `undef`.

A Short Note on `undef`

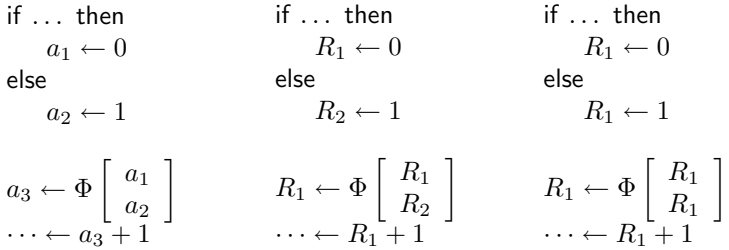
In Section 2.3.2, we introduced the dummy variable `undef` as a means to construct *strict* SSA-form programs. At each use of `undef`, its value is undefined. Hence, its value can be read from an arbitrary register, i.e. it does not interfere with any other variable, although it might be simultaneously live. As were are interested in keeping the liveness-based notion of interference, we will exclude `undef` from the set of variables on which register allocation is executed. Independently from the register allocation, we can assign an arbitrary register to `undef`.

4.1.1 Liveness and Φ-operations

The standard definition of liveness, as given in Definition 2.3, is based on the notion of usage. For ordinary operations, a variable x is used at a label ℓ if $x \in arg(\ell)$. However, this is not true for Φ-operations: control flow selects a single row out of the Φ-matrix. The variables of this row are copied to the result variables and the rest of the operands are ignored.

Applying the liveness-based definition of interference and treating Φ-operations like ordinary operations would directly lead to mutual interference

of variables in the Φ-matrix. Thus, two variables in different rows of a Φ-matrix could never share the same register. This is a valid definition which does not lead to wrong register allocations. However, it is more restrictive than reality. Figure 4.1 shows an SSA-form program and two valid register allocations. In Figure 4.1b the standard liveness definition is applied, hence a_1 and a_2 interfere. The register allocation in Figure 4.1c is valid although two interfering (in the sense of Definition 2.4) variables are assigned the same register. This is even the most desirable allocation since the Φ-operation then degrades to a no-operation.



(a) SSA (b) Naïve liveness definition (c) Realistic liveness definition

Figure 4.1: Liveness and Φ-operations

To reflect the control flow dependent behavior of Φ-operations, they have to be specially modeled concerning liveness. We express the liveness of SSA-form programs using the program transformation $\mathcal{T}_\Phi$ used to define the semantics of Φ-operations (see Definition 2.1).

Definition 4.1 (SSA-Liveness): *A variable x is live at ℓ in P if x is live at ℓ in $\mathcal{T}_\Phi(P)$.*

Corollary 4.1: *The interference graphs of P and $\mathcal{T}_\Phi(P)$ are isomorphic.*

Although being not defined in the above way, the liveness of Φ-operations is equivalently modelled by the SSA-tailored liveness analysis algorithm given in [Appel and Palsberg, 2002, page 429] and reflected in the dataflow equations given by Brisk et al. [2005].

4.1.2 Interference

Based on the liveness-based definition of interference (Definition 2.3) in SSA-form programs, Budimlić et al. [2002] gave three lemmas (4.2–4.4) relating liveness and interference to the dominance relation of the SSA form. We present these lemmas including the proofs since their understanding is vital for the rest of this thesis. Based on these lemmas, we prove that the interference graphs of strict SSA-form programs are chordal. Therefore, we consider the transformed program $\mathcal{T}_\Phi(P)$ of a strict SSA-form program P. A summary of the graph theory needed for this section can be found in Appendix A.

Lemma 4.1: *In $\mathcal{T}_\Phi(P)$ each use of a variable v is dominated by $\mathcal{D}_v$.*

Proof. For labels with non-Φ operations this is given by the SSA dominance property (Definition 2.2). Consider a strict program P, a label ℓ with $op(\ell) = \Phi$ and a variable v with $v \in arg(\ell, i)$. Due to the SSA dominance property $\mathcal{D}_v \preceq pred(\ell, i)$ holds. As the Φ^r-operation using v is placed directly after $pred(\ell, i)$ it is also dominated by $\mathcal{D}_v$. □

Lemma 4.2: *Each label ℓ where a variable v is live is dominated by $\mathcal{D}_v$.*

Proof. Assume, ℓ is not dominated by $\mathcal{D}_v$. Then there exists a path from *start* to ℓ not containing $\mathcal{D}_v$. From the fact that v is live at ℓ it follows that there is a path from ℓ to some usage ℓ' not containing $\mathcal{D}_v$ (see Definition 2.3). This implies, that there is a path from *start* to ℓ' not containing $\mathcal{D}_v$ which is impossible in a strict program. □

Lemma 4.3: *If two variables v and w are live at some label ℓ, either $\mathcal{D}_v$ dominates $\mathcal{D}_w$ or vice versa.*

Proof. By Lemma 4.2, $\mathcal{D}_v$ and $\mathcal{D}_w$ dominate ℓ. Since dominance is a tree order (see Section 2.2), either $\mathcal{D}_v$ dominates $\mathcal{D}_w$ or $\mathcal{D}_w$ dominates $\mathcal{D}_v$. □

Remark 4.1: Note that this lemma states that each edge in the interference graph G can be directed according to dominance. Thus, if $uv \in E_G$ and $\mathcal{D}_u \preceq \mathcal{D}_v$ the edge uv points from u to v. One can further prove that the edge directions induced by dominance form an R-orientation (see section A.2.1.4) which suffices to show G's chordality.

Lemma 4.4: *If v and w interfere and $\mathcal{D}_v \preceq \mathcal{D}_w$, then v is live at $\mathcal{D}_w$.*

Proof. Assume, v is not live at $\mathcal{D}_w$. Then, there is no path from $\mathcal{D}_w$ to some usage ℓ' of v. So v and w cannot interfere. □

Lemma 4.5: *Let $uv, vw \in E_G$ and $uw \notin E_G$. If $\mathcal{D}_u \preceq \mathcal{D}_v$, then $\mathcal{D}_v \preceq \mathcal{D}_w$.*

Proof. Due to Lemma 4.3, either $\mathcal{D}_v \preceq \mathcal{D}_w$ or $\mathcal{D}_w \preceq \mathcal{D}_v$. Assume $\mathcal{D}_w \preceq \mathcal{D}_v$. Then (with Lemma 4.4), w is live at $\mathcal{D}_v$. Since u and v also interfere and $\mathcal{D}_u \preceq \mathcal{D}_v$, u is also live at $\mathcal{D}_v$. So, u and w are live at $\mathcal{D}_v$ which cannot be by precondition. □

Finally, we can prove that the interference graph of a program in SSA form contains no induced cycle longer than three:

Theorem 4.1 (Chordality): *The interference graph G of a program in SSA form is chordal.*

Proof. We will prove the theorem by showing that G has no induced subgraph $H \cong C^n$ for any $n \geq 4$. We consider a chain in G

$$x_1 x_2 \dots x_n \in E_G \quad \text{with } n \geq 4 \text{ and } \forall i \geq 1,\, j > i+1 :\, x_i x_j \notin E_G$$

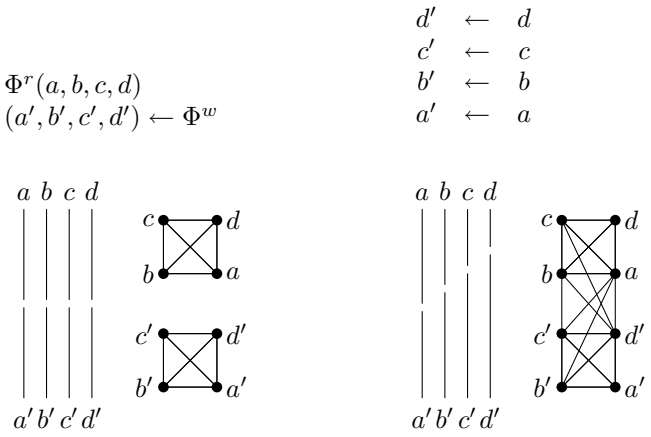
Without loss of generality we assume $\mathcal{D}_{x_1} \preceq \mathcal{D}_{x_2}$. Then, by induction with Lemma 4.5, $\mathcal{D}_{x_i} \preceq \mathcal{D}_{x_{i+1}}$ for all $1 < i < n$. Thus, $\mathcal{D}_{x_i} \preceq \mathcal{D}_{x_j}$ for each $j > i$.

Assume, there is an edge $x_1 x_n \in E_G$. Then, there is a label ℓ where x_1 and x_n are live. By Lemma 4.2, ℓ is dominated by $\mathcal{D}_{x_n}$ and due to the latter paragraph, ℓ is also dominated by each $\mathcal{D}_{x_i}, 1 \leq i < n$. Let us consider a label $\mathcal{D}_{x_i}, 1 < i < n$. Since $\mathcal{D}_{x_i}$ dominates ℓ, there is a path from $\mathcal{D}_{x_i}$ to ℓ. Since $\mathcal{D}_{x_i}$ does *not* dominate $\mathcal{D}_{x_1}$, there is a path from $\mathcal{D}_{x_i}$ to ℓ which does *not* contain $\mathcal{D}_{x_1}$. Thus, x_1 is live at $\mathcal{D}_{x_i}$. As a consequence, $x_1 x_n \in E_G$ implies $x_1 x_i \in E_G$ for all $1 < i \leq n$. So, G cannot contain an induced $C^n, n \geq 4$ and thus is *chordal.* □

4.1.2.1 SSA versus non-SSA Interference Graphs

The main difference between SSA and non-SSA interference graphs lies in the live range splitting induced by Φ-operations. Here, the parallel copy nature of Φ-operations is vital. Traditionally, Φ-operations are implemented using a sequence of copy instructions. These (sequential) copy instructions create interferences between the arguments and the results of the Φ-operation. These interferences were not present in the SSA program as shown in Figure 4.2.

In most compilers, SSA gets destructed before register allocation. Thereby, the sequence of the copies implementing a Φ-operation is fixed arbitrarily *before* register allocation. This imposes unnecessary constraints on the register allocator due to the added *artificial* interferences. It is more reasonable to allocate registers first and *then* arrange the copies in an order that fits the



(a) SSA (b) non-SSA

Figure 4.2: Copy instructions, live ranges and interferences in the SSA and non-SSA case

allocation, i.e. create only interferences between nodes that do not have the same color, which keeps the coloring of the SSA program valid for the destructed non-SSA program. Furthermore, it is just this premature fixing of the copy sequence that creates cycles in the interference graph rendering it unchordal and thus not efficiently treatable.

4.1.3 A Colorability Criterion

As coloring for chordal graphs has complexity $O(|V|^2)$ (see Remark A.1 in Appendix A) we can check the k-colorability of an interference graph by simply applying the coloring algorithm described in Remark A.1. However, by establishing a relation between the labels of the program and the cliques in the interference graph, we can check for k-colorability more efficiently. Trivially, all variables $v_1, \ldots, v_n$ being live at a label ℓ induce a clique in the interference graph of the program since they all interfere. While being not true in general, the converse does also hold for SSA-form programs[1]:

Lemma 4.6: *For each clique* $C \cong K^n \subseteq G, V_C = \{v_1, \ldots, v_n\}$*, there is a permutation* $\sigma : V_C \to V_C$*, such that* $\mathcal{D}_{\sigma(v_1)} \preceq \cdots \preceq \mathcal{D}_{\sigma(v_n)}$.

[1] This was proved independently by Bouchez et al. [2005]

Proof. By Lemma 4.3, for each v_i, v_j, $1 \leq i < j \leq n$ either $\mathcal{D}_{v_i} \preceq \mathcal{D}_{v_j}$ holds or $\mathcal{D}_{v_j} \preceq \mathcal{D}_{v_i}$. Thus $v_1, \ldots, v_n$ are totally ordered and every sorting algorithm can produce σ. □

Theorem 4.2: *Let G be the interference graph of a SSA-form program and C an induced subgraph of G. C is a clique in G if and only if there exists a label in the program where all $V(C)$ are live.*

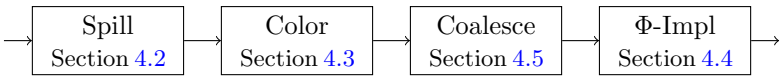
Proof. "⇐" holds by definition. "⇒": By Lemma 4.6, there exists a permutation σ of the $\{v_1, \ldots, v_n\} = V_C$ such that $\mathcal{D}_{\sigma(v_1)} \preceq \cdots \preceq \mathcal{D}_{\sigma(v_n)}$. Then, by Lemma 4.4, all $\sigma(v_1), \ldots, \sigma(v_{n-1})$ are live at $\mathcal{D}_{\sigma(v_n)}$. □

With the perfectness of chordal graphs, we directly obtain the following colorability criterion:

Theorem 4.3: *Let $\ell \in L_P$ be the label with the highest register pressure being k. Then, $\chi(\mathcal{I}_P) = k$.*

4.1.4 Directions from Here

The theoretical results of this section can be used for a new register allocation method circumventing many difficulties of the traditional approaches presented in Chapter 3. Most significantly, spilling and coloring can be completely separated. Theorem 4.3 assures that register pressure is identical to k-colorability. Thus, we can apply the pre-allocation spilling methods as described in Section 3.1.5.2 once. We then obtain a k-colorable interference graph as spilling lowered the register pressure at each label to k. This interference graph is colorable in $O\big(|V| \cdot \omega(G)\big)$. Furthermore, as $\omega(G) \leq k$ after spilling and k is constant, coloring is linear. As shown in Section 4.3, coloring can be done without materializing the interference graph itself by two passes over the program's dominance tree. Furthermore, as presented in Section 4.4, copy instructions can be used to turn the SSA program into a non-SSA program in a way that preserves the SSA coloring. The register allocator has then produced a *valid* register allocation. To lower the number of move instructions created by the Φ-implementation, Section 4.5 presents coalescing methods which optimize the coloring with respect to affinity edges. This leads to the following sequential register allocation process:



Section 4.6 discusses how the allocator has to be modified in order to handle register constraints imposed by the processor's instruction set architecture and the runtime's binary interface. This comprises an additional pass before spilling and a slight modification to the coloring phase.

4.2 Spilling

To apply the coloring algorithm of Section 4.3 and to obtain a valid register allocation, we need to establish that the program *is* k-colorable. In practice, it is likely that the register pressure in a program exceeds the number of available registers. Therefore, before coloring, spilling has to rewrite the program in order to make it k-colorable. This rewriting is done using load- and store-operations which transfer the values between the register file and the main memory. As the access to main memory can be several orders of magnitude slower than the access to registers, load- and store-operations must be used very carefully. The complexity of minimizing this memory traffic has been shown NP-complete by Farach and Liberatore [1998] and in particular by Bouchez et al. [2005] in the SSA context.

The failure-driven spilling of GCRAs is often not satisfactory, since the program's structure is insufficiently represented in the interference graph. Hence, most compilers apply a pre-coloring spilling phase which lowers the register pressure at the labels in the program to the amount of available registers (see Section 3.1.5.2). Due to Theorem 4.3 this heuristic suffices to produce a validly colorable program in the SSA-based setting.

Hence, the main focus of this section is to discuss how the existing and approved program-sensitive spilling approaches can be applied to SSA-form programs.

4.2.1 Spilling on SSA

Before presenting how existing techniques can be adapted to work in the SSA-based setting, we discuss several SSA specific issues a spilling algorithm has to deal with. Let us first define what spilling means. If the register pressure is too high at a certain label, one of the variables live there cannot be held in a register and must thus be moved to memory. Therefore, we enhance the program with a second set of variables $M = \{\mathbf{m_1}, \dots\}$ called *memory variables*. To make memory variables better distinguishable from ordinary variables, we typeset them in bold face. Memory variables are excluded from register allocation since they describe values in memory. However, each memory variable must have a location in memory where its value is held. Similarly to the map ρ which assigns each (ordinary) variable a register, we introduce a map $\& : M \to \mathbb{N}$ which assigns each memory variable a memory address. In contrast to the set of registers, we assume that there are at least as many memory addresses as memory variables so that memory variables never compete

for the same resource. Furthermore, we allow three operations to read/write memory variables:

1. An operation *spill*[2] which reads an ordinary variable and writes to a memory variable.
2. An operation *reload* which reads a memory variable and writes to an ordinary variable.
3. The Φ-operation.

All other operations must work on ordinary variables.[3] A spilling algorithm has to make sure that this condition is met by placing spill- and reload-operations at suitable places. Two aspects demand special attention and are specific to spilling on the SSA form.

4.2.1.1 Φ-operations with Memory Operands

A Φ-operation may define more variables as registers are available. Thus, an SSA-based spiller must not only insert spill and reloads but also rewrite Φ-operations. Let k be the number of registers. After spilling, $|res(\ell) \setminus M| \leq k$ must hold for each label ℓ containing a Φ-operation.

Note carefully that a Φ-operation with memory operands may introduce memory operations when the Φ-operation gets lowered by the Φ-implementation phase as described in Section 4.4.2. Whether a Φ-operation with memory operands causes memory traffic depends on the spill slot assignment &. If for each memory Φ-result $\mathbf{y}$ and each corresponding argument $\mathbf{x}_i$ there is $\&\mathbf{y} = \&\mathbf{x}_i$ then no code has to be generated for the memory part of the Φ-operation at all. Thus, "optimal" spilling approaches must not only consider the placement of spill and reload operations in order to fulfil the register pressure criterion *but* also the assignment of spill slots. To our knowledge, this problem has not been studied yet. Even the "optimal" approaches presented in Section 3.2.2 ignore this fact.

We also require that if the result variable of a Φ-operation is a memory variable, all corresponding operands are memory variables as well. One could also allow having mixed memory/register operands for a result but this would

[2] The reason why we name these instructions *spill* and *reload* is that we want to keep the spilling process as architecture-independent as possible. When the spiller inserted these operations the instruction selection mechanism can turn them into actual machine instructions.

[3] This reflects common load/store architectures. Some CISC machines, such as x86, have the possibility to directly process memory operands.

defer a part of the spill and reload code insertion to the SSA destruction phase. We found it more elegant to have only the spiller handling the spill and reload code.

4.2.1.2 SSA Reconstruction

After spilling, the program must be in SSA form to benefit from the results of Section 4.1. However, if a variable v is reloaded at several locations, these reloads form additional definitions of v. This clearly violates the static single assignment property (see Figure 4.3). Therefore, for each variable v for which reloads are introduced by the spiller, SSA has to be reconstructed.

Instead of performing an SSA construction algorithm on the whole program, we can restrict ourselves to the variables which were involved in spilling. All other variables still have only one definition.

