Towards Register Allocation for Programs in SSA-form

Sebastian Hack, Daniel Grund, Gerhard Goos

`(hack|daniel|ggoos)@ipd.info.uni-karlsruhe.de`
Institut für Programmstrukturen und Datenorganisation
Fakultät für Informatik
Universität Karlsruhe (TH)

22. September 2005

Internal Report 2005-27
ISSN 1432-7864

Abstract

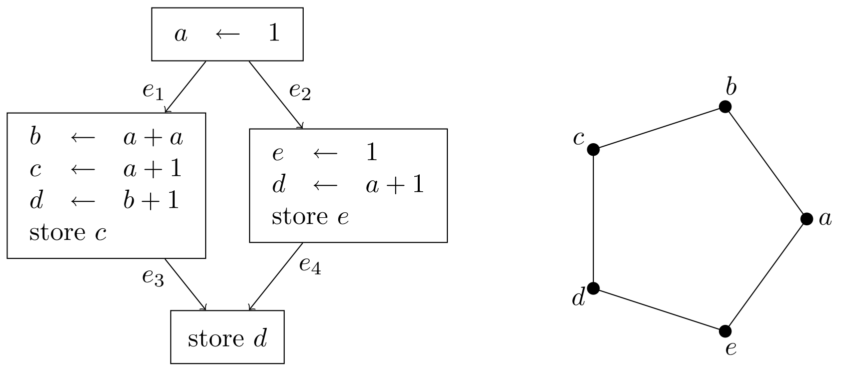
In this technical report, we present an architecture for register allocation on the SSA-form. We show, how the properties of SSA-form programs and their interference graphs can be exploited to develop new methods for spilling, coloring and coalescing. We present heuristic and optimal solution methods for these three subtasks.

1 Introduction

Register allocation is commonly considered as the task of mapping variables in a program to processor registers. Since a variable can be multiply assigned, all definitions of the variable write to the same register. This restriction may sometimes leads to register allocations using more registers than necessary, or even causing memory access. In this paper we investigate register allocation for programs in SSA-form, i.e. all variables are statically assigned only once and correspond to a unique definition of a variable. Thus, multiple definitions of the same (non-SSA) variable are allowed to reside in different registers.

A common technique for register allocation is graph coloring: Each variable in the program corresponds to a node in the undirected *interference graph.* If the compiler finds out that two variables cannot be held in the same register (they *interfere*) an edge is drawn between the two nodes in the interference graph. A k-coloring of the interference graph is thus a valid register allocation for the program.

Consider the program in figure 1(a). Its interference graph (shown in figure 1(b)) needs three colors to be colored. However, putting P in SSA-form makes two distinct variables d_1, d_2 for each definition of d, as shown in figure 2(a). This allows the register allocator to assign two different registers to



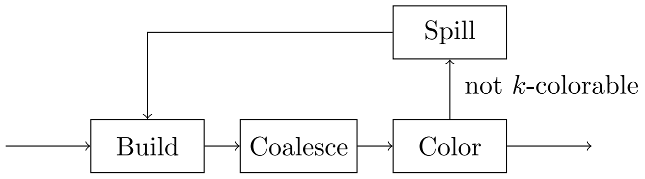
(a) Program P (b) Interference graph of P

Figure 1: A non-SSA program and its interference graph

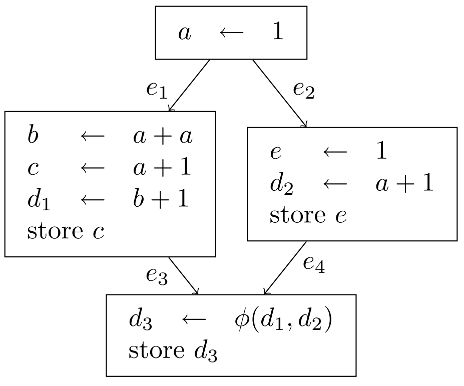
d_1 and d_2. However, this introduces a copy on one of the control flow edges e_3 and e_4. Assuming only two registers are available, one certainly would prefer introducing a copy to avoid memory access. If three registers are available, one would spend the third register, assign d_1, d_2, d_3 the same register and thus eliminate the copy on e_3.

Traditionally, ϕ-operations are removed before register allocation. In doing so, SSA variables which occur as operands of a ϕ-operations are mapped to as few as possible non-SSA variables. Effectively, removing ϕ-operations *before* register allocation results in prematurely coalescing the SSA variables, potentially leading to spills if registers are exhausted. Thus, eliminating a copy can cause a spill, which is a costly trade-off. Preferably, a register allocator should only eliminate a copy if it will not cause a spill.