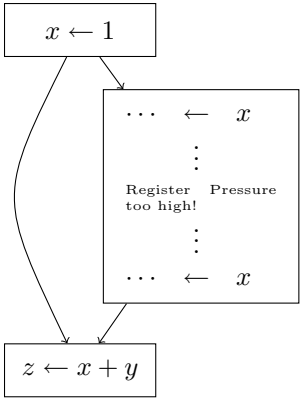
We follow the same principles as the classical SSA construction algorithm by Cytron et al. [1991]: For each spilled variable x compute the iterated dominance frontier of the set of labels L containing the definition and all labels containing a reload of x:

$$L = \{\ell \mid res(\ell) = \{x\} \wedge op(\ell) = reload\} \cup \{\mathcal{D}_x\}$$

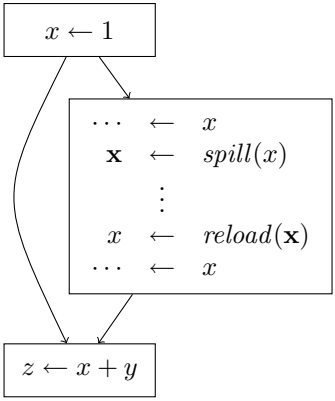
Firstly, give each variable defined by the labels in L a new version number. This reinstalls the single assignment property. Secondly, we have to rewire each use of former x to one of the variables defined at the labels in L. Let F bet the set of iterated dominance frontiers of L (for example, see [Appel and Palsberg, 2002, page 406] for an algorithm to compute these). Let U be the set of uses of x. For each $\ell \in U$ we search up the dominator tree. If we reach a label ℓ_r in L, we subsitute x in $arg(\ell)$ by $res(\ell_r)$. If ℓ is in the iterated dominance frontier F, we insert a Φ-operation and search the predecessors of ℓ for valid definitions of x. Algorithm 4.1 shows this procedure in pseudocode. Note that by re-wiring the usages of several variables, some variables defined by Φ-operations may not be used anymore. A dead code elimination pass after spilling will remove these.

4.2.2 Generic Procedure

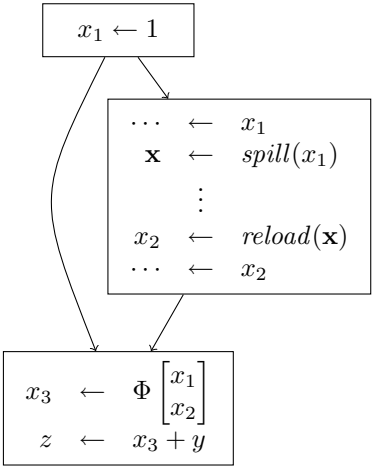
Assume some spilling algorithm produces a set of spills and reloads for some variable y in the original program. The new program P' has to be rewritten to reflect the spilling algorithm's results. Let us consider possible spills and reloads for y. Firstly, y can be the result of a Φ-operation or not. If it is, the spiller has to determine if y's entry in the Φ-operation shall be completely



(a) Example SSA Program before Spilling



(b) x spilled, SSA violated



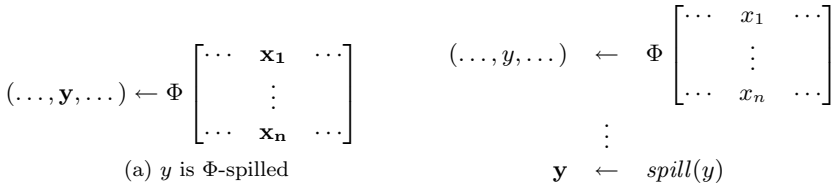
(c) x spilled, SSA reconstructed

Figure 4.3: Spilling and SSA

Algorithm 4.1 SSA reconstruction

```
procedure SSA-Reconstruction(set of variables D)
    F ← iterated dominance frontier of {D_d | d ∈ D}
    U ← {ℓ | D ∩ arg(ℓ) ≠ ∅}          ▷ All labels using one of the vars in D.
    for all u ∈ U do
        for all i for which arg(u, i) ∈ D do
            x ←Find-Def(u, i, D, F)
            arg(u, i) ← x

procedure Find-Def(label u, int p, set of variables D, set of labels F)
    if op(u) = Φ then
        u ← pred(u, p)
    while u ≠ start do                  ▷ While we are not at the start label
        if res(u) ∩ D ≠ ∅ then          ▷ Check if the label defines a var of D
            return res(u) ∩ D           ▷ If so return the defined variable
        else if u ∈ F then              ▷ If the label is in the dominance frontier, we have to insert a Φ
            if op(u) ≠ Φ then           ▷ If there is no Φ insert one
                Insert a new label u'
                pred(u') ← pred(u)
                pred(u) ← ⟨u'⟩
                op(u') ← Φ
                u ← u'
            Insert new result y' with an empty row in the Φ-matrix
            D ← D ∪ y'                  ▷ Needed to break recursion
            for i ∈ 1 ... |pred(u)| do
                Set i-th entry in y's row to Find-Def(u, i, D, F)
        u ← idom(u)                     ▷ Go up the dominance tree
```



(a) y is Φ-spilled

(b) y is spilled after the Φ

Figure 4.4: Spilling at a Φ-operation

transferred to memory or if y is ordinarily written to a register by the Φ and spilled afterwards. In the former case, we say that y was Φ-spilled. Figure 4.4 illustrates both situations. Section 4.4.2 deals with the implementation of spilled Φ-operations.

Before describing the program transformation to be performed after spilling let us state more precisely, what the result of a a spilling algorithm is. For each variable $y \in V$ we expect three items to be computed:

- A set $R_y \subseteq L$ containing inserted reload operations

$$\ell_i : y_i \leftarrow reload(\mathbf{y}_1)$$

  Note that we explicitly direct all reloads to the first inserted spill ($\mathbf{y}_1$) and rely on the SSA reconstruction algorithm to rewire them to their correct definitions as explained below.

- A set $S_y \subseteq L$ of labels with spill operations

$$\ell_i : \mathbf{y}_i \leftarrow spill(y)$$

- A flag φ_y denoting that y is the result of a Φ-operation and is to be Φ-spilled. In that case S_y solely contains the label of the Φ-operation which defines y.

What is left to do is to rewire all uses of the ordinary and memory variables related to y to their correct definitions. This comprises setting the memory variable arguments for the reloads as well as the ordinary variable arguments for the uses of y. Thus, we invoke the SSA reconstruction algorithm twice. Once passing the spills as definitions and once passing the reloads and the original definition of y as definitions.

Note that if a variable is to be Φ-spilled, spills for all x_i in the corresponding row of the Φ-matrix must be present, since the whole row has to be turned into memory variables.

4.2.3 Rematerialisation

An alternative to spilling a variable to memory is to recompute its value. This is commonly known as *rematerialisation*. Assume y is a variable whose value was computed by a label $y \leftarrow \tau(x_1, \ldots, x_n)$. Recomputing y at some other place of the program obviously requires that the x_i are live at this place. Hence, the spilling decisions for the x_i influence the recomputability of y. Thus, rematerialisation is especially attractive for operations with few operands or even *none*. This comprises constants as well as unaliased load instructions. These are loads from memory addresses for which the compiler can exactly determine the set of store instructions writing to this location. Between two stores to such an unaliased memory address the load instruction can be arbitrarily duplicated. Such load instructions occur frequently to access the parameters of functions passed on the stack.

Rematerialisation can be easily integrated into the SSA-based setting as discussed in the last paragraphs. The labels recomputing a variable correspond to the reloads of that variable. Thus, these labels can be added to the set R_y like reloads. If the spiller produced only rematerialisations for a variable there is obviously no spill.

4.2.4 A Spilling Heuristic

In this section, we show how a well-known spilling heuristic can be integrated into an SSA-based register allocator. Given k registers and a label ℓ in a block $B = \langle \ell_1, \ldots, \ell_n \rangle$, where $l > k$ variables are live. Clearly, $k - l$ variables have to reside in memory at ℓ for a valid register allocation. The method of Belady selects those to remain in memory, whose uses are farthest away from this label, i.e. $N(\ell, v)$ is maximal. Whereas "away" means the number of instructions that have to be executed from ℓ until the use is reached. If the use is in the same basic block, this number is simply given by the number of instructions between ℓ and the use. If the use is outside ℓ's block, we have to define a reasonable measure. This could by the minimum, average or maximum of all next uses in the successor blocks depending on how pessimistic/optimistic

the heuristic shall be. We will here use the minimum variant:

$$N'(\ell, v) = \begin{cases} \infty & \text{if } v \text{ is not live at } \ell \\ 1 + \min\limits_{\ell' \in succ(\ell)} N(\ell', v) & \text{otherwise} \end{cases} \tag{4.1}$$

$$N(\ell, v) = \begin{cases} 0 & \text{if } v \text{ is used at } \ell \\ N'(\ell, v) & \text{otherwise} \end{cases} \tag{4.2}$$

We will apply Belady's method to each basic block separately and combine the results to obtain a solution for the whole procedure. Consider a basic block B. We define P as the set of all variables live in at B and the results of the Φ-operation in B if there is one. These variables are *passed* to the block B from another block. A value v live in at B is passed on each incoming control flow edge to B. For a result y of a Φ-operation

$$(\dots, y, \dots) \leftarrow \Phi \begin{bmatrix} \cdots & x_1 & \cdots \\ & \vdots & \\ \cdots & x_n & \cdots \end{bmatrix}$$

the $x_1, \dots, x_n$ are passed along the respective incoming control flow edges to B.

Since we have k registers, only k variables can be passed to B in registers. Let $\sigma : P \to P$ be a permutation which sorts P increasingly according to N (relative to the entry of block B). The set of variables which we allow to be passed in registers to B is then

$$I := \left\{ p_{\sigma(1)}, \dots, p_{\sigma(\min\{k,l\})} \right\}$$

We apply the Belady scheme by traversing the labels in the block from entry to exit. A set Q of maximal size k is used to hold all variables that are currently in registers. Q is initialized with I, optimistically presuming that all variables of I are kept in registers upon entering the block B.

At each label ℓ, the arguments $arg(\ell)$ have to be in registers when the instruction is reached. Assume that some of the arguments are not contained in Q, i.e. $L := arg(\ell) \setminus Q$ is not empty. Thus, reloads have to be inserted for all variables in L. By inserting reloads, the values of the variables are brought into registers. Thus,

$$\max\{|L| + |Q| - k, 0\}$$

variables are removed from Q. As the method of Belady suggests, we remove the ones with the highest N. Furthermore, if a variable $v \in I$ is displaced

before it was used, there is no sense in passing v to B in a register. in_B denotes the set of all $v \in I$ which are used in B before they are displaced.

We assume, that all variables defined by $\tau = op(\ell)$ are written to registers upon executing τ. This means, that

$$\max\{|res(\ell)| + |Q| - k, 0\}$$

variables are displaced from Q when τ writes its results[4]. Note, that for all $v \in arg(\ell)$, there is $N(v, \ell) = 0$ which implies that the $arg(\ell)$ will be displaced lastly. However, it is not the uses at ℓ which decide but the uses *after* ℓ. This is resembled by N' as defined above. We define the set out_B to be the value of Q after processing each label in the block.

Finally, after processing each block as described above, we have to combine the results to form a valid solution for the whole program. In particular, we have to assure, that each variable in in_B for some block B must reach B in a register. Therefore, we have to check each predecessor P of B and insert reloads for all $in_B \setminus out_P$ on the edge from P to B[5]. All the spills and reloads determined by this procedure are then given to the algorithm described in Section 4.2.2 to restore the SSA property.

[4] Note, that $Q \cap res(\ell) = \emptyset$ since the program is in SSA form.

[5] Inserting code on an edge is possible by eliminating critical edges and placing the code in the appropriate block.

4.3 Coloring

Let us assume that the program is k-colorable, i.e. the spilling phase lowered the register pressure to k at all labels where it exceeded k. Hence, the chromatic number $\omega(G)$ of (chordal) interference graph G of the program is less or equal to k. A chordal graph G can be efficiently colored in $O\big(|V| \cdot \omega(G)\big)$ using perfect elimination orders (see Appendix A for the necessary graph theory and specifically Remark A.1). The coloring process is similar to the simplify/select-scheme presented in Section 3.1. The major difference is that a node has not only to be of insignificant degree but also *simplicial*[6] to be removed from the graph. Note that this is a strengthening of the criterion used in Chaitin-style allocators: if $\mathcal{I}_P$ is k-colorable, the largest clique is of size k due to the perfectness of chordal graphs. All neighbors of a simplicial node are contained in the same clique. Thus it has at most $k-1$ neighbors and is of insignificant degree.

Lemma 4.5 directly leads to the fact that dominance induces a perfect elimination order on the interference graph. Assume P to be the transformed version of an SSA-form program using $\mathcal{T}_\Phi$ and $\mathcal{I}_P = (V, E, A, c)$.

Theorem 4.4: *A variable v can be added to a PEO of $\mathcal{I}_P$ if all variables whose definitions are dominated by the definition of v have been added to the PEO.*

Proof. To be added to a PEO, v must be simplicial. Let us assume, v is *not* simplicial. Then, by definition, there exist two neighbors a, b of v that are not connected ($va, vb \in E$ and $ab \notin E$). By proposition, all variables whose definitions are dominated by $\mathcal{D}_v$ have been added to the PEO and removed from $\mathcal{I}_P$. Thus, $\mathcal{D}_a \preceq \mathcal{D}_v$. Then, by Lemma 4.5, $\mathcal{D}_v \preceq \mathcal{D}_b$ which contradicts the proposition. Thus, v is simplicial. □

Thus, we obtain a perfect elimination order on $\mathcal{I}_P$ by a post-order traversal of the dominance tree of the program. Conversely, we can start coloring at the label which dominates all other labels moving to the label dominating all others but the former one and so on. Thus, coloring can be done in a pre-order walk of the dominance tree as shown in Algorithm 4.2.

[6] i.e. all its neighbors form a clique.

Algorithm 4.2 Coloring an interference graph of a SSA-form program

```
procedure Color-Program(Program P)
    Color-Recursive(start block of P)

procedure Color-Recursive(Basic block B = ⟨ℓ_1, ..., ℓ_n⟩)
    for all x ∈ livein(B) do                    ▷ All variables live in have
                                                  already been colored
        assigned ← assigned ∪ ρ(x)              ▷ Mark their colors as occupied
    for i ← 1 to n do
        for all x ∈ arg(ℓ_i) do
            if last use of x then
                assigned ← assigned ∖ ρ(x)
        for all y ∈ res(ℓ_i) do
            ρ(y) ← one of R ∖ assigned

    for {C | B = idom(C)} do                    ▷ Proceed with all children
                                                  in the dominance tree
        Color-Recursive(C)
```

4.4 Implementing Φ-operations

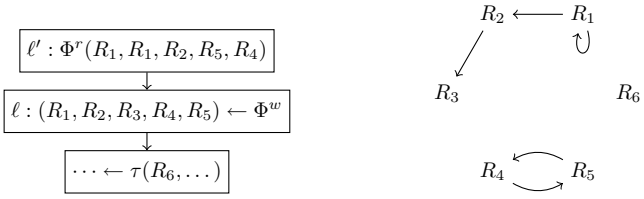
Traditionally, register allocation is done after SSA destruction mainly due to the fact that the SSA form and thus its destruction was invented after register allocation. Thus, to let existing register allocation approaches still be applicable, SSA form is commonly destroyed before register allocation is performed. However, to exploit the advantages of SSA-based register allocation, SSA destruction before register allocation is no longer necessary. Furthermore, as shown in Section 4.1, premature SSA destruction unnecessarily constrains register allocation.

However, since no standard processor provides instructions to directly implement Φ-operations, they have to be removed at some point in the compiler. In our setting, this point in time is *after* register allocation. To make use of the coloring found for the SSA program, Φ-operations have to be removed in a way that transfers the SSA coloring to the resulting non-SSA program. One could easily implement Φ-operations using loads and stores. However, this is inefficient due to the memory traffic. Thus, we will discuss how Φ-operations can be implemented with other standard instructions, such as register-register copies.

Consider a program P with a label ℓ containing a Φ-operation and the transformed version $\mathcal{T}_\Phi(P)$ as described in Section 2.3.1. In the transformed version, the Φ at ℓ is replaced by a Φ^w-operation. Each control flow edge to ℓ is extended by a special label which holds a Φ^r-operation corresponding to the row in the Φ-matrix. Firstly, we will only consider the register variable arguments and results of the Φ-operation. These arguments and results are contained in registers. The treatment of memory variables will be discussed afterwards.

4.4.1 Register Operands

Consider the (register) results $Y = y_1, \ldots, y_m$ of the Φ^w and the k-th predecessor ℓ' of ℓ corresponding to the k-th row $X = x_{k1}, \ldots, x_{km}$ in the Φ-matrix. Consider a valid register allocation $\rho : V \to R$ on P. Let R_Y be the set of registers assigned to the results of Φ^w and R_X be the set of registers assigned to the arguments of Φ^r. Note, that in general $|R_X| \leq |R_Y|$, since not all of the $arg(\ell')$ need to be distinct. Thus, multiple arguments of the Φ^r might be identical and thus have been assigned the same register. Let $\beta : Y \to X$ be a map with $\beta(y_i) = x_{ki}$ assigning each result the argument it gets assigned by the Φ-operation.



(a) Φ^r-Φ^w-pair (b) Register Transfer Graph T

Figure 4.5: A register allocated Φ^w-Φ^r-pair and the corresponding register transfer graphs T and T'

Let us now consider the (directed) *register transfer graph* $T = (R, T_E)$. The nodes of the transfer graph are formed by the registers. The edges $T_E \subseteq R \times R$ represent the value transport between the registers caused by the Φ-operation at ℓ: We draw an edge from R_i to R_j for each $y \in Y$ with $\rho(y) = R_j$ and $\rho(\beta(y)) = R_i$ to indicate that the value of R_i has to be moved to R_j. More formally:

$$T_E = \Big\{ \big(\rho(\beta(y)), \rho(y)\big) \mid y \in Y \Big\} \tag{4.3}$$

Note that each node has only one edge pointing at it since each register is written at most once by the Φ^w-instruction. However, a node might very well have more than one outgoing edge. This indicates that a variable is duplicated by the Φ-operation.

Figure 4.5a shows an example for a Φ^w-Φ^r-pair and the corresponding register transfer graph. The first two arguments of the Φ^r are the same as they are kept in the same register (R_1). The Φ^w creates a copy of the value in R_1 and stores it in R_2. Hence, the node R_1 in the corresponding transfer graph T (see Figure 4.5b) has two outgoing edges.

The register-register move instructions concerning T are now generated by the following scheme:

1. First, consider the set of registers

$$U = \big\{ \rho(y) \mid y \in Y \wedge \beta(y) = \texttt{undef} \big\}$$

which will hold a copy of the dummy variable `undef` (see Section 2.3.2). As their content is undefined, no move instruction has to be created to initialize their value. Hence, all incoming edges to the nodes in U can be deleted.

2. Initialize a set F containing all unused registers at the label of the Φ^r-operation ℓ'. Note that F is *not* the set of nodes in T which have no incident edges. It has to be calculated by regarding all variables live at ℓ' and the registers allocated to them. In our example in Figure 4.5b, R_6 has no incident edges but is not in F since it is allocated to a variable which is live in at ℓ.

3. While there is an edge $e = (r, s), r \neq s$ for which the outdegree of s equals 0:
   The register s does not carry an argument value which has to be transferred to a result. For example, the edge (R_2, R_3) in Figure 4.5b fulfills this property. Thus, we can safely write to register s by issuing a copy

   $$s \leftarrow r$$

   Then, e is deleted from T as the move was issued. To free the registers as early as possible, we let all remaining outgoing edges of r originate at s except for self-loops. (Although it would not be incorrect to redirect self-loops, it would create unnecessary move instructions.) We add r to the set of free registers F and remove s from F.

4. Now, T is either empty or contains one or more cycles of the form $C = \{(r_1, r_2), (r_2, r_2), \ldots, (r_n, r_1)\}$ such as the cycle (R_4, R_5), (R_5, R_4) in Figure 4.5b. Such a cycle corresponds to a permutation of registers which can be implemented in various ways:

   (a) Trivially, self-loops (cycles of length 1) generate no instruction.

   (b) Assume F is not empty and contains a free register r_t. Then we can implement the cycle C by a sequence of copy instructions

   $$\begin{aligned} r_t &\leftarrow r_1 \\ r_1 &\leftarrow r_2 \\ r_2 &\leftarrow r_3 \\ &\vdots \\ r_n &\leftarrow r_t \end{aligned}$$

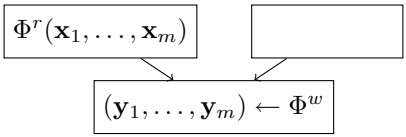
   (c) Assume F is empty and thus there is no free register. Note that this is only the case if transfer graph solely consists of cycles (including self-loops) and the register pressure is at maximum at ℓ'. As known from algebra, each permutation can be implemented using a series of transpositions, in our case an instruction that swaps two registers.

If the processor does not provide swap instructions, a transposition can be emulated using three xor-operations or, if the machine uses two's-complement arithmetics, additions or subtractions (see Warren [2002] for example). If no such alternative is available, e.g. for floating point registers, a load and a store instruction have to be issued to save and restore one register of the permutation. This is the only case where SSA-based register allocation can introduce additional memory traffic *after spilling*.

4.4.2 Memory Operands

The memory part of Φ-operations has to be implemented accordingly. Analogous to the register part, the implementation of the memory part is dependent on the spill slot allocation &. If a memory Φ-result $\mathbf{y}$ and an argument $\mathbf{x}$ in the corresponding row have different spill slots, the value of $\mathbf{x}$ has to be moved to $\&\mathbf{y}$. Note that the coalescing algorithms presented in Section 4.5 are directly applicable to memory variables. Hence, coalescing spill slots can (and almost always will) decrease the number of load and store instructions. The only difference is that the number of colors is unlimited and can thus be set to the number of memory variables in the program.

Unfortunately, most architectures will require a register to load a value to and store it back to a different location.[7] This additional register demand is only dependent on the spill slot allocation and therefore *not* considered by the spilling algorithm. Let $\mathbf{y}_1, \ldots, \mathbf{y}_m$ be the memory results of a Φ-operation and $\mathbf{x}_1, \ldots, \mathbf{x}_m$ be the corresponding arguments of a predecessor block B of the Φ's label:



The memory copy has to be implemented somewhere in block B where all $\mathbf{x}_i$ have been defined. As it is very likely that there will be some place in B where a register is free, ignoring this additional register demand while spilling is justified. Otherwise, the spill slot assignment has to be integrated into the spilling mechanism.

[7] On x86, one can use `push` and `pop` instructions with memory operands to perform the memory part of a Φ-operation without an additional register.

4.5 Coalescing

As we have seen in the past sections, live range splitting occurs quite frequently in an SSA-based register allocator. Firstly, by the implicit live range splitting of Φ-operations and secondly by the spilling phase (as presented in Section 4.2). Furthermore, register constraint handling will also introduce additional live range splits as presented in Section 4.6. As live range splitting causes the insertion of move instructions, aggressive live range splitting can result in a significant performance loss. Thus, an optimizing compiler should try to minimize the number of inserted moves. This optimization problem is commonly known as the copy coalescing problem.

In this chapter we will give a precise definition of the coalescing problem in the SSA-based setting and prove it NP-complete even for programs that do not contain register constraints but only Φ-operations. Then, we present a heuristic approach to the coalescing problem and an optimal method using integer linear programming (ILP).

4.5.1 The Coalescing Problem

As described in Section 4.4, pairs of Φ^r- and Φ^w-operations can be implemented using move instructions and register swaps. However, the number of required move/swap instructions depends on the register transfer graphs as introduced in that section. Obviously, if the registers allocated to the Φ^w-Φ^r-pair match properly, i.e. there are only self-loops in the transfer graph, no move instruction is needed at all. To express this wish, affinity edges are added to the interference graph as described in Section 2.4.2. For each Φ^r-Φ^w-pair

$$\begin{array}{lcl} & & \Phi^r(x_1, \ldots, x_m) \\ (y_1, \ldots, y_m) & \leftarrow & \Phi^w \end{array}$$

we add the affinity edges $x_1y_1, \ldots, x_my_m$. Each affinity edge can be annotated by a number denoting the costs incurred by failing to assign the incident nodes the same register. In a compiler, these costs can be derived from measured or calculated execution frequencies (cf. Wu and Larus [1994] for example) of the label the Φ^r is located at. Thus, more often executed Φ^r-operations create affinity edges having higher costs. Hence, we define coalescing as the task of finding a *good* coloring for the IG taking into account the costs of the affinity edges. Let $\mathcal{I}_P = (V, E, A, c)$ be the IG of some program P. Furthermore, let ρ be a valid register allocation on P and $|\cdot|_\rho : A \to \mathbb{N}$ the function returning

the cost of an affinity edge given the allocation ρ:

$$|xy|_\rho = \begin{cases} 0 & \text{if } \rho(x) = \rho(y) \\ c(xy) & \text{otherwise} \end{cases}$$

If $|xy|_\rho = 0$ we also say that ρ *fulfils* the affinity xy. The cost function summarizing the costs of the whole interference graph $G = (V, E, A, c)$ under a coloring ρ is then given as

$$\|G\|_\rho = \sum_{xy \in A} |xy|_\rho$$

Definition 4.2 (SSA-Coalescing): *Let $G = (V, E, A, c)$ be an interference graph of an SSA program. SSA-Coalescing is the optimization problem*

$$\min_\rho \|G\|_\rho$$

Remark 4.2: This formulation is only an approximation in terms of the number of instructions needed to implement the Φ^r-Φ^w-pair. It only allows for optimizing the *size* the resulting parallel copy, not their inner structure. Hence, there are register allocations with minimal costs which need more instructions to implement the pair than other allocations with higher costs. For example, consider the register transfer graphs shown in Figure 4.6. Let G be the IG

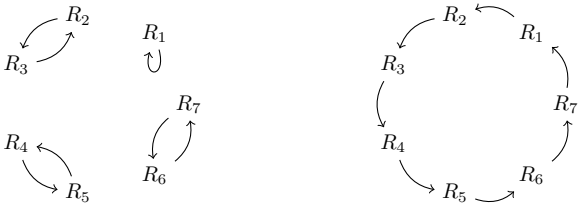


Figure 4.6: Register Transfer Graphs for Allocations ρ_1 and ρ_2

of the program and the costs of all affinity edges be 1. According to the cost function defined above, $\|G\|_{\rho_1} = 6$ and $\|G\|_{\rho_2} = 7$. Thus, ρ_1 is a better coloring with respect to $\|\cdot\|$. However, ρ_1 will demand nine copy instructions while ρ_2 will only demand eight. Furthermore, ρ_1 will demand three swap instructions while ρ_2 will demand six.

However, as the presented coalescing algorithms are capable of fulfilling the vast majority of affinities, this formulation is sufficient in practice.

4.5.1.1 The Complexity of SSA-Coalescing

Allowing arbitrary interference graphs G and especially arbitrary affinities between nodes of G, one can easily produce instances where coalescing directly corresponds to a general graph coloring problem. This has also been noted by Appel and George [2001].

However, this does not provide the complexity of coalescing on graphs arising from SSA programs we are concerned with, since our graphs do not exhibit arbitrary structure. The interference part is chordal and the affinity part is generated by live range splitting. For proving the NP-completeness of coalescing on this subset of graphs, we have to generate an SSA program from an instance of an NP-complete problem and show that coalescing on the program's interference graph induces a solution to the instance of the NP-complete problem.

The proof of the NP-completeness of SSA-Coalescing on SSA interference graphs is a classical reduction proof using graph coloring. First we select an arbitrary graph H and construct an SSA program from it. The Φ-operations in that program are placed in a way that an optimal coalescing induces an optimal coloring in H and vice versa. The technique is a variation of the one used by Rastello et al. [2003].

Theorem 4.5: *SSA-Coalescing is NP-complete concerning the number of Φ-operations in the program.*

Proof. We reduce the NP-complete problem of finding a k-coloring of a graph H to SSA-Coalescing. We construct a SSA-form program from H in the following way:

- There is one label ℓ containing a Φ-operation

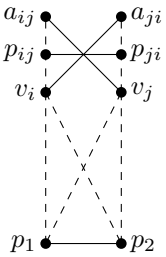
$$(p_1, \ldots, p_k) \leftarrow \Phi \begin{bmatrix} v_1 & \cdots & v_1 \\ \vdots & & \vdots \\ v_{|V_H|} & \cdots & v_{|V_H|} \end{bmatrix}$$

- For each node $v_i \in V_H$ there is a variable v_i, a label ℓ_i containing a definition of v_i and a control flow edge $\ell_i \to \ell$

- For each edge $v_i v_j \in E_H$ we add two variables a_{ij} and a_{ji} and two labels ℓ_{ij}, ℓ_{ji} which contain a definition of a_{ij} and a_{ji} respectively.

- For each edge $v_iv_j \in E_H$ we further add a label ℓ'_{ij} containing a Φ-operation
$$(p_{ij}, p_{ji}) \leftarrow \Phi \begin{bmatrix} v_i & a_{ji} \\ a_{ij} & v_j \end{bmatrix}$$
- For each edge $v_iv_j \in E_H$ following control flow edges are added:
  - $\ell_i \to \ell_{ji}$
  - $\ell_j \to \ell_{ij}$
  - $\ell_{ij} \to \ell'_{ij}$
  - $\ell_{ji} \to \ell'_{ij}$

In the interference graph G of the constructed program, an edge $v_iv_j \in E_H$ creates the following gadget:

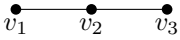


The dashed lines denote that assigning the two nodes the same color will lower the cost by one. First of all, let us consider a lower bound for the costs. Since each v_i can only be assigned one color, there are $k-1$ nodes p_i having other colors than v_i, resulting in costs of $k-1$ for v_i. So, each optimal solution incurs costs of at least $|V_H|(k-1)$.

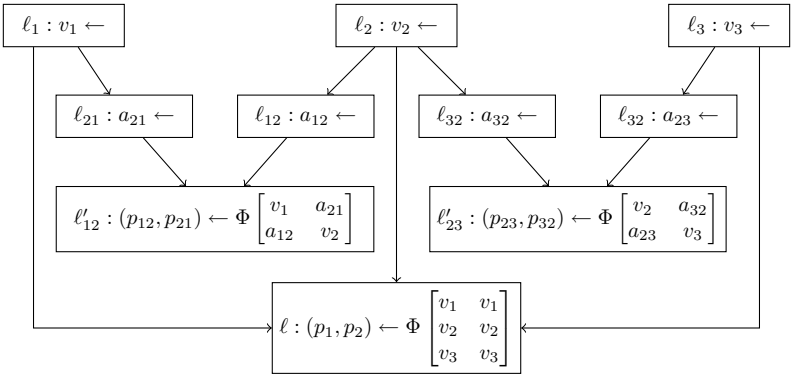
Assume H is k-colorable and let g be a k-coloring of H. Assign each $v_i \in V_G$ the color $g(v_i)$, $v_i \in V_H$. Since $g(v_i) \neq g(v_j)$ for each $v_iv_j \in E_H$, the nodes v_i, a_{ij}, p_{ij} can be assigned the same color. Thus, g incurs costs of exactly $|V_H|(k-1)$. So each k-coloring of H induces an optimal solution of SSA-Coalescing.

A coloring f of G, which does not correspond to a k-coloring of H is no (optimal) solution of SSA-Coalescing: Since f corresponds to no k-coloring of H, there is $v_iv_j \in E_H$ for which $f(v_i) = f(v_j)$. Thus, $f(a_{ij}) \neq f(v_i)$ and $f(a_{ji}) \neq f(v_j)$ resulting in costs strictly greater than $|V_H|(k-1)$. Thus, SSA-Coalescing is NP-complete with the number of Φ-operations. □

Example 4.1: Consider the following example graph H:



The graph H above generates following program fragment:



Remark 4.3: This proof also demonstrates that the quality of copy minimization is generally not only dependent on the number of colors available but also on the interference graph's structure. No matter how many colors (registers) we have, the costs of $|V_H|(k-1)$ are not eliminable. Thus, each interference graph exhibits a lower bound of copy costs caused by its structure.

4.5.2 A Coalescing Heuristic

In this section, we present a heuristic for finding good colorings with respect to the affinity costs. Our approach is based on similar observations as the optimistic register coalescing method by Park and Moon [2004] as presented in Section 3.1.1. However, we never modify the *structure* of the interference graph as we wish to maintain its chordality. Merging nodes in the graph can easily destroy its chordality and hence all the knowledge about colorability this provides. We only modify the coloring of the graph by trying to assign affinity related *chunks* of nodes the same color. Thereby, we try to find a *better* coloring according to $\|G\|$.

Note that up to now, we did not need to construct the interference graph as a data structure. For this coalescing heuristic, we need interference checks and iterating over the interference neighbors of each node. As shown below, even this can be done without constructing the graph itself.

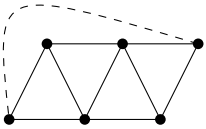


Figure 4.7: Graph with an unfulfillable affinity for $k = 3$

Before continuing, let us give some definitions related to affinity edges which we will use in the following. In the following, let $G = (V, E, A, c)$ be an interference graph.

Definition 4.3 (Affinity related): *Let $x, y \in V$. We say x and y are* affinity related *if there exists a path from x to y in G consisting of affinity edges. We then write $x \sim y$. Note that $\sim$ is a equivalence relation. Furthermore, we say a set X is* affine *if all its members are pairwise affinity related.*

Definition 4.4 (Affinity Component): *X is an affinity component if and only if X is closed under $\sim$.*

In general, not all nodes inside an affinity component can be assigned the same color due to interferences. Either two nodes in an affinity component directly interfere or there is simply no coloring allowing them to possess the same color. The graph in Figure 4.7 shows an example where two non-interfering nodes cannot be assigned the same color (assuming that three colors are available for coloring). As we want to assign all nodes in the chunk the same color, since the chunks do not directly correspond to affinity components since an affinity component is not necessarily free of interferences. Thus, upon creating the chunk, it is often the case that several affinity edges inside the same affinity component are "sacrificed", i.e. we deliberately do not fulfil the affinity. The following greedy method is applied to partition the graph into interference-free chunks:

Place each node in a separate chunk. Sort all affinity edges according to their costs from high to low. Now process each affinity edge $a = xy$ from the most expensive to the cheapest. If no node in x's chunk does interfere with any node of y's chunk, the chunks of x and y are unified (see Algorithm 4.5).

Afterwards, the chunks are put into a priority queue with the most expensive one in front. The cost of a chunk is the sum of the costs of all affinity edges whose nodes are contained in the chunk. Then, we start (re-)coloring the chunks. While the priority queue is not empty, we pick the first (most ex-

pensive) chunk C. For each color c we try to color each node inside the chunk with c. This is done with recursive recoloring which is explained below. The order of the nodes in which the recoloring is attempted is free since in practice there are rarely two nodes of the same affinity component in the same interference component; i.e. recoloring one node has no influence on other nodes inside the same chunk. Consider the interference graph in Figure 5.5 on page 96. It consists of several interference components which are linked by affinities. Recoloring one node just affects the interference component of the node and *no* node outside the interference component.

After recoloring to c, probably not all of the affinities inside the chunk C will be fulfilled. Some of the nodes in C might just not be colorable with c. Consider the subset $S_c \subseteq C$ containing all nodes which were recolorable to c. S_c in turn is partitioned into disjoint sets $S_c = S_c^1 \uplus \cdots \uplus S_c^n$ by the relation $\sim$, i.e. there several affine subsets in S_c. Each of the S_c^i represents a certain amount of eliminated costs which is determined by the weights of the affinity edges connecting the nodes inside S_c^i. Of course, as all nodes in S_c^i are colored with c, all of these affinity edges are fulfilled. Let $|S_c^i|$ denote the costs eliminated by S_c^i.

Let $M_c \in \{S_c^1, \ldots, S_c^n\}$ be the subset of S_c which eliminates the most costs for color c, i.e.

$$M_c = S_c^j, \quad j = \arg \max_{1 \le i \le n} |S_c^i|$$

As we try the recoloring for all colors, there is a color c' for which $|M_{c'}|$ is maximal, or more formally

$$c' = \arg \max_{1 \le c \le k} |M_c|$$

This color is chosen to be the definite color for the nodes in $M_{c'}$. Then all nodes in $M_{c'}$ are *fixed* to c', i.e. changing their color by future recoloring attempts is prohibited to prevent already eliminated costs from arising again.

The rest of the chunk, i.e. all nodes of the chunk not inside $M_{c'}$ are packed together to a new chunk and re-inserted into the priority queue (probably at another position due to the changed costs). See Algorithm 4.3 for details.