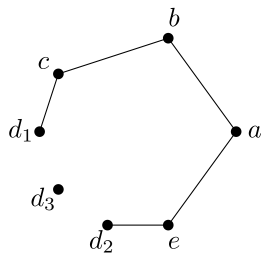
As Chaitin showed in [CAC$^+$81], each undirected graph can occur as an interference graph to some (non-SSA) program. As minimal coloring undirected graphs is $\mathcal{NP}$-complete, usually a heuristic is applied. The variables which have to be spilled are determined while coloring the graph when a node cannot be assigned a color by the heuristic.



The feedback is necessary, since determining the chromatic number (the amount of colors (registers) needed to color the graph) is $\mathcal{NP}$-complete. So, it is not possible to tell how many colors you will need by "simply looking at the graph". Furthermore, the coalescing phase, which tries to eliminate useless copies, may change the colorability of the graph. So making a reasonable trade-off between



(a) SSA version P' of P



(b) Interference graph of P'

Figure 2: SSA version of P and its interference graph

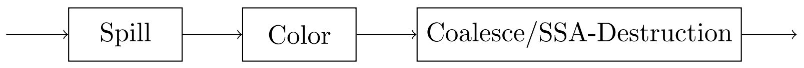
coalescing and spilling is hard for such a register allocator.

However, the situation changes totally if register allocation is based on the SSA-form, as we show in this paper:

- Due to a special property (see [Hac05]) of the interference graphs of SSA-form programs, called *chordality,* the spilling and coalescing phases can be completely decoupled during register allocation (This fact has also and independently been discovered by Pereira and Palsberg [PP05], Brisk [Bri05] and Bouchez, Darte and Rastello [Bou05]).

- 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. The interference graph of program P shown in figure 1(a) is an example for a non-perfect graph since its chromatic number is 3 and the largest clique contains 2 nodes. In other words chordality ensures that local 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. In contrast to usual (non-SSA) graph coloring approaches, no additional register demand can arise from inter-block effects.

- 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.*



As we show in section 3, inserting additional copy instructions will *not* lower the demand of registers in SSA-form programs. Thus, the SSA-form can be seen as a sufficiently "un-coalesced" representation to avoid inserting spills in favor of copies.

- Coloring a chordal graph can be done in $O(|V|^2)$. However, if the dominance relation and live ranges have been computed in advance (which is commonly the case), coloring can be done in $O(\omega(G) \cdot n)$, where n is the number of instructions in the program and $\omega(G)$ the size of the largest live set (which is smaller or equal than the number of available registers after spilling).

- The major source of copy instructions in a program are ϕ-operations. As shown in the introductory example, coalescing these copies too early might result in an unnecessary 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. Since the graph then remains chordal, coalescing can easily keep track of the graph's chromatic number and refuse coalescing two variables if this increases the chromatic number beyond the number of available registers.

  Section 5.2 defines the coalescing problem and proves it $\mathcal{NP}$-complete. In section 5.3, we present a heuristic approach and in section 5.4 we give an optimal solution method for the coalescing problem.

- In section 6 we show how common register targeting constraints can be reduced to the coalescing problem.

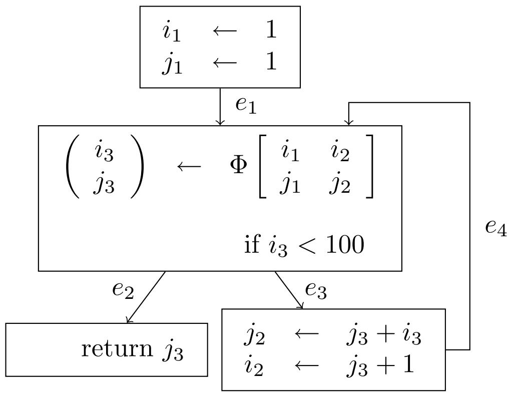
2 Foundations

In this section, we give some foundations which are vital for the rest of this paper. First, we introduce a more compact notation for ϕ-operations, secondly, we discuss several properties of interference graphs of SSA-form programs.

In this report, we assume that all instructions have already been scheduled. Thus, we consider a program as being given by its control flow graph consisting of labelled instructions

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

Each label ℓ has set $pred_\ell$ of control flow predecessors and a set $succ_\ell$ of control flow successors. A basic block is a set of labels $\{\ell_1, \ldots, \ell_n\}$ for which holds: If ℓ_1 is executed, all $\ell_i, 1 < i \leq n$ are executed. Furthermore, we require the program to be in SSA-form (see [CFR$^+$91] for example), i. e. each variable is statically only assigned once.



$i_1 \leftarrow 1$
$j_1 \leftarrow 1$
e_1
$\begin{pmatrix} i_3 \\ j_3 \end{pmatrix} \leftarrow \Phi \begin{bmatrix} i_1 & i_2 \\ j_1 & j_2 \end{bmatrix}$
if $i_3 < 100$
e_4
e_2
e_3
return j_3
$j_2 \leftarrow j_3 + i_3$
$i_2 \leftarrow j_3 + 1$

(a) SSA program Q

$$\begin{array}{lcl}
\text{func } f_1() & = & \text{let } i_1 = 1,\, j_1 = 1 \text{ in } f_2(i_1, j_1) \\
\text{func } f_2(i_3, j_3) & = & \text{if } i_3 < 100 \text{ then } f_3(j_3) \text{ else } f_4(i_3, j_3) \\
\text{func } f_3(j_3) & = & j_3 \\
\text{func } f_4(i_3, j_3) & = & \text{let } j_2 = j_3 + i_3,\, i_2 = j_3 + 1 \text{ in } f_2(i_2, j_2)
\end{array}$$

(b) CPS for Q

Figure 3: SSA and CPS

2.1 ϕ-operations

The way ϕ-operations are commonly written is misleading in two ways:

1. Writing ϕ-operations as functions suggests that two different variables being their operands may never be contained in the same register. For a normal operation such as $+, -, \ldots$ this is true, but not for a ϕ-operation. ϕ "computes" its value based on control flow information in a way that only one operand is used for the computation. Interferences of operands of a ϕ-operation are never caused by the ϕ-operation. So, if the operands of a ϕ-operation do not interfere, it is explicitly wanted to hold them in the same register, since the ϕ then degrades to a no-op. Basically, a ϕ-operation is a notational trick for control flow dependent value renaming.

2. Since a ϕ-operation conventionally only allows to define one value, one usually writes multiple ϕ-operations at the beginning of a basic block to indicate, that multiple values are passed along an incoming edge of this block. This suggests that these ϕ-operations are processed in order, which is wrong as the semantics of SSA requires that all ϕ-operations in a basic block are to be *simultaneously* evaluated as the *first* instruction in this block.

The semantics of ϕ-operations is best described by the *continuation passing scheme* (CPS) (see [Kel95] and [App98]), which transforms a SSA-form program to a functional one by converting each block into a function and converting control flow edges into function calls. The values live along a control flow edge are passed as arguments to the function representing the basic block to which it jumps to, as shown in figure 3. To circumvent some of the notational deficiencies of ϕ-operations, we will subsume all ϕ-operations in a basic block in one matrix-like Φ-operation. The brackets shall indicate, that the x_{ij} are not used as in an ordinary operation.

$$\begin{array}{ccc} y_1 & \leftarrow & \phi(x_{11}, \ldots, x_{1n}) \\ \vdots & & \vdots \\ y_m & \leftarrow & \phi(x_{m1}, \ldots, x_{mn}) \end{array} \quad \Longrightarrow \quad \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix} \leftarrow \Phi \begin{bmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{bmatrix}$$

2.2 Interference graphs of SSA-form programs

We consider a program given by its control flow graph in which each node (called *label*) represents a single instruction (the program is already scheduled). The label where a variable v is defined, is denoted by $\mathcal{D}_v$. If a label ℓ dominates a label ℓ', we write $\ell \preceq \ell'$. Let $G = (V_G, E_G)$ be the interference graph of an SSA-form program P.

The first lemma states that interfering nodes are totally ordered regarding dominance and is due to Budimlić [BCH$^+$02].

Lemma 1. *If two values v and w are live at some label ℓ, either $\mathcal{D}_v$ dominates $\mathcal{D}_w$ or vice versa.*

Lemma 2. *If v and w interfere and $\mathcal{D}_v \preceq \mathcal{D}_w$, then v is live at $\mathcal{D}_w$.*

The next lemma shows, that the dominance relation is not arbitrarily directed for chains in the interference graph.

Lemma 3. *Let $ab, bc \in E_G$ and $ac \notin E_G$. If $\mathcal{D}_a \preceq \mathcal{D}_b$, then $\mathcal{D}_b \preceq \mathcal{D}_c$.*

The proofs of these lemmas can be found in [Hac05].

3 Spilling

Normally, spilling in graph-based register allocators removes nodes from the interference graph to make the graph k-colorable. But the node in the interference graph does not reflect where and how often a variable is used; it just records all interferences throughout the variable's live range by its incident edges. Since removing the node from the graph would cause inserting memory access code after each definition and each usage of the variable, also at places in the program where registers suffice, a lot of work has been done to break the live ranges of variables in reasonable pieces, see [CH90] and [BDEO97] for example.

Bouchez [Bou05] investigates the problem of how to split ranges in order to minimize the number of reload instructions. He gives precise descriptions of the complexity of several variants of this problem and shows, that splitting live ranges to lower the register pressure to a fixed k while inserting a minimum amount of reload instructions is $\mathcal{NP}$-complete depending on the chromatic number of the graph.