Recoloring is performed recursively through the graph (see Algorithm 4.4). If a node x shall be recolored to a color c and c is admissible for that node, all its interference neighbors are scanned for color c. All neighbors of x having color c are then recolored to a color different from c. Thereby, we keep all recolored nodes in a list and remember the old color so that the recoloring can be rolled back if c is not feasible for x. This transactional character ensures the correctness of the approach since the validity of the coloring is always

Algorithm 4.3 Coalescing

```
procedure Recolor-Chunk(IG G, Chunk C, Queue Q)
    best_costs ← 0
    for all c ∈ R do                                   ▷ For all colors
        Unfix all nodes in chunk
        for all x ∈ C do                               ▷ Try to bring all nodes to c
            Recolor(G, x, c)
            fixed(x) ← true
        M ← best affine subset colored to c
        if |M| ≥ best_costs then
                                   ▷ Memorize the best color/costs/component
            best_color ← c
            best_costs ← |M|
            best_set ← M
    Unfix all nodes in the chunk
    for all x ∈ best_set do
                ▷ Bring all nodes of the best component to the best color
        Recolor(G, x, best_color)
        fixed(x) ← true
    rest ← best_set \ C                    ▷ Make a new chunk out of the rest
    if rest ≠ ∅ then
        v ← some node of rest
        v.chunk ← v
        for all x ∈ C \ best_set do
            x.chunk ← v.chunk
        Add v.chunk to Q


procedure Coalesce(Interference Graph G = (V, E, A, c))
    Build-Chunks(G)
    priority queue Q ← ∅
    for all chunks C do
        Insert C into Q
    while Q is not empty do
        C ← pop(Q)
        Recolor-Chunk(G, C, Q)
```

Algorithm 4.4 Recoloring

```
procedure Set-Color(Node x, Color c, Set of nodes N)
    fixed(x) ← true                          ▷ Mark the node as fixed
    old_color(x) ← ρ(x)                      ▷ Remember its old color
    N ← N ∪ {x}                              ▷ Add it to the changed set
    ρ(x) ← c                                 ▷ Set the new color

function Avoid-Color(IG G, Node x, Color c, Set of nodes N)
    if ρ(x) ≠ c then return true
    r ← false
    if ¬fixed(x) ∧ admissible(x) ∖ {c} ≠ ∅ then
        Select adm. color c′ ≠ c which is used least by x's neighbors
        Set-Color(x, c′, N)
        r ← true
        for all {y | xy ∈ E_G} do
            r ← Avoid-Color(G, y, c′, N)
            if r = false then
                break
    return r

procedure Recolor(IG G = (V, E, A, c), Node x, Color c)
    if c is admissible for x and ¬fixed(x) then
        N ← ∅          ▷ This set stores all nodes whose color is changed
        Set-Color(x, c, N)                   ▷ Set the color of x to c
        for all {y | xy ∈ E} do              ▷ Look at all neighbors of x
            if ρ(y) = c then                 ▷ If one has also color c
                r ← Avoid-Color(G, y, c, N)  ▷ Try to recolor it
                if r = false then            ▷ If that failed, recoloring x failed
                    for all v ∈ N do         ▷ Rollback to the former coloring
                        ρ(v) ← old_color(v)
        for all v ∈ N do
            fixed(v) ← false
```

retained. Furthermore, to guarantee termination, we do not change the color of a node again if it has already been changed in the current recoloring effort.

Algorithm 4.5 Building Chunks

```
procedure Build-Chunks(Interference Graph G = (V, E, A, c))
    for v ∈ V do
        v.chunk ← v
    for each a = xy ∈ A sorted according to c from large to small do
        if ¬∃vw ∈ E : v.chunk = x.chunk ∧ w.chunk = y.chunk then
            y.chunk ← x.chunk
```

4.5.2.1 Mimicking the Interference Graph

Budimlić et al. [2002] already used the Lemmas 4.2–4.4 for checking interference of two variables (nodes) without building the IG itself. However, using only these lemmas can generate false positives since they are only a necessary but not a sufficient condition for interference. Here, we present an enhancement which provides exact interference information and a method to list all interference neighbors of some node x. Therefore, we make use of the def-use chains, i.e. the list of all uses of each variable. To sum up, we require the following information to be already computed:

- The dominance relation.
- Liveness information between basic blocks, i.e. all variables live in and live out of a basic block.
- A list of all uses for each variable.

This information is often used in modern optimizing compilers and likely to be computed a priori. Note that since coalescing does not modify the program this information is static and does not have to be updated or recomputed.

Interference Test

Let us assume we want to check whether x interferes with y. Lemma 4.3 clearly shows that the definitions of x and y are always dominance-related if they interfere. So, if neither $\mathcal{D}_x \preceq \mathcal{D}_y$ or $\mathcal{D}_y \preceq \mathcal{D}_x$, x and y cannot interfere. Let us assume without loss of generality that $\mathcal{D}_x \preceq \mathcal{D}_y$. Note, that $\mathcal{D}_x \preceq \mathcal{D}_y$ is necessary but not sufficient for interference. Thus, we use Lemma 4.4 to

find the definitive answer. Thus, we have to check if x is live at $\mathcal{D}_y$. Firstly, we make use of the computed liveness information. If x is not killed in the block of $\mathcal{D}_y$ it is live at the end of that block if x and y interfere. If x is not live at the end of $\mathcal{D}_y$'s block, we have to check if there is a usage of x which is (strictly) dominated by $\mathcal{D}_y$. Algorithm 4.6 shows the pseudocode of the interference check.

Algorithm 4.6 Interference Check

```
function Interference-Check(program P, variables x, y)
    if D_y ≼ D_x then
        t ← y
        b ← x
    else if D_x ≼ D_y then
        t ← x
        b ← y
    else
        return false
                                        ▷ From now on holds D_t ≼ D_b
    if t is live out at the block of D_b then
        return true

    for all uses ℓ of t do
        if D_b ≼ ℓ then
            return true

    return false
```

Remark 4.4: Theoretically, the interference check is expensive since a variable could have many uses. However, measurements in our compiler shows an average use count of 1.7 per variable. Hence, the loop body in the interference check algorithm is not executed very often.

Enumerating all Neighbors

The second important operation on interference graphs is to list all interference neighbors of a given node x. Again, we use the results of Section 4.1 to circumvent the explicit materialization of the graph. As the live range of x is restricted to the dominance subtree of $\mathcal{D}_x$ (cf. Lemma 4.2) we only have to consider that part of the program. The set of interfering nodes can be

obtained by a small modification to the SSA liveness algorithm presented by [Appel and Palsberg, 2002, Page 429]. Consider Algorithm 4.7. All variables which are simultaneously live with x are recorded in a set N. This is done by starting a depth first search on the control flow graph from each usage of x upwards to $\mathcal{D}_x$. By walking over the labels in the blocks a neighbor might be added redundantly to N as it might occur multiple times in the argument arrays of different labels. As these sets are usually bitsets, these redundant additions are not very costly. However, for large programs one can compute the neighbors for each node once and cache them using adjacency lists.

Algorithm 4.7 Enumerating all Neighbors

```
procedure Int-Neighbors(variable x, set of variables N)
    N ← ∅
    visited ← ∅
    for all uses ℓ of x do
        Find-Int(block of ℓ, x, N, visited)

procedure Find-Int(block B = ⟨ℓ_1, ..., ℓ_n⟩, variable x, set of variables N, set
of labels visited)
    visited ← visited ∪ B
    L ← liveout(ℓ_n)
    ℓ ← ℓ_n
    while ℓ ≠ ℓ_1 ∧ ℓ ≠ D_x do
        L ← L ∖ res(ℓ)
        L ← L ∪ arg(ℓ)
        if x ∈ L then
            N ← N ∪ L
        ℓ ← pred(ℓ, 1)
    for all ℓ ∈ pred_{ℓ_1} do
        if D_x ⪯ ℓ ∧ block of ℓ ∉ visited then
            Find-Int(block of ℓ, x, N, visited)
```

4.5.2.2 A Note on Memory Variable Coalescing

As mentioned in Section 4.4.2, the coalescing heuristic presented here can also be used to coalesce spill slots. In fact, it is sufficient to run only the Build-Chunks procedure as shown in Algorithm 4.5. This will subdivide the memory variables into chunks. As there are at least as many spill slots as variables,

each resulting chunk can be assigned a different spill slot. Hence, recursive recoloring is not necessary for memory variable coalescing.

4.5.3 Optimal Coalescing by ILP

In this section we describe a method yielding optimal solutions for SSA-Coalescing using integer linear programming (ILP). For readers not familiar with basic ILP terminology, we refer to Appendix B. Firstly, we give a standard problem formulation for SSA-Coalescing. Then, we present further refinements of the formulation to prune the search space for the ILP solver. This work has been contributed by Grund [2005] in his diploma thesis and is presented in Grund and Hack [2007].

4.5.3.1 Formalization

Let $\mathcal{I}_P = (V, E, A, c)$ be the interference graph of some program P. Since every solution of SSA-Coalescing is also a valid coloring, we start by modeling a common graph coloring problem. For each $v_i \in V$ and each possible color c the binary variables x_{ic} indicate the color of node v_i: if $x_{ic} = 1$ the variable x_i has color c. The following constraint enforces that each node has exactly one color:

$$\forall v_i \in V : \sum_c x_{ic} = 1$$

Furthermore, incident nodes must have different colors. Normally, constraints like $x_{ic} + x_{jc} \leq 1$ are used for each edge $x_i x_j$ in the graph. For chordal graphs the number of these inequalities can be drastically lowered using a minimum clique cover:

Definition 4.5 (Minimum Clique Cover, MCC): *A minimum clique cover of a graph G is a minimal set of cliques $\{C_1, \ldots, C_m\}$ such that $V_G = V_{C_1} \cup \cdots \cup V_{C_m}$.*

For general graphs finding a MCC is NP-complete. However, for chordal graphs a MCC can be computed in $O(|V|^2)$, see Gavril [1972] for example. Given a MCC $\{C_1, \ldots, C_m\}$, the edges in the graph can be modeled by clique inequations expressing that no two nodes in the same clique can have the same color. Thus, we add for each C_j in the MCC and each color c:

$$\sum_{v_i \in C_j} x_{ic} \leq 1$$

Register constraints (as discussed in Section 4.6) can be integrated in the model by either setting the forbidden x_{ic} to 0, or by omitting the corresponding variables and adjusting the respective constraints accordingly.

So far, the model results in a valid coloring of G. Now we define the cost function and add additional variables and constraints to model the affinities. For each edge $v_iv_j \in A$ the binary variable y_{ij} shall indicate, whether the incident nodes have the same color ($y_{ij} = 0$), or different colors ($y_{ij} = 1$). To model the positive costs $c(v_iv_j)$ arising from assigning v_i and v_j different colors, the cost function to be minimized is given by

$$\sum_{v_iv_j \in A} c(v_iv_j) \cdot y_{ij}$$

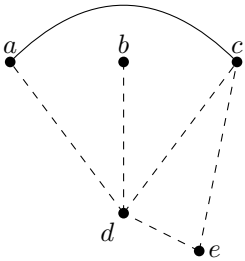
To relate the y to the x variables, we add a constraint per color and affinity edge in A. Since the y_{ij} are automatically forced towards 0 due to minimization, we only must enforce $y_{ij} = 1$, if v_i and v_j have different colors:

$$\begin{aligned} y_{ij} &\geq x_{ic} - x_{jc} \\ y_{ij} &\geq x_{jc} - x_{ic} \end{aligned}$$

Our measurements (see Grund and Hack [2007]) showed that the ILP solving process takes very long if the solver is provided solely the equations presented above. Therefore, we present two improvements to the ILP formulation which significantly decrease the solution time of the ILP. The first improvement aims at reducing the *size* of the problem by modelling only "relevant" nodes of the IG in the ILP. This is achieved by exploiting a simple graph theoretical observation. The second improvement *increases* the size of the problem by adding additional inequalities. However, these new inequalities are so called *cuts*, i.e. they do not rule out any feasible solutions but shrink the search space. One could say, they provide the solver with more knowledge about the problem's structure to enable it to detect and discard invalid solutions earlier.

4.5.3.2 Reducing the Problem Size

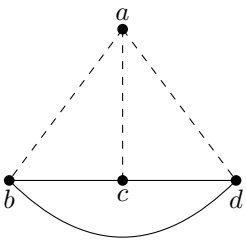
To reduce the number of variables and constraints, we use the same observation as the coloring heuristic presented in Section 3.1: if a node has less than k neighbors, it definitely can be colored. Hence, before setting up the ILP, we remove successively all nodes from G, which have less than k neighbors and do exhibit neither affinities nor register constraints. The remaining graph G' then represents the "core" of the problem. The coloring for G' found by the ILP solver can then be completed by adding the removed nodes again.



$y_{ad} + y_{cd} \geq 1$

$y_{ad} + y_{de} + y_{ec} \geq 1$

(a) Path-inequalities



$y_{ad} + y_{bd} + y_{cd} \geq 2$

(b) Clique-Path-inequality

Figure 4.8: Examples for the two classes of inequalities.

4.5.3.3 Additional Inequalities

Finally, we present two types of inequalities which can be used to prune the search space further: *Path-* and *Clique-Ray*-cuts. The basic idea of both types is to provide the solver with lower bounds for the costs by modelling facts about the interplay of affinity- and interference edges.

Path-Cuts

The first kind of cuts we are adding to the formulation is based on the observation that if there is a path of affinity edges between two interfering nodes, at least one of the affinity edges cannot be fulfilled as both nodes interfere. Consider the example in Figure 4.8a. As a and c interfere, at least the affinity ad or the affinity dc cannot be fulfilled.

Definition 4.6: *Let $G = (V, E, A, c)$ be an interference graph. We call two nodes $a, b \in V$* affinity violating *if a and b interfere and there exists a path $P = (V_P, A_P)$, $V_P = \{a, \ldots, b\}$ of affinity edges such that the only interference of nodes in the path is between a and b, i.e. $G[V_P] = (V_P, \{a, b\})$*

Let v_1 and v_n be affinity violating and $P = v_1 \ldots v_n$ be an affinity violating path. Note that in general there can exist multiple such paths. As v_1 and v_n interfere, at least one affinity on P can not be fulfilled which is represented

by the following inequality:

$$\sum_{i=1}^{n-1} y_{i,i+1} \geq 1$$

Clique-Ray-Cuts

The second type of inequality is based on a clique $C = \{v_1, \ldots, v_n\}$ in the interference graph combined with a node $v_m \notin C$. If v_m is affinity related with each v_i in C, only one of the affinities can be fulfilled, i.e. $n-1$ affinities will be violated. Consider the example in Figure 4.8b. a wants the same color as b, c and d. As b, c, d interfere they have different colors so at most one of the affinities ab, ac, ad can be fulfilled. This is reflected in following *Clique-Ray*-cut:

$$\sum_{i=1}^{n} y_{im} \geq n - 1$$

Adding both kinds of constraints decreases the solution time of the problems significantly as shown in Grund and Hack [2007]

4.6 Register Targeting

So far, we considered only *homogeneous register sets*; each register was assignable to each variable at each label. In reality however, there are instructions imposing constraints on the set of allocatable registers of their results and/or arguments. Furthermore, the runtime system often restricts register assignment on several occasions such as subroutine calls . Fulfilling these constraints is not a matter of optimization but required to ensure the correctness of the allocation. Section 5.1.2 gives several examples of constrained instructions and runtime system conventions.

To formally express the constraints imposed by the processor/runtime-system, we consider two maps

$$\begin{aligned} \mathcal{A}^{arg} &: L \to \mathscr{L}(2^R) \\ \mathcal{A}^{res} &: L \to \mathscr{L}(2^R) \end{aligned}$$

which assign each argument and result of each label a set of admissible registers. For example, if $\mathcal{A}^{arg}(\ell, 2) = \{R_1, R_2\}$, then the variable used at ℓ's second argument has to be assigned to R_1 or R_2.

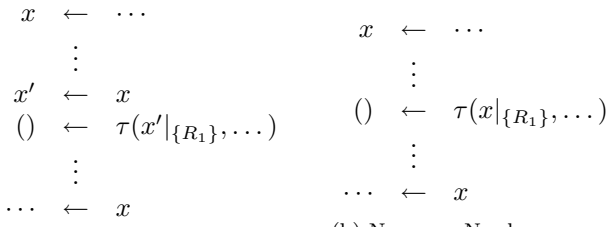
As shown in Section 3.1.5.1, register constraints can lead to uncolorable interference graphs due to the unauspicious interaction between interference and targeting constraints. This problem is commonly solved by minimizing the constrained live ranges by inserting copies. The constraint is transferred to the copy, so that the live range itself remains unconstrained. The live range of a constrained copy remains restricted to the label that caused the constraints.

In the case of arguments, this method might result in additional register demand: if an argument x of some label ℓ lives through ℓ, it also interferes with its copy which was created due to the constraint at ℓ. However, it could have been possible to assign x a register fulfilling the constraint directly, saving the copy *and* not increasing the register pressure as shown in Figure 4.9.

To handle register targeting constraints appropriately in an SSA-based register allocator, three steps are necessary:

1. As shown above, label- or operation-based constraints can produce programs that do not possess a valid register allocation. Thus, the program has to be modified to make register allocation possible. In the course of doing so, the operation-based constraints are transformed to variable-based ones as registers are assigned to variables not to operations.

2. The register demand of the constrained instruction has to be modelled precisely by register pressure to keep Theorem 4.3 valid.



(a) x and copy x' interfere. Need two Registers

(b) No copy. Need one Register

Figure 4.9: Insertion of copies causes additional register demand

3. The coloring algorithm (Algorithm 4.2) has to be modified to respect the variable based constraints.

We start with step 1 by considering an isolated, arbitrarily constrained instruction in a single basic block. We show, that determining the copies which have to be inserted to make such a single instruction allocatable is NP-complete. Consequently, we investigate a restriction on the constraints which allows for an efficient assignment for such operations. Afterwards, we consider step 2 and examine which conditions have to hold such that register pressure always correctly reflects the register demand. At last, we discuss how these results concerning a single constrained instruction can be transferred to a program with multiple constrained instructions.

4.6.1 Copy Insertion

Let us start by investigating the simplest case and restrict ourselves to a program P with a single basic block and a single constrained label ℓ with arbitrary constraints located right after the first label *start* of P. For now, we will only consider the label ℓ and how we can find a valid register allocation for ℓ's arguments and results.

The first eye-catching oddity is that the same variable x can be used multiple times at ℓ but under different constraints as shown in Figure 4.10a. This imposes unsatisfiable constraints to the variable x as x shall assigned to both R_1 and R_2. To enable variable-based constraints, x has to be copied to another variable x'. x is then constrained to R_1 and x' to R_2 (or vice versa)

$$\cdots \leftarrow \tau(x|_{\{R_1\}}, x|_{\{R_2\}})$$

(a) Same variable, different constraints at two arguments

$$\begin{aligned} x' &\leftarrow x \\ \cdots &\leftarrow \tau(x|_{\{R_1\}}, x'|_{\{R_2\}}) \end{aligned}$$

(b) One constraint per variable

Figure 4.10: Uncolorable program and transformed version

as shown in Figure 4.10b. Afterwards, there will be at least one admissible register for each argument of ℓ.

The next obstacle are variables living through the constrained label, i.e. they are live in and live out at that label. Consider the following situation:

A variable x lives through ℓ and its set of admissible registers is not disjoint with the set of admissible registers of the (probably constrained) results. It might be necessary to insert a copy for x since there will be no admissible register free for x depending on the admissible registers of the results. For example, for an instruction

$$y|_{\{R_1\}} \leftarrow \tau(x|_{\{R_1\}}, z)$$

and x interfering with y, x has to be copied to some temporary which then is consumed by ℓ. Unfortunately, unless P = NP, there is no efficient algorithm to decide whether there is a register allocation for ℓ without inserting a copy. We prove this by reducing from the NP-complete problem Sim-P-Match. Consequently, determining the minimum number of copy instructions to make ℓ assignable is NP-complete, too.