The following theorem is the foundation for the spilling techniques we propose in this section. It is trivial but *not* obvious and has been independently proven by Bouchez [Bou05]. For non-SSA programs it does generally not hold. For example, consider the non-SSA version of program Q in figure 3(a), which will be discussed in section 5.1.

Lemma 4. *For each clique $C \subseteq G, V_C = \{v_1, \dots, v_n\}$, there is a permutation $\sigma : V_C \longrightarrow V_C$, such that $\mathcal{D}_{\sigma(v_1)} \preceq \cdots \preceq \mathcal{D}_{\sigma(v_n)}$.*

Proof. By lemma 1, for each v_i, v_j, $1 \leq i < j \leq n$ either $\mathcal{D}_{v_i} \preceq \mathcal{D}_{v_j}$ holds or $\mathcal{D}_{v_j} \preceq \mathcal{D}_{v_i}$ (dominance is total for interfering variables), every sorting algorithm can produce σ. □

Theorem 1. *Let G be the interference graph of a SSA-form program and C an induced subgraph of G. C is a clique in G iff there exists a label in the program where all $V(C)$ are live.*

Proof. "$\Leftarrow$" holds by definition. "$\Rightarrow$": By lemma 4, there exists a permutation σ of the $\{v_1, \dots, v_n\} = V_C$ such that $\mathcal{D}_{\sigma(v_1)} \preceq \cdots \preceq \mathcal{D}_{\sigma(v_n)}$. Then, by lemma 2, all $\sigma(v_1), \dots, \sigma(v_{n-1})$ are live at $\mathcal{D}_{\sigma(v_n)}$. □

This implies, that the amount of registers needed for the program can be determined, by checking the set of variables live at some instruction of the program. Now it is obvious why splitting a live range by a copy instruction does *not* lower the register demand in a SSA-form program: The number of live variables does not change for any instruction in the program, thus the chromatic number of the graph remains the same.

3.1 An adaption of Belady's algorithm

Guo et al. [GGP03] show the power of Belady's algorithm [Bel66] for spilling in long basic blocks in SSA-form. Using the properties of SSA-form programs and their interference graphs, the method by Guo can be extended to work on the whole procedure, not just on a linear sequence of code.

Given k registers and a label ℓ, 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 usages are farthest away from this label. Whereas "away" means the number of instructions which have to be executed from ℓ until the usage is reached. If the usage is in the same basic block, this number is simply given by the number of instructions between

ℓ and the usage. If the usage is outside ℓ's block, we have to define a reasonable measure, as e. g.:

$$nextuse(\ell, v) = \begin{cases} \infty & \text{if } v \text{ is not live at } \ell \\ 0 & \text{if } v \text{ is used at } \ell \\ 1 + \min\limits_{\ell' \in succ_\ell} nextuse(\ell', v) & \text{else} \end{cases}$$

We will apply the Belady method to each basic block seperately 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 Φ-result y in a Φ-operation

$$\begin{pmatrix} \vdots \\ y \\ \vdots \end{pmatrix} \leftarrow \Phi \begin{bmatrix} \vdots & & \vdots \\ x_1 & \cdots & x_n \\ \vdots & & \vdots \end{bmatrix}$$

the $x_1, \ldots, x_n$ are passed along the respective incoming control flow edges to B and are referenced under the "name" y in B.

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

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

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 which 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

$$\ell : (\underbrace{y_1, \ldots, y_m}_{D_\ell}) \leftarrow \tau(\underbrace{x_1, \ldots, x_n}_{U_\ell})$$

the U_ℓ have to be in registers when the instruction is reached. Assume that some of the U_ℓ are not contained in Q, i. e. $R := U_\ell \setminus Q$ is not empty. Thus, reloads have to be inserted for all variables in R. By inserting reloads, the values of the variables are brought into registers. Thus,

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

variables are removed from Q. As the method of Belady suggests, we remove the ones with the highest *nextuse*. 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 τ are written to registers upon executing τ. This means, that

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

variables are displaced from Q when τ writes its results[1]. Note, that for all $v \in U_\ell$, there is $\mathit{nextuse}(v, \ell) = 0$ which implies that the U_ℓ will be displaced lastly. However, not the uses at ℓ are deciding but the uses *after* ℓ. Thus, we need a slightly modified version of *nextuse*

$$\mathit{nextuse}'(\ell, v) = 1 + \min_{\ell' \in \mathit{succ}_\ell} \mathit{nextuse}(\ell', v)$$

which disregards the usages at ℓ. After processing each label in the block, we memorize the last contents of Q in the set out_B.

Finally, after processing each block as described above, we have to combine the results to form a valid solution for the whole procedure. Particularily, 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 $\mathit{in}_B \setminus \mathit{out}_P$ on the edge from P to B[2].

The pseudocode of the algorithm is presented in appendix D.