Definition 4.7 (Sim-P-Match): *Given a bipartite graph $G = (X, Y, E)$ and a collection $\mathcal{F} \subseteq 2^X$ of subsets of X. Is there a subset $M \subseteq E$ of edges such that for each $C \in \mathcal{F}$ the set of edges $M \cap (C \times Y)$ is a C-perfect match in G?*

Elbassioni et al. [2005] prove Sim-P-Match NP-complete even for $|\mathcal{F}| = 2$.

Theorem 4.6: *Register allocation of a basic block and a single, arbitrarily constrained label is NP-complete.*

Proof. Let $G = (X, Y, E)$ be a bipartite graph and $\mathcal{F} = \{F_1, F_2\} \subseteq 2^X$ a

family of subsets of X. Y coincides with the set of registers. Construct ℓ in the following way:

- $arg_\ell = F_1$, $res_\ell = F_2 \setminus F_1$.
- Let $N(x) = \{y \mid xy \in G\}$ be the set of neighbors of x.
- Set $\mathcal{A}^{arg}(\ell, i) = N(x)$ if $arg(\ell, i) = x$ or $\mathcal{A}^{res}(\ell, i) = N(x)$ if $res(\ell, i) = x$.
- if $x \in F_1 \cap F_2$, then x shall live through ℓ, i.e. we add a label ℓ' after ℓ and make ℓ' use x. Thus x interferes with each $y \in res(\ell)$.

A valid register assignment of all arguments and results of ℓ directly corresponds to a solution of Sim-P-Match. Figure 4.11 gives an example for a Sim-P-Match instance and the program resulting from the reduction. □

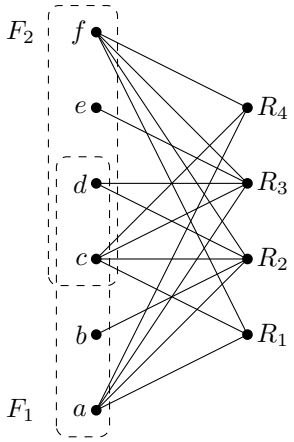
However, the case where the constraints are completely arbitrary is rare in practice. More common are situations where the variables are either unconstrained or limited to a single register.

Definition 4.8 (Simple Constraint Property):

1. *Each argument / result is either unconstrained or constrained to a single register:*
$$\mathcal{A}^x(\ell, i) = R \text{ or } |\mathcal{A}^x(\ell, i)| = 1$$
2. *Two arguments / results may not be constrained to the same register:*
$$1 = |\mathcal{A}^x(\ell, i)| = |\mathcal{A}^x(\ell, j)| \implies \mathcal{A}^x(\ell, i) \cap \mathcal{A}^x(\ell, j) = \emptyset$$
$$\text{for } x \in \{arg, res\} \text{ and } 0 \leq i < j \leq |x_\ell|$$

Under this assumption, the amount of copies is efficiently determinable: if a live-through variable is constrained to a single register, it has to be copied if and only if there is a result being constrained to the same register. From now on, we assume that each instruction fulfils the simple constraint property. Instructions for which this property does not hold a priori can be prepared in the following way: consider the argument and result constraints separately. For each constrained argument / result find an admissible register and consider it as a simple constraint. This can be done by computing separate bipartite matchings for arguments to admissible registers and results to admissible registers. Since this is only a heuristic, superfluous copy instructions could be inserted. However, this should be rarely the case in practice.

We can now transfer the constraint annotation from the label to the variables of P. Therefore, we define a map $\alpha : V \to 2^R$ as follows:



(a) Bipartite Graph G with two sets $F_1 = \{a, b, c, d\}$ and $F_2 = \{c, d, e, f\}$

$$
\begin{array}{rcl}
(a, b, c, d) & \leftarrow & \\
(e|_{\{R_3\}}, f) & \leftarrow & \tau(a, b|_{\{R_2\}}, c, d|_{\{R_2, R_3\}}) \\
 & \leftarrow & \tau(c, d, e, f)
\end{array}
$$

(b) Program generated from G

Figure 4.11: A Sim-P-Match instance and the generated Program

- Let $I = \{i_1, \ldots, i_n\}$ be the set for which $x = arg(\ell, i)$ for all $i \in I$. The copies we inserted ensure that there is at least one admissible register for x, i.e.

$$A = \bigcap_{i \in I} \mathcal{A}^{arg}(\ell, i) \neq \emptyset$$

  We thus set $\alpha(x) = A$.

- If $\mathcal{D}_x = \ell$ there exists exactly one i with $res(\ell, i) = x$ since P is in SSA form. We thus set $\alpha(x) = \mathcal{A}^{res}(\ell, i)$.

- Otherwise $\alpha(x) = R$

4.6.2 Modelling Register Pressure

Up to now, we only considered the copies needed to produce a valid coloring. By enforcing the simple constraint property this can be done efficiently. To be able to keep Theorem 4.3 valid, we furthermore have to ensure that the register *pressure* at constrained labels exactly resembles the register *demand*. In the presence of constraints this is not always the case, as shown in Figure 4.12a. The label on the left of that figure requires two registers although the register pressure is one. This leads us directly to the definition of a *register pressure faithful* label:

$$y|_{\{R_1\}} \leftarrow \tau(x|_{\{R_2\}})$$

(a) Non register pressure faithful label

$$\begin{aligned} z &\leftarrow \cdots \\ y|_{\{R_1\}} &\leftarrow \tau(x|_{\{R_2\}}, z) \end{aligned}$$

(b) Fixed version

Figure 4.12: Register Pressure Faithful Instructions

Definition 4.9 (Register Pressure Faithful Label): *A label ℓ is register pressure faithful if there is a register allocation of ℓ's arguments and results with $\max\{|res(\ell)| + t, |arg(\ell)|\}$ registers where t is the number of arguments of ℓ living through ℓ.*

Clearly, the label in Figure 4.12a is not register pressure faithful. However, such a situation should seldomly appear in practice. Nevertheless, one can add dummy arguments to the label to fix this as shown in Figure 4.12b.

4.6.3 Interference Graph Precoloring

If we simplify the constraints as described in the above section, the program consisting of a single basic block and a single constrained label can be colored efficiently. The next step is to investigate more complex programs with more basic blocks and more constrained labels. Again, we benefit from recent results of graph theory. The simple constraint problem directly corresponds to the *Precoloring Extension Problem.*

Definition 4.10 (Precol-Ext): *Given a graph $G = (V, E)$ where some nodes are already colored. Can we extend this precoloring to a complete k-coloring of G?*

Biró et al. [1992] prove that Precol-Ext is NP-complete for interval graphs and thus for chordal graphs. Thus, register allocation in presence of simple constraints is NP-complete. Hence, we cannot generalize the results of the last section to the whole program. However, a relaxed version of Precol-Ext called 1-Precol-Ext which restricts the number of precolored nodes to one per color is polynomially solvable for chordal graphs as shown by Marx [2004].

As generally the same constrained operation can occur multiple times in a program, the number of nodes precolored with the same color is generally larger than one. This renders the constrained coloring problem NP-complete again. As we are interested in efficiently coloring the graph, we transform the program in a way that it fulfils the properties of 1-Precol-Ext. As the following theorem shows live range splitting provides exactly what is needed:

Theorem 4.7: *By splitting the live ranges of all variables live at the entry to ℓ the interference graph is separated into two disconnected components.*

Proof. We will show that there is no interference edge from any variable whose definition is dominated by ℓ to any variable whose definition is *not* dominated by ℓ. Consider the interference graph $\mathcal{I} = (V, E, \dots)$ of the program. Assume some x with $\ell \preceq \mathcal{D}_x$ and some y with $\ell \not\preceq \mathcal{D}_y$. Further assume there is an edge $xy \in E$. Then either $\mathcal{D}_x \preceq \mathcal{D}_y$ or vice versa due to Lemma 4.3. If $\mathcal{D}_x \preceq \mathcal{D}_y$ then also $\ell \preceq \mathcal{D}_y$ due to the transitivity of $\preceq$. Hence, $\mathcal{D}_y \preceq \mathcal{D}_x$. Because $\ell \not\preceq \mathcal{D}_y$ there is $\mathcal{D}_y \preceq \ell$ due to the tree property of $\preceq$ as described in Section 2.2. Due to Lemma 4.4, y is live at $\mathcal{D}_x$. Since $\ell \preceq \mathcal{D}_x$ there is a path from ℓ to some usage of y. Thus y is live at ℓ. This contradicts the premise that all live ranges of variables live at ℓ have been split. Hence, all variables defined at labels dominated by ℓ form a separate component in the interference graph. □

Hence, each connected component of the interference graphs contains exactly one simple-constrained label. By splitting all the live ranges at ℓ, we delegate all the complexity of Precol-Ext to the coalescing phase (as discussed in Section 4.5) which will try to remove the overhead caused by the live range split.

The live range split before ℓ is done be inserting a Φ^r-Φ^w-pair using each variable which was at the entry to ℓ and defining new versions of these. Note that splitting all live ranges introduces copies of existing variables which are not yet used in the program. Let $x_1, \ldots, x_n$ be the set of variables live at the entry to ℓ. By inserting a Φ^r-Φ^w-pair

$$\begin{array}{lcl} & & \Phi^r(x_1, \ldots, x_n) \\ (x'_1, \ldots, x'_n) & \leftarrow & \Phi^w \end{array}$$

for each x_i a copy x'_i is inserted. Thus, we might need to re-wire some of the uses of x_i to x'_i. Note that this can even cause the insertion of additional Φ-operations. From another point of view, one could argue that x'_i is just another definition of x_i and x_i has now multiple definitions violating the single assignment property. Hence we apply the SSA reconstruction algorithm presented in Section 4.2.1 to re-install SSA and re-wire each use of x_i to the correct definition.

4.6.4 Coloring

Finally, let us discuss how we can obtain a valid coloring of the graph component with the constrained label ℓ. The label ℓ_s immediately preceding ℓ is the one containing the Φ^w-operation as discussed above. As stated by Theorem 4.7, ℓ_s dominates all other definitions of variables in the interference graph component. Thus, due to Theorem 4.4 there is a PEO of the graph's component ending in $res_{\ell_s} = \langle x'_1, \ldots, x'_n \rangle$. As the constrained label comes right after ℓ_s the tail of the PEO is $res(\ell), res(\ell_s)$. This means that the results of the ℓ_s and ℓ can be colored first. Note that all arguments to ℓ are results of ℓ_s and are thus colored before ℓ's results.

The following preconditions are met by doing so:

- ℓ fulfils the simple constraint property.
- ℓ is register pressure faithful.
- Spilling lowered the register pressure to at most k everywhere.
- The results of ℓ_s and ℓ can be colored first for the graph component under consideration.

We thus assign all constrained variables their color. Note that the simple constraint property restricts a constrained variable to a single register. Further note that the register pressure truth guarantees that there is a valid register allocation of ℓ's arguments and results with $r = \max\{|res(\ell)| + t, |arg(\ell)|\}$ registers. As the register pressure before ℓ is at least $|arg(\ell)|$ and after ℓ at least $t+|res(\ell)|$, there can only be $k-r$ variables live through ℓ without being used by ℓ. As ℓ needs at most r registers there will be enough registers for all variables living by ℓ. So our only task is to find a register allocation of ℓ needing no more than r registers.

Algorithm 4.8 computes the initial coloring at a constrained label. Firstly, all constrained variables at ℓ are colored with their designated color. Afterwards, all arguments dying at ℓ are colored. Thereby, colors used by the constrained results are re-used if possible (not already in use by an argument). The same applies to the remaining unconstrained results. At last, all arguments live through ℓ (set T) have to be colored with colors different from those already used, since they interfere with both the dying variables (set A) and the variables defined (set D).

To complete the coloring for the labels ℓ_s and ℓ, all variables defined by the Φ^w at ℓ_s which are not used at ℓ have to be colored with colors not used by any argument or result of ℓ. The rest of the component can be colored with Algorithm 4.2 starting with the label after ℓ.

4.6.5 An Example

To demonstrate the process of register targeting in an SSA-based register allocator, we consider a call instruction on the x86 platform. Processors of the x86 family provide seven general purpose registers named `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp`. The GNU C compiler provides a fast call extension by which the first two (integer) parameters to a function can be passed in the registers `ecx` and `edx`. All other parameters are passed on the stack. The registers `eax`, `ecx`, edx may be overwritten by the callee, i.e. they are caller-save. The other registers have to be saved by the callee if necessary, i.e. they are callee-save.

For functions returning an integer (which we assume for this example), the result of the function call is passed in `eax`. As the caller has to assume that the contents of the caller-save registers are destroyed upon returning from the call, they can be treated as if they were additional call results. Hence, we insert two dummy variables d_1 and d_2 to model the destruction of `edx` and `ecx` after the call. Let us furthermore assume that the call is indirect, i.e. the called address is passed in a register which is *not* constrained. Consider

Algorithm 4.8 Coloring at a constrained Label

```
procedure Constrained-Coloring(label ℓ)
    T ← {x | x ∈ arg(ℓ) ∧ x lives through ℓ}
    A ← arg(ℓ) ∖ T
    D ← res(ℓ)
    C_R ← ∅
    C_A ← ∅
    R' ← R

    for all constrained arguments x do
        C_A ← C_A ∪ α(x)                    ▷ Mark the register as occupied
        A ← A ∖ x
        T ← T ∖ x                           ▷ Mark the argument as assigned
        ρ(x) = α(x)                         ▷ Set its color
        if x ∈ T then
            R' ← R' ∖ ρ(x)
    for all constrained results x do        ▷ The same for the results
        C_D ← C_D ∪ α(x)
        D ← D ∖ x
        ρ(x) = α(x)

    for all x ∈ A do                        ▷ The rest arguments are not constrained
        if C_D ∖ C_A ≠ ∅ then               ▷ Try to use a color of the results
            ρ(x) ← one of C_D ∖ C_A
        else                                ▷ If there is none, take a fresh one
            ρ(x) ← one of R' ∖ C_A
    for all x ∈ D do                        ▷ Same for the results
        if C_A ∖ C_D ≠ ∅ then
            ρ(x) ← one of C_A ∖ C_D
        else
            ρ(x) ← one of R' ∖ C_D

    Assign registers to T from the set R' ∖ (C_D ∪ C_A)
```

following program fragment. It shows the call instruction with its address argument (variable a) and the two arguments to the called function (x, y).

$$\begin{array}{rcl} x & \leftarrow & \cdots \\ y & \leftarrow & \cdots \\ & \vdots & \\ (z|_{\{\texttt{eax}\}}, d_1|_{\{\texttt{ecx}\}}, d_2|_{\{\texttt{edx}\}}) & \leftarrow & \texttt{call}(a, x|_{\{\texttt{ecx}\}}, y|_{\{\texttt{edx}\}}) \\ & \vdots & \\ \ldots & \leftarrow & \tau(z, x) \end{array}$$

The result of the call and one argument are used later on by τ. Thus, one argument lives through the call. Obviously, the call fulfils the simple constraint property since each variable is either unconstrained or restricted to a single register. It is easy to see that a copy has to be inserted for argument x since it is used later on. This is because the register `ecx` which x is supposed to be in, is destroyed by the call itself. Below is the program fragment with the inserted copy.

$$\begin{array}{rcl} x & \leftarrow & \cdots \\ y & \leftarrow & \cdots \\ & \vdots & \\ x' & \leftarrow & x \\ (z|_{\{\texttt{eax}\}}, d_1|_{\{\texttt{ecx}\}}, d_2|_{\{\texttt{edx}\}}) & \leftarrow & \texttt{call}(a, x'|_{\{\texttt{ecx}\}}, y|_{\{\texttt{edx}\}}) \\ & \vdots & \\ \cdots & \leftarrow & \tau(z, x) \end{array}$$

This copy has even made the call instruction register pressure faithful since there is a register allocation of its arguments with

$$\underbrace{|res(\ell)|}_{=3} + \underbrace{t}_{=0} = 3$$

One such allocation is:

z	`eax`	a	`eax`
d_1	`ecx`	x'	`ecx`
d_2	`edx`	y	`edx`

5 Implementation and Evaluation

We conducted various measurements to show the competitiveness of the register allocator presented in this thesis. Since the pre-spilling heuristics as presented in Section 3.1.5.2 are sufficient, our allocator will never introduce additional spill code after spilling has happened. Thus, concerning spilling, the presented allocator is at least as good as any other allocator available, since it can use the same register-pressure-based spilling methods. The main question arising from this paradigm, is whether the live range splitting implied by Φ-operations and register constraints causes significant performance penalties. Thus, we chose the x86 architecture for measurements since it is known to possess many irregularly constrained instructions. Therefore , we expect a large amount of live range splits due to these constraints.

Before presenting the results of the experimental investigation, we describe the compiler used to conduct the measurements.

5.1 Implementation

We implemented the register allocator described in the previous chapter into the C compiler developed at the *Institut für Programmstrukturen und Datenorganisation* at the *Universität Karlsruhe*. This compiler is built upon *Firm* (see Lindenmaier et al. [2005]), a graph-based SSA intermediate representation which is similar to the *Sea of Nodes* IR described by Click and Paleczny [1995].

5.1.1 The *Firm* Backend

Based on the IR *Firm*, we implemented a completely SSA-based backend. After all architecture independent optimizations in the compiler's "middle end" have been finished, the code represented by data and control flow graphs

gets lowered successively. Firstly we implement the rules of the application binary interface (ABI). This comprises building stack frames, deciding which parameters of function calls will reside in registers or on the stack and so on. Afterwards, the architecture independent IR nodes are replaced by nodes representing machine instructions.

As register allocation requires a totally ordered instruction sequence to compute liveness (and thus interference) correctly, the data dependence graphs inside the basic blocks are scheduled using a common latency based list scheduling algorithm as described in Muchnick [1998].[1]

Now register allocation takes place. The allocation is performed separately for each register class. As described in the previous chapter, the register allocator runs in several phases:

1. Copy instructions for constrained instructions are inserted to make the register pressure match the actual register demand (see Section 4.6).
2. Spilling is performed using the heuristic described in Section 3.1.5.2 and the book of [Morgan, 1998, page 319]. It is adapted using the methods described in Section 4.2.1.
3. Then live range splits for constraint handling are inserted (see Section 4.6.3) and registers are assigned to the variables (see Section 4.3)
4. Afterwards, coalescing runs as an (optional) optimization of the coloring, trying to remove as many move instructions as possible.
5. Φ-operations are implemented as described in Section 4.4 and the assembly text file is emitted.

5.1.2 The x86 Architecture

In this section we describe how the features of the x86 are displayed to the register allocator. Due to its long history, the x86 architecture exhibits significant differences compared to standard RISC architectures. We will briefly present the features which deserve special treatment:

1. x86 processors provide eight 32-bit integer registers. One register (`esp`) is reserved as the stack pointer and is hence excluded from register allocation. The other registers are called `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi` and `ebp`. Most instructions also allow to access the lower 16 bit of

[1] Future implementations could enhance the scheduling algorithm by heuristics which keep track of the register pressure as described in Hoxey et al. [1996].

`eax`,...,`ebp` through aliased registers `ax`,...,`bp`. The lower 16 bit of `eax`,...,`edx` can also be accessed in 8 bit portions through registers `al`,`ah`,...,`dl`,`dh`.

2. Almost all binary instructions use 2-address code, i.e. the first operand of the instruction is also the target of the instruction. Thus, if the first operand is assigned to register `eax` the result of the operation is written to `eax`.

3. Several instructions exhibit register constraints. For the argument as well as the result registers. Mostly, these constraints demand that an argument or result must reside in a specific register.

4. The floating point registers are organized as a stack. Most instructions implicitly use the top of the stack (TOS) as operand and result. Thus, registers cannot be assigned directly to floating point variables.

We address these irregularities as follows:

5.1.2.1 Aliased Registers

C dictates that all `char` and `short` operands are casted to `int` in integer computations and as an `int` is 32-bit wide in our compiler, most integer operations will be 32-bit operations. Thus, we do not support access to the aliased register parts of the integer registers. This sometimes leads to slightly slower code since narrowing results in additional mask instructions that clear the respective bits.

5.1.2.2 2-Address Code

If the operand variable assigned to the target register becomes dead at that label in question, 2-address code can be seen as a kind of code compression since the instruction word needs to store only two registers. But if that operand (let us call it the overwritten operand) lives past the instruction it has to be copied to the result register before the instruction.[2] However, this "trick" bears a subtle problem. Consider a non-commutative operation such as `sub` and a label

$$\ell : z \leftarrow \texttt{sub}(x, y)$$

[2]Note that these copies can be optimized away by the coalescing phase if they are not necessary.

where x lives through ℓ and y dies at ℓ. Assume, the register allocator produced a register allocation where $\rho(z) = \rho(y) = \texttt{eax}$ and $\rho(x) = \texttt{ebx}$. Then using the "copy trick" would result in the following code sequence destroying the contents of `eax` before it is used:

$$\begin{aligned} x' : \texttt{eax} &\leftarrow x : \texttt{ebx} \\ z : \texttt{eax} &\leftarrow \texttt{sub}(x' : \texttt{eax}, y : \texttt{eax}) \end{aligned}$$

Some instructions such as `sub` can be re-formulated to circumvent this difficulty. In the example above, one could write

$$\begin{aligned} x' : \texttt{eax} &\leftarrow \texttt{neg}(x : \texttt{eax}) \\ z : \texttt{eax} &\leftarrow \texttt{add}(x' : \texttt{eax}, y : \texttt{ebx}) \end{aligned}$$

thereby making the dying operand the overwritten one. However, for the most non-commutative instructions this is not possible. Therefore, for non-commutative operations we have to ensure that the result register is different from the register assigned to the overwritten operand. This is achieved by adding an artificial use after the label which forces this operand to live through the label. Then the not overwritten operand interferes with the result and never gets assigned the same register. Concerning our example above this yields the following code:

$$\begin{aligned} \ell : z &\leftarrow \texttt{sub}(x, y) \\ &\quad\ \ \texttt{use}(y) \end{aligned}$$

Note that this artificial use can be omitted if both operands are the same. Furthermore, if the overwritten operand dies at the label and the second one lives through it a third register is needed. Thus, 2-address code can cause additional register pressure.

5.1.2.3 Register Constraints

Almost all constrained instructions require that an operand (or result) must reside in *one* specific register. Examples are shown in the table below.

Instruction	Constraints
multiplication	one operand must be in `eax`; result is in `edx:eax`
division	64-bit operand and result must be in `edx:eax`
shift and rotate	shift count must be in register `ecx`
byte load	result is 8-bit register, i.e. one of `eax`, ..., `edx`

Besides the byte loads, all constrained x86 instructions only exhibit constraints to single registers. For byte loads, only the subset `eax`, ..., `edx` is admissible. To fulfill the single constraints property (see Section 4.6.1), one of these registers can be arbitrarily selected. However, our coalescing algorithms (see 4.5) can take all admissible registers into account.

5.1.2.4 Stack-based FPU

Leung and George [2001] describe an approach which allows to assign virtual registers to the floating point variables and turn them into FPU stack operands afterwards. Alternatively, on more modern incarnations of the x86 architecture, the SSE2 instruction set is available which allows to perform double precision floating point arithmetic in the lower halves of eight 128-bit vector registers.

5.2 Measurements

In principle, k-colorable chordal graphs can by colored optimally with the simplify/select-scheme.[3] Since the original allocator presented by Chaitin merges nodes in the coalescing phase, spills might be introduced in favor of copies, which is unacceptable. This cannot happen using conservative coalescing allocators. However, they remove far too few copies (see Section 3.1.1 and Appel and George [2001]) due to their conservative heuristic. We also experimented with Park and Moon's optimistic register coalescing (see Section 3.1.1) that is able to undo some coalescings. We found that it frequently inserted spills into the k-colorable chordal IGs. This is mainly because it can only undo the coalescing of the node that is currently to be colored. If the coloring for a node fails due to already colored coalesced nodes their coalescing cannot be undone. This results in current allocators either inserting spills in favor of copies or removing far too few copies to prevent spills from being inserted.

In comparison to existing, non-SSA register allocators our allocator will not add a single spill after a pressure-based spilling algorithm has run. Since these spilling heuristics are also applied by conventional register allocators, we never insert more spills than a conventional allocator using the same pressure-based spilling heuristic. Hence, our investigations concentrate on the overhead caused by the live range splitting due to Φ-operations, spilling and register

[3] A chordal graph always has a simplicial node and simplicial nodes have insignificant degree, so there is always a node available for elimination.

targeting, and on evaluating the performance of the coalescing heuristic presented in this thesis.

5.2.1 Quantitative Analysis of Coalescing

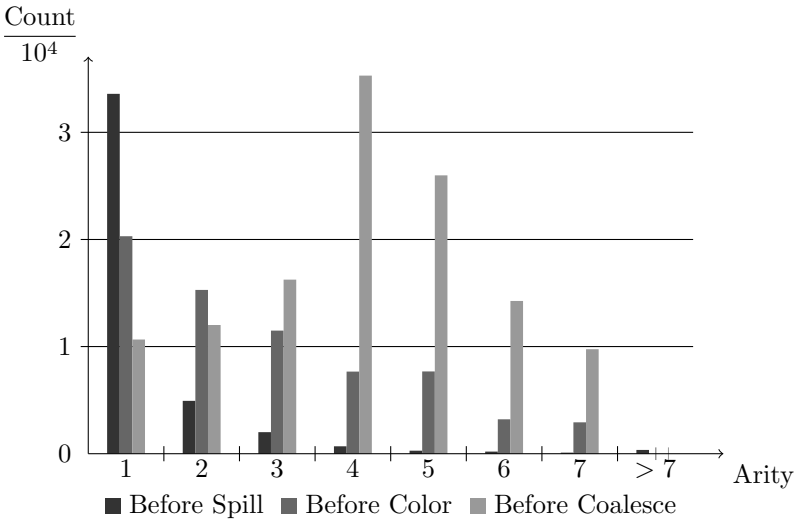
We applied the heuristic presented in Section 4.5.2 to all C programs in the integer part of the CPU2000 benchmark (see `http://www.spec.org/cpu2000`). This comprises all programs in `CINT2000` but `252.eon` which is C++. In total, 4459 functions were examined. Afterwards, we applied the ILP formulation to all those functions to compare the heuristic's results to the solutions found by the ILP solver.

5.2.1.1 Φ^r-Operations

Let us start with a general statistics of the test programs. The table in Figure 5.1 shows the total number of Φ^r-operations (broken down by their arity) of all compiled programs before spilling, before coloring and before coalescing. The numbers are presented in Figure 5.1. A Φ^r of arity three will result in three move instructions if the respective results and operands could not be assigned the same color. The "Before Spill" column gives the numbers of Φ-operations before any program modification by the register allocator took place. Spilling adds more Φ-operations (or extends existing ones by adding more columns). Obviously there are many Φ^r-operations with one operand initially. After the spilling phase there are more Φ^r-operations with multiple operands. The coloring phase, which splits live ranges to handle register constraints, inserts many new Φ^r-operations with about four or five operands. This is reasonable since they are used to split all live ranges in front of a constrained instruction, the resulting Φ^r-operation has all live variables as operands. Values between three and five are common for the register pressure when there are seven registers available. Obviously, the register allocator adds many live range splits. This is mostly due to the register constraints imposed by the x86 architecture. For machines with few constraints, the number of Φ^r-operations after spilling is more representative. Nevertheless, this makes a powerful coalescing approach indispensable.

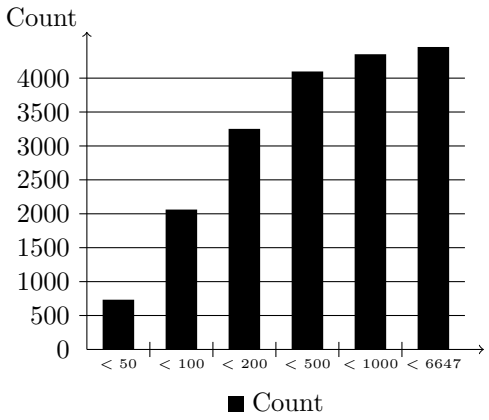
5.2.1.2 Interference Graphs

Most interference graphs were rather small having less than 500 nodes, as shown in Figure 5.2. The enormous amount of live range splitting is reflected in the interference graphs as shown in Figure 5.3: the column "$\varnothing$ Aff. Nodes"



Arity	Before Spill	Before Color	Before Coalesce
1	33596	20299	10656
2	4934	15292	12014
3	2009	11495	16248
4	692	7668	35294
5	280	7677	25986
6	200	3212	14261
7	99	2928	9753
> 7	350	0	0
Σ	42160	68571	124212

Figure 5.1: Number of Φ^r-operations



#Nodes	Count
< 50	734
< 100	2062
< 200	3252
< 500	4098
< 1000	4352
< 6647	4459

Figure 5.2: Interference Node Distribution

lists the average of all nodes having incident affinity edges. Comparing to the average number of nodes in the graph (given by column "$\varnothing|V|$"), always more than 50% of the nodes are involved with affinities. Moreover, the number of nodes inside a connected interference component is around ten (see column "$\varnothing$ Comp. Size"). The average degree concerning affinity edges (column "$\varnothing\Delta_{\text{Aff}}$") is around two which reflects the breaking of live ranges in basic blocks by constraint handling. This is also nicely displayed in the graph in Figure 5.5.

Concerning copies to eliminate/minimize we discriminate between two parameters: the number of satisfied affinity edges, i.e. edges for which the incident nodes are assigned the same color, and eliminated costs. The costs incorporate the dynamic behavior of the program by weighting affinity edges with a factor dependent on the program location where the corresponding copy resides. Maximizing the number of satisfied edges corresponds to minimizing the costs under the premise that all costs are equal. We consider the sum of all affinity costs in the IG.[4] Figure 5.4 presents the parameters of the test programs relevant for coalescing. The column "Init Costs" represents the costs of the coloring before coalescing was applied. Accordingly, "Init Unsat" gives the number of unsatisfied affinity edges before coalescing. Note, that the coloring before coalescing is valid, though it is not *good*.

[4]There are graphs for which there is *no* coloring such that all affinity edges are unsatisfied.

Benchmarks	∅ $\|V\|$	max $\|V\|$	∅ Aff. Nodes	∅ Comp. Size	∅Δ_{Aff}
`164.gzip`	164.38	930	109.08	8.67	1.92
`175.vpr`	169.41	2293	85.24	9.30	1.82
`176.gcc`	201.56	6646	112.28	9.84	1.95
`181.mcf`	116.27	388	65.04	10.80	1.68
`186.crafty`	721.12	2752	534.41	10.96	1.81
`197.parser`	123.83	1684	68.48	8.53	1.92
`253.perlbmk`	198.88	3658	122.55	9.53	1.99
`254.gap`	220.01	3574	151.50	9.75	2.03
`255.vortex`	201.01	4380	106.94	10.47	2.07
`256.bzip2`	148.45	1016	92.12	8.43	1.87
`300.twolf`	318.07	3152	197.11	12.04	1.65

Figure 5.3: Interference Graph Statistics

	Max Costs	Init Costs	Max Unsat	Init Unsat
`164.gzip`	3456885	2225991	9010	6574
`175.vpr`	17105748	6949543	18716	13057
`176.gcc`	221483452	145321851	124379	88535
`181.mcf`	136390	71807	1422	928
`186.crafty`	27781660	19681278	8233	5664
`197.parser`	22227033	17270365	21272	15828
`253.perlbmk`	49321969	34860898	89714	66871
`254.gap`	131547137	89237335	113067	84846
`255.vortex`	28572683	21770254	98745	75164
`256.bzip2`	7007580	4882642	6368	4540
`300.twolf`	162497955	63259503	30904	18151

Figure 5.4: Costs and Number of Affinity Edges

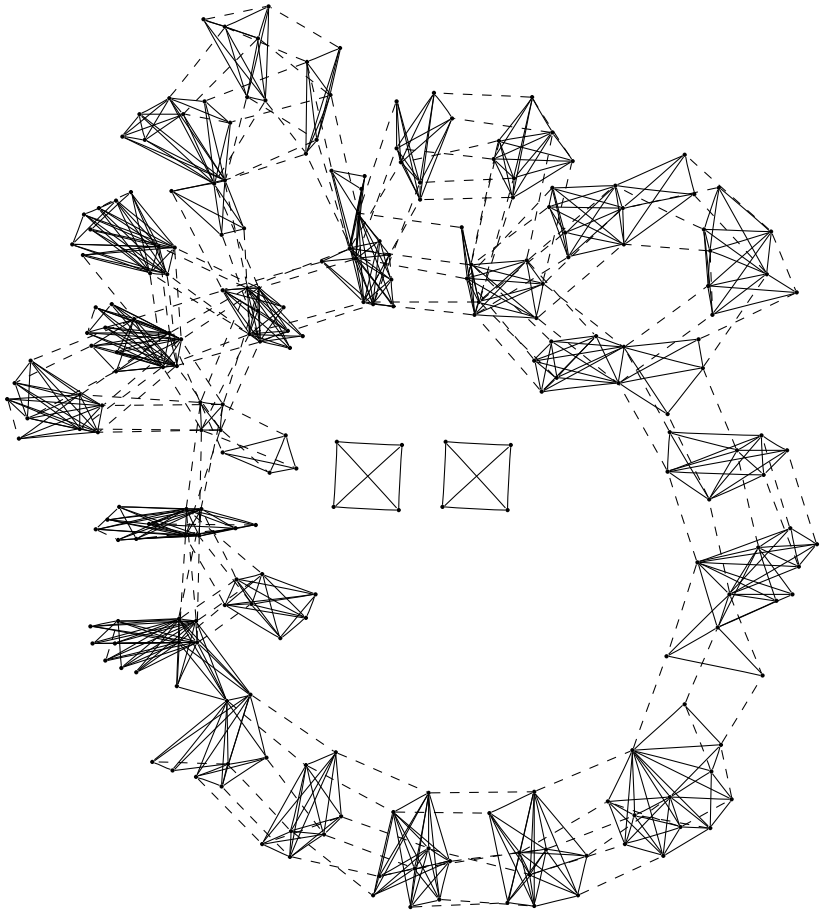


Figure 5.5: A typical interference graph with affinities as it is produced by our compiler for the x86 architecture. Note the interference components connected by affinity edges which are caused by intensive live range splitting.

5.2.1.3 The Coalescing Heuristic

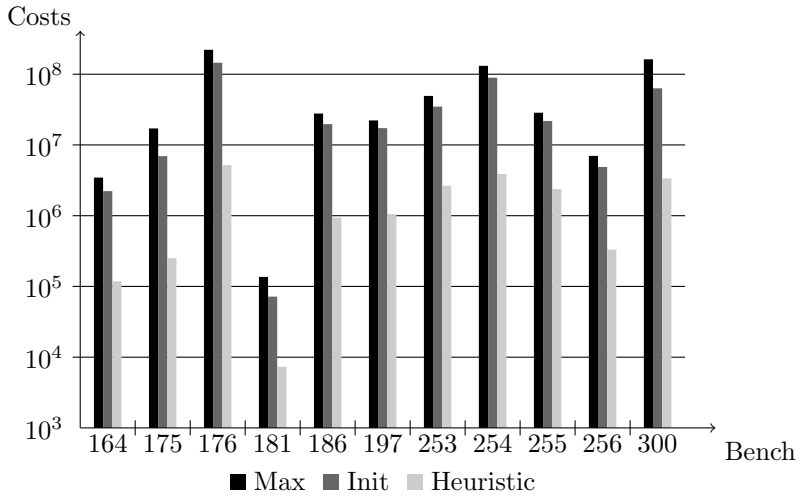
To improve their coloring with respect to affinity costs, all IGs were processed by coalescing heuristic presented in Section 4.5.2. The results are displayed in Figure 5.6. The diagram shows three columns per benchmark. The black one shows the maximal costs of all IGs in the respective benchmark. The dark grey column "Init" shows the costs before coloring and the light gray one "Heuristic" shows the costs after the coalescing heuristic ran. The table in Figure 5.6 also lists the number of unsatisfied affinity edges after coalescing ("After Unsat") and the percentage with respect to the total number of affinity edges ("%After Unsat"). The column "%After Costs" gives the percentage of affinity costs present in the IG after coalescing. We can see that the percentage of costs that are eliminated by the heuristic ranges from 91% to 98%. Hence, the presented heuristic is a reasonable means to eliminate most of the copies inserted due to live range splitting. Figure 5.7 shows the runtime distribution of the coalescing heuristic. For 80% percent of all interference graphs the heuristic takes less than 500ms to run. It should be noted that the heuristic is written in Java and the compiler is implemented in C. This runtime includes the transfer of the interference information from C to Java. A pure C implementation would surely be faster.

5.2.1.4 ILP Coalescing

To better assess the quality of the coalescing heuristic we applied an ILP solver to all benchmark programs.[5] We used the ILP formulation presented in Section 4.5.3 and set the time limit for the solver to five minutes. Prior to the ILP, we ran the heuristic on the graph and provided its solution as a start value to the ILP solver. Hence, the solver did not need to spend time on searching a feasible solution in the beginning. Out of 4459 problems, only 211 were not solved *optimally* within this time. In those cases we also recorded the best lower bound (abbreviated BB in the following) known to the solver after five minutes. The optimal solution of a problem lies between BB and the objective value returned. However, a common observation in practice is that the solver starts lowering the objective value[6] and then remains a long time at the same best solution, only raising the lower bound. Therefore, there might be problems among the 211 non-optimal ones, for which the ILP solution *is* optimal but the solver failed to prove it within the time limit.