3.2 Spilling using integer linear programming

Based on theorem 1, we formulate an ILP-based approach for the spilling problem. For means of easier specification, assume that each basic block B contains a terminating instruction which uses all variables live at the end of the basic block[3]. Thus, each variable live at the end of a block B, has at least one use in B. Let ℓ_B denote the label of the terminating instruction in block B.

Let us consider a label

$$\ell : (\underbrace{y_1, \dots, y_m}_{D_\ell}) \leftarrow \tau(\underbrace{x_1, \dots, x_n}_{U_\ell})$$

as shown in figure 4. Let L_ℓ be the set of variables live out at ℓ without the variables defined at ℓ. Since all variables which are operands to τ must be present in registers at ℓ and the variables defined by τ are also written to registers, the following inequalities must hold:

$$\begin{aligned} |L_\ell| + |U_\ell| - |L_\ell \cap U_\ell| &\leq k \\ |L_\ell| + |D_\ell| &\leq k \end{aligned}$$

Thus, the amount p_ℓ of variables which can be "passed by" ℓ in registers is

$$p_\ell := k - \max\{|D_\ell|, |U_\ell| - |L_\ell \cap U_\ell|\}$$

[1]Note, that $Q \cap D_\ell = \emptyset$ since the program is in SSA-form.

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

[3]A variable is live at the end of a block, if it is live out at the block or occurs as an argument to a Φ-operation in a successor of B in the column corresponding to B.

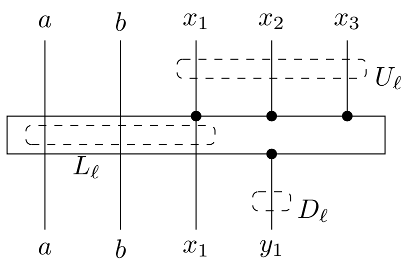


Figure 4: Uses, definitions and variables live through at a label

At each label ℓ where more than k variables are live, the spilling phase has to select a subset smaller or equal to k of variables which shall reside in registers at ℓ. If a variable v is not in this subset, it has to be reloaded at its next use. So, equivalently the spilling phase decides which usage of a variable is preceded by a reload and which not. To avoid memory traffic, the number of reload operations should be kept as small as possible.

3.2.1 Reloads

For every use of a (SSA-)variable x at label ℓ, we introduce a decision variable $m_{(x,\ell)}$ which shall be 1, if x reaches ℓ in memory causing a reload, or 0 if x reaches ℓ in a register. If a variable is selected to remain in memory at some label ℓ, the next use of this variable will imply a reload, thus the decision variable for that use must be 1. We define the function $\mathit{next}(v, \ell)$ to obtain the decision variable for the next use of a variable v live at some label ℓ. Thus, for each label ℓ, we add the following constraint:

$$\sum_{v \in T_\ell} \mathit{next}(v, \ell) \geq p_\ell$$

Consider a variable v live in at some successor S of some block B. If the first use of v in S (note that there is at least one, by construction) is from memory, there is no sense in keeping v in a register at the end of B. Analogously, if the first use of v in S is from a register, v must also be kept in a register at the end of B. If it was not, one would need a reload of v at the beginning of S. Shortly: a variable cannot change its storage location at basic block borders. Thus, for each variable live in at some block B and all predecessors $P_1, \ldots, P_n$ of B where v is live out, we create the following constraints:

$$\begin{aligned} m_{(v,\ell_{P_1})} &= \mathit{next}(v, \text{first label in } B) \\ &\vdots \\ m_{(v,\ell_{P_n})} &= \mathit{next}(v, \text{first label in } B) \end{aligned}$$

Similarly, if v is a result of a Φ-operation

$$\begin{pmatrix} \vdots \\ v \\ \vdots \end{pmatrix} \leftarrow \Phi \begin{bmatrix} \vdots & & \vdots \\ x_1 & \cdots & x_n \\ \vdots & & \vdots \end{bmatrix}$$

we insert constraints

$$\begin{aligned} m_{(x_1,\ell_{P_1})} &= \mathit{next}(v, \text{first label in } B) \\ &\vdots \\ m_{(x_n,\ell_{P_n})} &= \mathit{next}(v, \text{first label in } B) \end{aligned}$$

meaning that if the first use of v (from the beginning of B) is from memory, all the arguments of v's row shall also be in memory at the end of the Φ's predecessor blocks.

The cost function to minimize then consists of a weighted sum of all decision variables $m_{(v,\ell)}$:

$$\sum_{(v,\ell)\in \mathit{Usages}} \mu_{(v,\ell)} \cdot m_{(v,\ell)}$$

3.2.2 Rematerialization

Rematerializing a variable means recomputing it instead of reloading it from memory. The decision variable $r_{(v,\ell)}$ indicates, that v shall be rematerialized at ℓ. Clearly, a variable v can only be rematerialized at a label ℓ, if the operands $o_1, \ldots, o_n$ of the instruction defining v are live at ℓ.

$$\sum_{i=1}^{n} m_{(o_i,\ell)} + n \cdot r_{(v,\ell)} \leq n$$

Additionally, rematerializing v at ℓ only makes sense, iff v reaches ℓ in memory:

$$r_{(v,\ell)} \leq m_{(v,\ell)}$$

If one can rematerialize a variable v at a label ℓ, the costs incurred by $m_{(v,\ell)}$ should be reduced. So it is sensible, to choose the weight $\rho_{(v,\ell)}$ of $r_{(v,\ell)}$ in the target function between 0 and $\mu_{(v,\ell)}$.

3.2.3 Stores

Up to now, we only considered the costs of reloads. If one also wants to take into account the costs of store operations, additional decision variables and constraints have to be added. For each (SSA-)variable v, we introduce one decision variable s_v to indicate whether v must be stored to memory. s_v must be 1, if there is some label ℓ with $m_{(v,\ell)} = 1$. Furthermore, store costs should not be incurred, if a value can be rematerialized. This is expressed by following constraints:

$$\begin{array}{rcl} s_v & \geq & m_{(v,\ell_1)} - r_{(v,\ell_1)} \\ & \vdots & \\ s_v & \geq & m_{(v,\ell_n)} - r_{(v,\ell_n)} \end{array}$$

We then add

$$\sum_{v \in \mathit{Variables}} \sigma_v \cdot s_v$$

to the cost function, where the σ_v are arbitrarily selectable positive weights. Since the cost function gets minimized, s_v will be automatically pulled to 0, if all $m_{(v,\ell_i)}$ are 0.

To sum up, the cost function looks as follows:

$$\sum_{(v,\ell) \in \mathit{Usages}} \mu_{(v,\ell)} \cdot m_{(v,\ell)} - \sum_{(v,\ell) \in \mathit{RematUsages}} \rho_{(v,\ell)} \cdot r_{(v,\ell)} + \sum_{v \in \mathit{Variables}} \sigma_v \cdot s_v$$

Note, that the constraints and variables for stores and rematerialization are optional. If one wants to trade solving time against solution quality, one can omit the variables/constraints for stores and/or rematerialization.

4 Coloring

The main tool for coloring chordal graphs are perfect elimination orders (PEO) (see [Gol80]). PEOs determine in which order the nodes of the graph have to be removed and put on a coloring stack so that the graph can be optimally colored[4] by re-inserting them in reverse order and giving them the smallest color which is not allocated to one of its already inserted neighbors.

PEOs are based on so-called simplicial nodes. A node is simplicial, if all its neighbors belong to the same clique. A lemma by Dirac (see [Gol80]) states that each chordal graph has a simplicial node. Since removing a node (and its incident edges) from a chordal graph preserves chordality, there is always a simplicial node to remove. Now it is obvious, why chordal graphs can be colored so efficiently: When a removed simplicial node is re-inserted, all its previously inserted neighbors form a clique (the node was simplicial as it was removed), so the node is assigned the next free color.

PEOs are closely related to the dominance relation of a SSA-form program. Consider a SSA-form program and its interference graph G:

Theorem 2. *A SSA variable v can be added to a PEO of G if all variables whose definitions are dominated by the definition of v have been added to the PEO.*

[4]using as few colors as possible

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 which are not connected ($va, vb \in E$ and $ab \notin E$). By the proposition, all variables whose definitions are dominated by $\mathcal{D}_v$ have been added to the PEO and removed from G. Thus, $\mathcal{D}_a \preceq \mathcal{D}_v$. Then, by lemma 3, $\mathcal{D}_v \preceq \mathcal{D}_b$ which contradicts the proposition. Thus, v is simplicial. □

As a direct consequence, a PEO can be constructed by a post-order pass over the dominance tree. Equivalently, the interference graph can also be colored by assigning a color to a variable v, when all variables whose definitions dominate the one of v have been colored. Thus, if dominance information and the set of variables live at v's definition are present, the graph can be colored with a simple pass over the dominance tree *without* materializing the interference graph itself. For pseudo code see Algorithm 1.

Algorithm 1 Coloring an interference graph of a SSA-form program

```
procedure COLOR-PROGRAM(Program $P$)
    COLOR-RECURSIVE(start block of $P$)
end
procedure COLOR-RECURSIVE(Basic block $B$)
    $assigned \leftarrow$ colors of the live-in
    for each instruction $(b_1, \dots, b_m) \leftarrow \tau(a_1, \dots, a_n)$ from entry to exit do
        for $a \in \{a_1, \dots, a_n\}$ do
            if last use of $a$ then
                $assigned \leftarrow assigned \setminus color(a)$
            fi
        od
        for $b \in \{b_1, \dots, b_m\}$ do
            $color(b) \leftarrow$ one of $all\_colors \setminus assigned$
        od
    od

    for $\{C \mid B = idom(C)\}$ do
        COLOR-RECURSIVE($C$)
    od
end
```

5 Coalescing

The coalescing phase tries to minimize the number of copy instructions by coalescing variables together. In our setting, copy instructions solely occur due to the destruction of Φ-operations and to handle register constraints imposed by the target architecture. Let us first discuss, how Φ-operations can be implemented using real processor instructions.