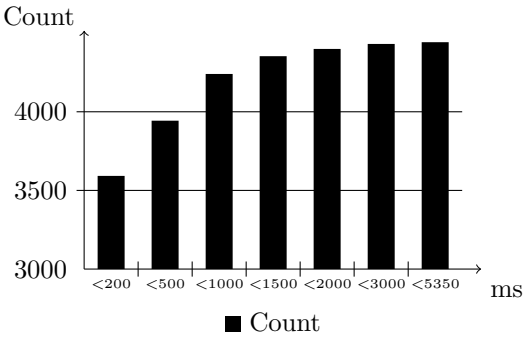
[5] The solver was CPLEX 9.1 (see http://www.ilog.com) on a 2.4Ghz Pentium with 1GB main memory.

[6] Note that we expressed coalescing as a minimization problem



	After Costs	%After Costs	After Unsat	%After Unsat
`164.gzip`	118165	3.42	569	6.32
`175.vpr`	250873	1.47	1209	6.46
`176.gcc`	5180274	2.34	7635	6.14
`181.mcf`	7335	5.38	93	6.54
`186.crafty`	938870	3.38	477	5.79
`197.parser`	1052474	4.74	1763	8.29
`253.perlbmk`	2651188	5.38	6948	7.74
`254.gap`	3875514	2.95	7637	6.75
`255.vortex`	2379601	8.33	7881	7.98
`256.bzip2`	331873	4.74	386	6.06
`300.twolf`	3360710	2.07	1548	5.01

Figure 5.6: Quality of the Coalescing Heuristic



Time (ms)	Count
< 200	3592
< 500	3943
< 1000	4240
< 1500	4353
< 2000	4399
< 3000	4431
< 5350	4442

Figure 5.7: Speed of the Coalescing Heuristic

Figure 5.8 shows a diagram comparing the heuristic's results to the ones of the ILP solver expressed in costs after coalescing per benchmark. The bars entitled "ILP 5min" represent the objective values as they were returned by the solver. This is a *optimistic* estimation on the optimality since it considers the best known solution as the optimal one in the case the solver was not able to prove the optimality in time.

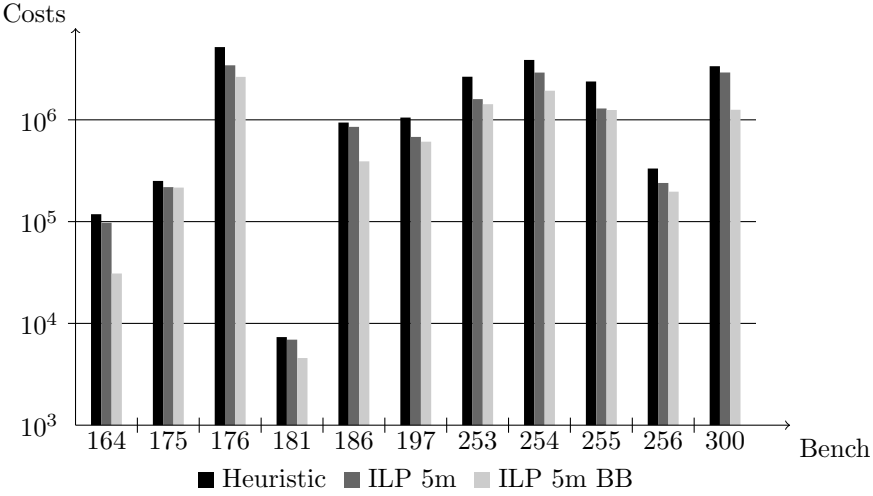
The bar "ILP 5m BB" shows the costs if the best solutions are substituted with the best lower bounds for the non-optimal problems. This is a *pessimistic* estimation on the optimality as the optimal cannot be better than the bound. The true optimum lies somewhere between "ILP 5min" and "ILP 5m BB".

The columns "Quot" (and "Quot BB") relate the results of the heuristic to the ILP. It shows the percentage of the costs left by the heuristics in relation to the costs left after the ILP solution (and the best lower bound, both after five minutes solving time).

Concluding, the heuristic's results are 80% worse than the ones of the ILP in the worst case and 10% worse in the best case. Considering the huge reduction in costs the heuristic yielded, these results can be seen as close to the optimum. This proposition is supported by the following runtime measurements.

5.2.2 Runtime Experiments

To investigate the effects of coalescing several measurements concerning the speed of the compiled programs were carried out. We present the results for

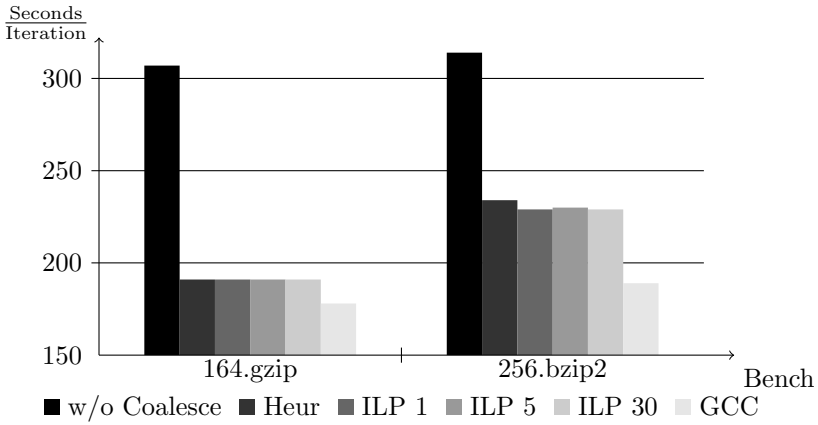


	Heuristic	ILP 5m	ILP 5m BB	Quot	Quot BB
`164.gzip`	118165	97356	30935	121.37	381.97
`175.vpr`	250873	218105	215758	115.02	116.28
`176.gcc`	5180274	3429671	2641368	151.04	196.12
`181.mcf`	7335	6925	4567	105.92	160.61
`186.crafty`	938870	852833	390419	110.09	240.48
`197.parser`	1052474	678415	609249	155.14	172.75
`253.perlbmk`	2651188	1596567	1424011	166.06	186.18
`254.gap`	3875514	2908392	1930799	133.25	200.72
`255.vortex`	2379601	1292513	1248252	184.11	190.63
`256.bzip2`	331873	239528	196840	138.55	168.60
`300.twolf`	3360710	2915713	1253567	115.26	268.09

Figure 5.8: Quality of Coalescing Heuristic

the benchmarks `164.gzip` and `256.bzip2`. Both programs were compiled in five versions:

1. Without any coalescing.
2. With the heuristic presented in Section 4.5.2.
3. With the ILP-based method as described in Section 4.5.3. Again, the solver was equipped with the coloring of the heuristic as a start solution. The ILP measurements were done with three different time limits: 1, 5 and 30 minutes.



Benchmark	w/o Coalesce	Heur	ILP 1	ILP 5	ILP 30	GCC
`164.gzip`	307	191	191	191	191	178
`256.bzip2`	314	234	229	230	229	189

Figure 5.9: Runtime Experiments

The machine on which the runtime tests ran was an AMD Athlon XP 3000+ with 1 GB of main memory and SuSE Linux 9.3 installed. We used the standard SPEC tools to measure the speed of the program. As a reference we also ran the tests with GCC-compiled versions of the test programs. We used GCC version 3.3.5 with the switches `-O3 -march=athlon-xp`. Figure 5.9 shows the runtime in seconds for the two selected benchmarks.

One can see that with coalescing, the runtime decreases massively. The ILP solutions however (independent of the solver time limit), do not improve

the runtime significantly. The good values for GCC are mainly due to other deficiencies in our compiler, such as bad scheduling leading to many spills or lack of other optimizations such as loop unrolling and aggressive exploitation of machine features.

6 Conclusions and Future Work

6.1 Conclusions

In this thesis we present a novel approach of performing register allocation on the SSA form. Our main contribution is the proof that interference graphs of SSA-form programs are chordal. This theoretical result has far reaching consequences for the architecture of global register allocators which we comprehensively cover in this thesis:

- Pressure based spilling heuristics as they are commonly used are now sufficient to make the interference graph colorable. No further spills will be introduced by the coloring phase. This allows to consider coloring and spilling separately, which is an enormous simplification to register allocation since the expensive iterative approach used in most global graph-coloring register allocators can be abandoned.

- We provide a thorough analysis of the register targeting problem as it occurs in all contemporary runtime systems and processor architectures. We show, that for general register constraints, register allocation is NP-complete even for a single constrained instruction in one basic block. Based on this result, we consider a restricted version of the targeting problem and show how it can be seamlessly integrated in our register allocator framework.

- We show that coloring the interference graph can actually be done without constructing the graph itself. This is very important for scenarios where compile speed matters, such as Just-In-Time compilers, as interference graphs are usually complex data structures that are tedious to administer.

- As our maxim is to never make spills in favor of copies, the elimination

of copies (coalescing) may never be allowed to turn a valid coloring into an invalid one. Thus, we express coalescing as an optimization problem over interference graphs: find not only a valid coloring but a *good* one with respect to copy costs. Thereby, we take care that the interference graph itself is never modified to preserve its chordality; the only thing that changes is its coloring. Furthermore, we describe efficient ways how coalescing algorithms can query the interference graph without materializing it as a data structure.

After laying the theoretical foundations, we evaluate our approach using a modern C compiler targeted at the x86 architecture. Hereby, our focus is directed on the quality of the proposed coalescing methods. As constrained instructions appear frequently in the code for x86 processors, a lot of live range splitting is performed by our allocator. As our runtime experiments show, this makes coalescing crucial as it reduces the runtime of the test programs by 40%.

Furthermore, as the evaluation shows, the proposed coalescing heuristic delivered solutions which are very close (between 10% and 80% worse) to the ones found by an ILP solver. These ILP solutions were proven to be optimal in 95% of all compiled functions. Concerning the runtime of the compiled programs, the better solutions of the ILP solver provide only marginal speedup. Hence, the proposed heuristic provides very good results.

6.2 Future Work

Future research in this area could include the following topics:

- We experimented with machines with few registers. Evaluating this allocator for machines with more registers (especially newer DSPs with 64 or 128 registers) is desirable. Especially the effects of having a large register file on coalescing have to be investigated.
- Our copy minimization problem only takes into account the given locations for the live range splits. It is not capable of moving copies to other places in the program. However, allowing live range splits at arbitrary locations could result in lower total costs.
- The integration of more complicated register targeting issues like paired registers and aliased registers is important especially for DSPs.
- The potential compile time benefits of SSA-based register allocation could be investigated in Just-In-Time compilation scenarios.

A Graphs

Let $[A]^k$ be the set of all subsets of A containing k elements. A graph $G = (V, E)$ is a pair of a set of vertices V and a set of edges $E \subseteq [V]^2$. If there is $\{v, w\} \in E$ we will write $vw \in E$. A graph $G' = (V', E')$ is a *subgraph* of G if $V' \subseteq V'$ and $E' \subseteq [V']^2$. A subgraph $G' \subseteq G$ is an *induced subgraph* if whenever there is $vw \in E$ for $v, w \in V'$ there is $vw \in E'$. If $G' = (V', E')$ is an induced subgraph of G, we will also write $G[V']$ since it is exactly the graph containing all edges of G which end in vertices of V'.

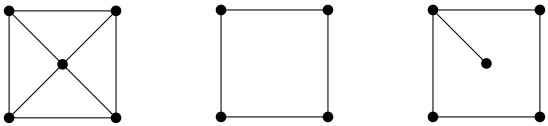


Figure A.1: G' and G'' are both subgraphs of G. G' is also an induced subgraph of G

A graph $G = (V, E)$ with $E = [V]^2$ is called a *complete graph*. A complete graph with n vertices is abbreviated K^n. If H is a induced complete subgraph of G, then H is called a *clique*. The size of the largest clique in a graph G is called its *clique number*. It is abbreviated by $\omega(G)$. In Figure A.1, $\omega(G) = 3$ and $\omega(G') = \omega(G'') = 2$.

A *path* $P = (V, E)$ is a graph with

$$V = \{v_1, \ldots, v_k\} \text{ and } E = \{v_1v_2, \ldots, v_{k-1}v_k\}$$

If $P = (\{v_1, \ldots, v_k\}, E)$ is a path, we shortly write $P = v_1v_2 \ldots v_k$. Let $P = (V, E)$ be a path and $|V| \geq 3$, then $C = (V, E \cup v_kv_1)$ is a *cycle*. Note that $K^3 \cong C^3$. For example, the graph G' in Figure A.1 is isomorphic to C^4.

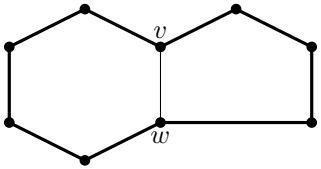


Figure A.2: A graph with an C^9 subgraph, a chord vw and induced subgraphs C^6 and C^5

An edge which connects two vertices of a cycle in some graph G but is not member of the cycle is called a *chord*. Thus, a cycle in a graph is chordless if it is induced. Figure A.2 shows a C^9, a chord and two *induced* cycles.

A.1 Bipartite Graphs and Matchings

Let $G = (V, E)$ by a graph. A set of edges $M \subseteq E$ is a *matching* of G if for each $e_1, e_2 \in M$ there is $e_1 \cap e_2 = \emptyset$. A matching M is *perfect* if the edges in M cover all nodes, i.e. $V = \bigcup_{e \in M} e$.

A graph $G = (V, E)$ is called *bipartite* if V can be subdivided into two disjoint subsets A and B such that for each edge $e = xy$ there is either $x \in A$, $y \in B$ or $x \in B$, $y \in A$. We then also write $G = (A, B, E)$.

A.2 Perfect Graphs

Let $G = (V, E)$ be a graph. A (vertex) coloring $c(G) : V \to R$ maps the vertices of G to some set R with the condition that $c(v) \neq c(w)$ for each edge vw in E. The smallest $k = |R|$ for which there exists a coloring of G is called the *chromatic number* of G and is denoted by $\chi(G)$. A coloring $c(G) : V \to R$ with $|R| = \chi(G)$ is called a *coloring* of G. Trivially for each graph G holds $\omega(G) \leq \chi(G)$. For all graphs in Figure A.1 there is $\omega(G) = \chi(G)$. However $\chi(C^5) = 3$ and $\omega(C^5) = 2$.

Finding a minimal coloring, determining the chromatic number and checking if a graph is k-colorable are all NP-complete, cf. [Garey and Johnson, 1979, Problem GT4] for example.

A graph G is called *perfect* if $\chi(H) = \omega(H)$ for each induced subgraph H of G. Trivially, if a graph contains an induced odd cycle ≥ 5 (C^{2k+3}) it is not perfect. Thus, the graph in Figure A.2 is not perfect.

A.2.1 Chordal Graphs

A graph is called *chordal* (also *triangulated* or *rigid-circuit*) if it does not contain any induced cycle longer than 3. All graphs in Figure A.1 are *not* chordal since they have induced cycles longer than 3. Chordal graphs are perfect, see e.g. the textbook by Golumbic [1980]. There exist several characterizations of chordal graphs which are helpful for understanding their properties, so we discuss some of them here, briefly.

A.2.1.1 Gluing on Cliques

By the definition above, each K^n is chordal since a complete graph does not possess any induced cycle longer than three. Let G be graph with subgraphs G_1, G_2 and S for which there is $G = G_1 \cup G_2$ and $S = G_1 \cap G_2$. We then say, G is obtained by *gluing* G_1 and G_2 along S. One can show that iff G_1 and G_2 are chordal and S is a complete graph, then G is also chordal (cf. Diestel [2005] for example). Thus, chordal graphs can be constructed by recursively gluing chordal graphs on cliques starting with complete graphs. See Figure A.3 for an example.

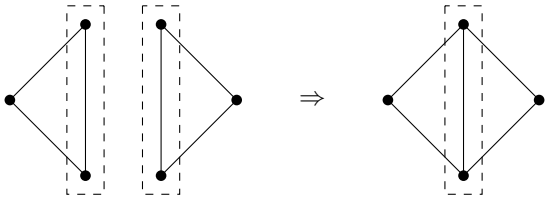
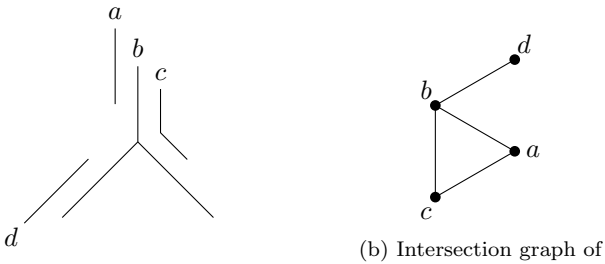


Figure A.3: Gluing on Cliques

A.2.1.2 Intersection Graphs of Subtrees

Gavril [1974] showed that the class of chordal graphs are exactly the class of intersection graphs of subtrees. A subtree can be thought of as a tree-shaped interval having multiple ends. Figure A.4a shows a set of subtrees and Figure A.4b their intersection graph.



(a) Set of subtrees T

(b) Intersection graph of subtrees T

Figure A.4: Subtrees and their Intersection Graphs

A.2.1.3 Perfect Elimination Orders

One of the simplest method to check if a graph is chordal is to construct a *perfect elimination order (PEO)*. A PEO is obtained by successively removing *simplicial* vertices from the graph. Let $G = (V, E)$ be a graph. A vertex v is *simplicial* if v and all its neighbors induce a clique in G. For example, the vertices a, c and d are simplicial in the graph shown in Figure A.4b. By a theorem of Dirac [1961], each chordal graph with more than one vertex possesses at least two simplicial vertices. Furthermore, by removing a vertex from a chordal graph, the graph remains chordal. Thus, chordal graphs can successively be eliminated by removing simplicial vertices. Note, that a cycle longer than three does not have any simplicial vertex.

Remark A.1: PEOs provide a simple way to color chordal graphs optimally. Let $G = (V, E)$ be chordal and $\omega(G) = k$. Re-insert the vertices of the PEO in reverse order and assign each vertex a color that has not been used by its already inserted neighbors. When a vertex v is inserted, it is simplicial, thus all its already inserted neighbors form a clique. As the largest clique in the graph is k, v has as at most $k - 1$ neighbors and there is a color left to color v.

A.2.1.4 R-Orientations

Another way of characterising chordal graphs are so called R-orientations. An *orientation* of an undirected graph $G = (V_G, E_G)$ is a directed graph $D = (V_G, E_D)$ with $E_D \subseteq V_G \times V_G$ for which holds if $ab \in E_G$ there is either

$(a, b) \in E_D$ or $(b, a) \in E_D$. An R-orientation D of an undirected graph G is an orientation which further fulfills following properties:

1. D contains no directed cycles
2. If $(b, a) \in E_D$ and $(c, a) \in E_D$ then there is $bc \in E_G$, i.e. $(b, c) \in E_D$ or $(c, b) \in R_D$.

Rose [1970] shows that the graphs possessing an R-orientation are chordal and vice versa.