5.1 Implementing Φ-operations

Conventionally, while translating out of the SSA-form, Φ-operations are substituted by a sequence of copy instructions. However, the main property of a copy operation is that it *duplicates* a value and therefore brings the value into another register. But duplicating a value is not always necessary when implementing Φ-operations. For example, have a look at the Φ-operation in program Q in figure 3(a). If the basic block of the Φ-operation is reached via edge e_4, j_3 is assigned j_2 and i_3 is assigned i_2. Replacing the Φ by a sequence of copies

$$\begin{aligned} i_3 &= i_2 \\ j_3 &= j_2 \end{aligned}$$

during SSA-destruction suggests that i_3 interferes with j_2 and they thus cannot be assigned the same register. This adds an edge from i_3 to j_2 in the interference graph of Q (see figure 5(a)) raising the register demand from 2 to 3.

However, Φ-operations can be removed in a way that the register demand for that Φ never exceeds the number of variables the Φ defines[5]. Consider an operation

$$(b_1, \dots, b_n) = Perm(a_1, \dots, a_n)$$

which works as a "bulk copy" assigning a_1 to b_1, a_2 to b_2, and so forth in one step. This corresponds to a permutation of the registers assigned to the a_i and b_i. For example, consider the Φ in program Q in figure 3(a) and its register allocated version in figure 5(b). To eliminate the Φ, a *Perm* has to be inserted on the edge e_4 which swaps R_1 and R_2.

There are however situations where a value *must* be duplicated. For instance consider a slight modification of program Q by deleting the variable j_1 and modifying the Φ-operation as shown in figure 5(c). The Φ is now utilized to duplicate i_1 into i_3 and j_3 which can only be achieved with a copy instruction on the edge e_1. The same situation occurs, if the Φ-operation has an argument, which is live-in at the Φ's block (confer to figure 5(c)). Since i_1 then interferes with the Φ, a copy from i_1 to i_3 is inevitable on edge e_1.

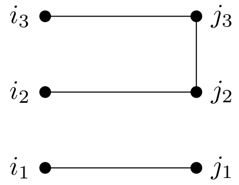
To sum up, duplicates are only needed, if a Φ-argument is used multiply in one column (and thus on the path from the predecessor to the Φ's block) or if a Φ-argument is live-in at the Φ's block. Note that mere interference with a value defined by the Φ is not sufficient: Consider program Q in figure 3(a). j_2 and j_3 interfere but there is no duplicate of j_2 needed, since it is not live-in at the Φ's block.

Considering machine code, permutations can be implemented in various ways (refer to appendix B for basic definitions concerning permutations):

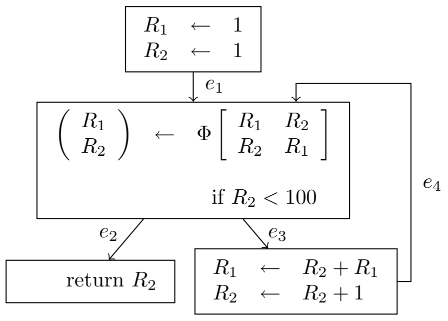
Register swaps

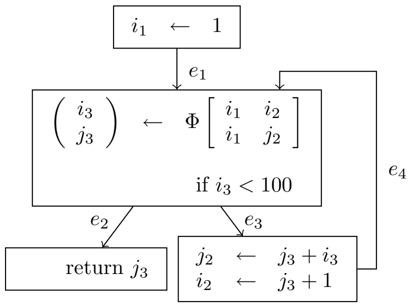
Some architectures, like the x86, have an instruction which swaps the contents of two registers. This directly implements a transposition. Architectures without such an instruction can implement a swap using three

[5]Note that all variables defined by the Φ interfere



(a) Interference Graph of Q


$R_1 \leftarrow 1$
$R_2 \leftarrow 1$
e_1
$\begin{pmatrix} R_1 \\ R_2 \end{pmatrix} \leftarrow \Phi \begin{bmatrix} R_1 & R_2 \\ R_2 & R_1 \end{bmatrix}$
if $R_2 < 100$
e_4
e_2
e_3
return R_2
$R_1 \leftarrow R_2 + R_1$
$R_2 \leftarrow R_2 + 1$
(b) Register allocated Q