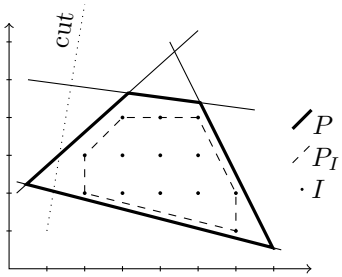
B Integer Linear Programming

Integer linear programming (ILP) is the problem of maximizing or minimizing a linear objective function subject to linear (in)equality constraints and integrality restrictions on a set of variables. In general, solving an ILP is NP hard, but in practice even large instances (depending on the formulation of the problem) can be solved in a reasonable amount of time. Let $P = \{x \mid Ax \geq b, x \in \mathbb{R}^n_+\}$, $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, $A \in \mathbb{R}^{m \times n}$. Then ILP is the problem:

$$\min f = c^T x, \quad x \in P \cap \mathbb{Z}^n$$

The set P is called the *feasible region* as it contains all *feasible solutions* of the problem. P is called *integral* if it is equal to the *convex hull* $P_I = conv(P \cap \mathbb{Z}^n)$ of the integer points I. Droping the integrality constraints leads to the *relaxed* version of the problem:

$$\min f = c^T x, \quad x \in P$$



The figure to the right gives an example for the relation between the feasible region and the convex hull in two-dimensional space: the thick lines denote the feasible region P while the dashed lines denote the convex hull of $P \cap \mathbb{Z}^n$. All real points in P are feasible concerning the relaxed problem while the feasible points for the ILP are given by all (integral) points in P_I.

In constrast to solving an ILP, solving the relaxed problem is possible in polynomial time. Hence, if one can formulate the problem in a way that $P = P_I$ then solving the relaxed problem leads to the solutions of the ILP. Thus, one is generally interested in finding additional inequalties for the ILP formulation which do not constrain the problem further but close the gap

between the feasible region and the convex hull. Such inequalities are called *cuts*. In our example, the dotted line represents a cut. Cuts are generally a mean to provide the solver with more information of the problem's structure so that it can discard infeasible solutions more quickly. For an in-depth coverage on (I)LP we refer to Schrijver [1986], Nemhauser and Wolsey [1988].

Bibliography

C. Andersson. Register Allocation by Optimal Graph Coloring. In G. Hedin, editor, *CC 2003*, volume 2622 of *LNCS*, pages 33–45, Heidelberg, 2003. Springer-Verlag.

A. Appel and L. George. Optimal Coalescing Challenge. Website, August 2000. URL http://www.cs.princeton.edu/~appel/coalesce/.

A. W. Appel and L. George. Optimal Spilling for CISC Machines with Few Registers. In *ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 243–253, June 2001.

A. W. Appel and J. Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, second edition, 2002.

L. Belady. A Study of Replacement of Algorithms for a Virtual Storage Computer. *IBM Systems Journal*, 5:78–101, 1966.

M. Biró, M. Hujter, and Z. Tuza. Precoloring extension. I. Interval Graphs. *Discrete Mathematics*, 100:267–279, 1992.

F. Bouchez, A. Darte, C. Guillon, and F. Rastello. Register Allocation and Spill Complexity under SSA. Technical Report 2005-33, ENS-Lyon, Aug 2005.

P. Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, April 1992.

P. Briggs, K. D. Cooper, and L. Torczon. Improvements to Graph Coloring Register Allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, 1994. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/177492.177575.

P. Brisk, F. Dabiri, J. Macbeth, and M. Sarrafzadeh. Polynomial Time Graph Coloring Register Allocation. In *14th International Workshop on Logic and Synthesis*. ACM Press, 2005.

Z. Budimlić, K. D. Cooper, T. J. Harvey, K. Kennedy, T. S. Oberg, and S. W. Reeves. Fast Copy Coalescing and Live-Range Identification. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 25–32. ACM Press, 2002. ISBN 1-58113-463-0. doi: http://doi.acm.org/10.1145/512529.512534.

D. Callahan and B. Koblenz. Register Allocation via Hierarchical Graph Coloring. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 192–203, New York, NY, USA, 1991. ACM Press. ISBN 0-89791-428-7. doi: http://doi.acm.org/10.1145/113445.113462.

G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register Allocation via Graph Coloring. *Journal of Computer Languages*, 6:45–57, 1981.

F. Chow and J. Hennessy. Register Allocation by Priority-based Coloring. In *SIGPLAN '84: Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, pages 222–232, New York, NY, USA, 1984. ACM Press. ISBN 0-89791-139-3. doi: http://doi.acm.org/10.1145/502874.502896.

F. Chow and J. L. Hennessy. The Priority-based Coloring Approach to Register Allocation. *ACM Trans. Program. Lang. Syst.*, 12(4):501–536, 1990. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/88616.88621.

C. Click and M. Paleczny. A simple Graph-based Intermediate Representation. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, pages 35–49, New York, NY, USA, 1995. ACM Press. ISBN 0-89791-754-5. doi: http://doi.acm.org/10.1145/202529.202534.

K. D. Cooper, T. J. Harvey, and L. Torczon. How to build an Interference Graph. *Software – Practice and Experience*, 28(4):425–444, 1998.

R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadek. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

R. Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer, third edition, July 2005.

G. Dirac. On rigid Circuit Graphs. *Abh. Math. Sem. Univ. Hamburg*, 25: 71–76, 1961.

K. Elbassioni, I. Katriel, M. Kutz, and M. Mahajan. Simultaneous Matchings. In *Proceedings of the ISAAC 2005*, volume 3524 of *LNCS*, pages 168–182. Springer, 2005.

A. P. Ershov. ALPHA – An automatic Programming System of high Efficiency. *ALGOL Bull.*, (19):19–27, 1965. ISSN 0084-6198. doi: http://doi.acm.org/10.1145/1060998.1061006.

J. Fabri. Automatic Storage Optimization. In *SIGPLAN '79: Proceedings of the 1979 SIGPLAN symposium on Compiler construction*, pages 83–91, New York, NY, USA, 1979. ACM Press. ISBN 0-89791-002-8. doi: http://doi.acm.org/10.1145/800229.806957.

M. Farach and V. Liberatore. On local register allocation. In *SODA '98: Proceedings of the ninth annual ACM-SIAM Symposium on Discrete Algorithms*, pages 564–573, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics. ISBN 0-89871-410-9.

C. Fu and K. Wilken. A faster optimal Register Allocator. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international Symposium on Microarchitecture*, pages 245–256, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. ISBN 0-7695-1859-1.

M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, January 1979. ISBN: 0716710455.

F. Gavril. Algorithms for Minimum Coloring, Maximum Clique, Minimum Covering by Cliques, and Independent Set of a Chordal Graph. *SIAM Journal on Computing*, 1(2):180–187, June 1972.

F. Gavril. The Intersection Graphs of Subtrees in Trees are exactly the Chordal Graphs. *J. Combin. Theory Ser. B*, (16):47–56, 1974.

L. George and A. W. Appel. Iterated Register Coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, 1996. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/229542.229546.

M. C. Golumbic. *Algorithmic Graph Theory And Perfect Graphs*. Academic Press, 1980.

D. W. Goodwin and K. D. Wilken. Optimal and near-optimal Global Register Allocation using 0-1 Integer Programming. *Software – Practice & Experience*, 26(8):929–965, August 1996.

D. Grund. Kopienminimierung in einem SSA-basierten Registerzuteiler. Master's thesis, Universität Karlsruhe, August 2005. URL http://www.info.uni-karlsruhe.de/papers/da_grund.pdf.

D. Grund and S. Hack. A fast Cutting-Plane Algorithm for Optimal Coalescing. accepted for Compiler Construction 2007, March 2007.

S. Hack. Interference Graphs of Programs in SSA-form. Technical Report 2005-15, Universität Karlsruhe, June 2005. URL http://www.info.uni-karlsruhe.de/~hack/ifg_ssa.pdf.

S. Hack and G. Goos. Optimal Register Allocation for SSA-form Programs in Polynomial Time. *Information Processing Letters*, 98(4):150–155, May 2006. URL http://dx.doi.org/10.1016/j.ipl.2006.01.008.

S. Hack, D. Grund, and G. Goos. Towards Register Allocation for Programs in SSA-form. Technical Report 2005-27, September 2005. URL http://www.info.uni-karlsruhe.de/~hack/ra_ssa.pdf.

S. Hack, D. Grund, and G. Goos. Register Allocation for Programs in SSA-form. In A. M. Andreas Zeller, editor, *Compiler Construction 2006*, volume 3923. Springer, March 2006. URL http://dx.doi.org/10.1007/11688839_20.

J. L. Hennessy and D. A. Patterson. *Computer Architecture. A quantitative Approach*. Morgan Kaufmann, second edition, 1997.

S. Hoxey, F. Karim, B. Hay, and H. Warren. *The Power PC Compiler Writer's Guide*. Warthman Associates, 1996. ISBN 0-9649654-0-2. URL http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF778525699600755 8C6.

A. B. Kempe. On the Geographical Problem of the Four Colours. *American Journal of Mathematics*, 2(3):193–200, Sep 1879.

T. Lengauer and R. E. Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph. *Transactions on Programming Languages And Systems*, 1(1): 121–141, July 1979.

A. Leung and L. George. Some Notes on the New MLRISC X86 Floating Point Code Generator. Internet, 2001. URL http://www.smlnj.org/compiler-notes/x86-fp.ps.

G. Lindenmaier, M. Beck, B. Boesler, and R. Geiß. Firm, an Intermediate Language for Compiler Research. Technical Report 2005-8, 3 2005. URL http://www.ubka.uni-karlsruhe.de/cgi-bin/pslist?path=ira/2005.

G.-Y. Lueh, T. Gross, and A.-R. Adl-Tabatabai. Fusion-based Register Allocation. *ACM Trans. Program. Lang. Syst.*, 22(3):431–470, 2000. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/353926.353929.

D. Marx. *Graph Coloring with Local and Global Constraints*. PhD thesis, Budapest University of Technology and Economics, 2004.

R. Morgan. *Building an Optimizing Compiler*. Digital Press, Newton, MA, USA, 1998. ISBN 1-55558-179-X.

H. Mössenböck and M. Pfeiffer. Linear Scan Register Allocation in the Context of SSA Form and Register Constraints. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 229–246, London, UK, 2002. Springer-Verlag. ISBN 3-540-43369-4.

S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1998. 1558603204.

G. Nemhauser and L. Wolsey. *Integer and Combinatorial Optimization*. Wiley-Interscience New York, 1988.

F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

C. Norris and L. L. Pollock. Register Allocation over the Program Dependence Graph. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 266–277, New York, NY, USA, 1994. ACM Press. ISBN 0-89791-662-X. doi: http://doi.acm.org/10.1145/178243.178427.

M. Paleczny, C. Vick, and C. Click. The Java HotSpot$^{\text{TM}}$Server Compiler. In *Proceedings of the Java^{TM} Virtual Machine Research and Technology Symposium (JVM '01)*, April 2001.

J. Park and S.-M. Moon. Optimistic Register Coalescing. *ACM Trans. Program. Lang. Syst.*, 26(4):735–765, 2004. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/1011508.1011512.

F. M. Q. Pereira and J. Palsberg. Register Allocation via Coloring of chordal Graphs. In *Proceedings of APLAS'05*, volume 3780 of *LNCS*, pages 315–329. Springer, November 2005.

M. Poletto and V. Sarkar. Linear Scan Register Allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, 1999. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/330249.330250.

F. Rastello, F. de Ferrière, and C. Guillon. Optimizing Translation Out of SSA with Renaming Constraints. Technical Report RR-03-35, LIP, ENS Lyon, June 2003.

F. Rastello, F. de Ferrière, and C. Guillon. Optimizing Translation Out of SSA using Renaming Constraints. In *International Symposium on Code Generation and Optimization*, pages 265–278. IEEE Computer Society Press, March 2004.

D. J. Rose. Triangulated graphs and the elimination process. *Journal of Mathematical Analysis and Applications*, 32(3):597–609, 1970.

B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global Value Numbers and Redundant Computations. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–27, New York, NY, USA, 1988. ACM Press. ISBN 0-89791-252-7. doi: http://doi.acm.org/10.1145/73560.73562.

A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1986.

R. Sethi and J. D. Ullman. The Generation of Optimal Code for Arithmetic Expressions. *J. ACM*, 17(4):715–728, 1970. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321607.321620.

M. D. Smith, N. Ramsey, and G. Holloway. A Generalized Algorithm for Graph-Coloring Register Allocation. In *Proceedings of the ACM SIGPLAN*

2004 Conference on Programming Language Design and Implementation PLDI '04, volume 39, June 2004.

V. C. Sreedhar, R. D.-C. Ju, D. M. Gillies, and V. Santhanam. Translating Out of Static Single Assignment Form. In *SAS '99: Proceedings of the 6th International Symposium on Static Analysis*, pages 194–210, London, UK, 1999. Springer-Verlag. ISBN 3-540-66459-9.

O. Traub, G. Holloway, and M. D. Smith. Quality and speed in linear-scan register allocation. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 142–151, New York, NY, USA, 1998. ACM Press. ISBN 0-89791-987-4. doi: http://doi.acm.org/10.1145/277650.277714.

H. S. Warren. *Hacker's Delight*. Addion Wesley, 2002.

C. Wimmer and H. Mössenböck. Optimized interval splitting in a linear scan register allocator. In *VEE '05: Proceedings of the 1st ACM/USENIX international Conference on Virtual Execution Environments*, pages 132–141, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-047-7. doi: http://doi.acm.org/10.1145/1064979.1064998.

Y. Wu and J. R. Larus. Static Branch Frequency and Program Profile Analysis. In *MICRO 27: Proceedings of the 27th annual international Symposium on Microarchitecture*, pages 1–11, New York, NY, USA, 1994. ACM Press. ISBN 0-89791-707-3. doi: http://doi.acm.org/10.1145/192724.192725.

Index

`undef`, 14, 37
$\mathcal{T}_\Phi$, 12, 38, 39, 53, 55
SSA-Coalescing, 61
Φ-operation, *4*, 11
    Φ^w, 12, 55, 59, 83
    Φ^r, 12, 55, 58, 59
    implementation, *55*
    matrix, 11, 37, 55
    memory operands, 58
    semantics, 12
    swap, 16
Precol-Ext, 81
Sim-P-Match, 77

affinity, *see* edge, affinity
    component, 64
    fulfil, 60, 60
    related, 64
allocation
    spill slot, 58
architecture
    ARM, 21
    x86, 21, 31, 58, 83, 88

basic block, 9
Belady's algorithm, 33

CFG, *see* control flow, graph
chord, 106
chromatic number, 2, 19, 106
chunks, 63
clique, 19, 105
    glueing, 107
clique number, 105
coalesce, 27
coalescing, *3*, 5, 19, 34
    aggressive, 25, 26
    conservative, 26
    ILP, 71
    optimal, 71
    optimistic, 27, 36
    spill slot, 58
    SSA, 59
        heuristic, 63
colorability, 19
coloring, 5
    constrained, 82
    cost function, 60
    minimal, 106
    optimistic, 26
    pessimistic, 26
    SSA, 41, 53
complete graph, 105
control flow, 11
    graph, 9
    predecessor, 8
    successor, 8
convex hull, 111
copy, 57, 88
    insertion due to constraints, 76
    memory, 58

cut, *112*
cuts
  clique-ray, 74
  path, 73
cycle, 40, 105

def-use chains, 68
definition, 9
degree
  significant, 27
dominance, 10, 68
  immediate, 10
  tree order, 10, 39

edge
  affinity, 19, 59
  critical, 10
execution
  frequencies, 59

feasible region, 111
feasible solution, 111
Firm, 87
freeze, 27

graph
  1-perfect, 30
  bipartite, 77, 106
  chordal, 4, 5, 31, 39, 107
  chromatic number, 5
  coloring, *2*
  interference
    uncolorable, 75
  perfect, 5, 30, 106
  rigid-circuit, 107
  triangulated, 107

IG, *see* graph, interference
induced subgraph, 105
integer linear programming, 34, 59, 111
interference, *2*, 17, 18
  graph, *2*, 17, 21
    chordal, 40
    precoloring, 81
    SSA, 40
    SSA vs. non-SSA, 40
    storage, 33
  neighbors, 69
  test, 68

Just-In-Time compilation, 33
just-in-time compilation, 3

label, *see* program, label
linear scan, 34
linearly ordered set, 7
list, 7
live range, 17
  splitting, 18, 19, 26, 28, 34, 36, 82, 88
liveness, *1*, 17, 68
  and Φ-operations, 37
  SSA, 38
lost copies, 15

matching, 106
  C-perfect, 77
  perfect, 106
maximum cardinality search, 30

orientation, 108
  R, 39, 108

parallel copy, 40, 60
path, 9, 105
PEO, *see* perfect elimination order
perfect elimination order, 108
processors, 21
program, 8
  label, 8
  strict, 10, 18, 39

uncolorable, 75

recoloring, 65
register, 1
constraints, 21, 31, 71, 75, 90
demand, 80
homogeneous set, 75
pressure, 3, 17, 19, 32, 42, 75, 88
faithful, 80
targeting, 21, 31, 34, 75
transfer graph, 55, 60
register allocation, *1*, 16
global, 23
graph coloring, 24
hierarchical, 29
on trees, 23
optimal, 35
priority based, 28
region based, 29
valid, 16
relaxed problem, 111
rematerialisation, 21, 34, 50

select, 25, 36, 91
simple constraint property, 78
simplicial, 108
simplify, 25–27, 36, 91
spill slot, 58
spilling, *2*, 3, 5, 19, 34, 88
pre-allocation, 32
SSA, *4*, 11, 19, 26, 29, 31
colorability criterion, 42
destruction, 15, 35, 41, 55
dominance property, 13, 39
IR, 87
single assignment property, 11
static single assignment, *see* SSA
subgraph, 105
swap instruction, 57

two-address-code, 21, 89

usage, 9

variable, 8
memory, 44, 58, 70

xor instruction, 58

Register allocation is one of the most important optimizations in a modern compiler. It aims for minimizing memory accesses by storing as many variables of the program as possible in the processor‘s registers.

This thesis proposes a novel approach to graph-coloring register allocation. We demand that the input programs are in static single assignment form, a program representation property widely used in compiler optimizations nowadays. We show that the interference graphs of SSA-form programs are chordal and discuss the consequences of this result to register allocation. In contrast to non-SSA based approaches, this allows us to use the register pressure to efficiently determine the colorability of a program. This leads to a separation of spilling and coalescing to ensure that after spilling no further memory accesses are introduced. Coloring is no longer an issue since it can be done optimally by two linear passes ver the program. Furthermore, we show how the handling of register targeting constraints can be treated in this setting.

We propose algorithms for all the three subphases that take the peculiarities of the SSA form into account and propose an architecture for a register allocator completely based on SSA. Finally, an experimental evaluation that shows the feasibility of this new approach is presented.

ISBN: 978-3-86644-180-4

www.uvka.de