$i_1 \leftarrow 1$
e_1
$\begin{pmatrix} i_3 \\ j_3 \end{pmatrix} \leftarrow \Phi \begin{bmatrix} i_1 & i_2 \\ i_1 & j_2 \end{bmatrix}$
if $i_3 < 100$
e_4
e_2
e_3
return j_3
$j_2 \leftarrow j_3 + i_3$
$i_2 \leftarrow j_3 + 1$
(c) SSA program Q'

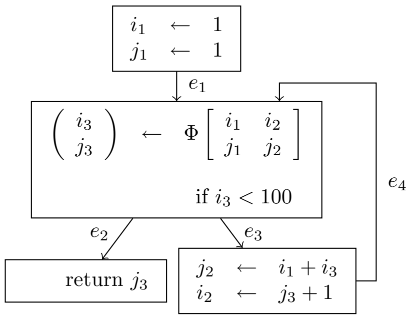

$i_1 \leftarrow 1$
$j_1 \leftarrow 1$
e_1
$\begin{pmatrix} i_3 \\ j_3 \end{pmatrix} \leftarrow \Phi \begin{bmatrix} i_1 & i_2 \\ j_1 & j_2 \end{bmatrix}$
if $i_3 < 100$
e_4
e_2
e_3
return j_3
$j_2 \leftarrow i_1 + i_3$
$i_2 \leftarrow j_3 + 1$
(d) SSA program Q''

Figure 5: Register allocated Q and example programs Q', Q

instructions for the classical xor trick (which also works with addition and subtraction operations).

Moves

Having one backup register, each cycle ζ can be implemented with $|\zeta| + 1$ moves. For example, $\zeta = (2\ 3\ 4)$ can be written as

```
rX = r4
r4 = r3
r3 = r2
r2 = rX
```

where `rX` is a free register not used in the permutation.

5.2 Optimizing Φ-operations

As we have seen, we can eliminate Φ-operations in a way that no additional register demand arises. Thus, a coloring of the interference graph of the SSA-form program is a valid register allocation for the non-SSA one containing *Perm*-operations. In order to lower the number of transpositions needed for a *Perm*, we investigate the problem of maximizing the number of fixed points of a *Perm*[6].

A variable x is a fixed point of a *Perm*

$$(\dots, x', \dots) = \mathit{Perm}(\dots, x, \dots)$$

if x and its target x' are assigned the same register. (see appendix B for a detailed discussion on permutations). Clearly, for fixed points no code has to be generated. Even more, if the *Perm* only consists of fixed points, no code has to be generated for the *Perm* at all.

Given an SSA-form program P, its interference graph $G = (V, E)$ and the set $\mathbf{\Phi}$ of all Φs in P. For a valid k-coloring $f : V \longrightarrow \{1, \dots, k\}$ of G, we define the costs of a Φ-operation

$$p : \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix} = \Phi \begin{bmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{bmatrix}$$

as follows:

$$c_f(p) = \sum_{i=1}^{m} \sum_{j=1}^{n} \mathit{cost}_f(y_i, x_{ij}) \qquad \text{with } \mathit{cost}_f(a, b) = \begin{cases} w_{ab} & \text{if } f(a) \neq f(b) \\ 0 & \text{else} \end{cases} \tag{1}$$

where the $w_{ab} \geq 0$ are costs for copying b to a. The overall costs of the program under the coloring f are then

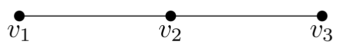
$$c(P, f) = \sum_{p \in \mathbf{\Phi}} c_f(p)$$

[6]Note, that optimizing fixed points is only an approximation corresponding to the traditional coalescing paradigm but does not generally minimize the number of transpositions.

Definition 1 (SSA-Maximize-Fixed-Points)**.** *Given a SSA-form program P and its interference graph G. Find a coloring f of G for which $c(P, f)$ is minimal.*

Theorem 3. SSA-Maximize-Fixed-Points *is $\mathcal{NP}$-complete depending on the number of Φ-operations.*

Proof. We reduce the $\mathcal{NP}$-complete problem of finding a k-coloring of a graph H to SSA-Maximize-Fixed-Points. The method of reduction is based on a brilliant idea by Rastello et al. [RdFG03]. Consider the following example graph H:



We construct a SSA-form program from H in the following way:

- There is one block B
- For each node $v_i \in V_H$ there is a block B_i and a control flow edge $B_i \longrightarrow B$
- For each edge $v_iv_j \in E_H$ there are three basic blocks B'_{ij}, B_{ij}, B_{ji} and the following control flow edges
  - $B_i \longrightarrow B_{ji}$
  - $B_j \longrightarrow B_{ij}$
  - $B_{ij} \longrightarrow B'_{ij}$
  - $B_{ji} \longrightarrow B'_{ij}$
- In each block B'_{ij} a Φ-operation

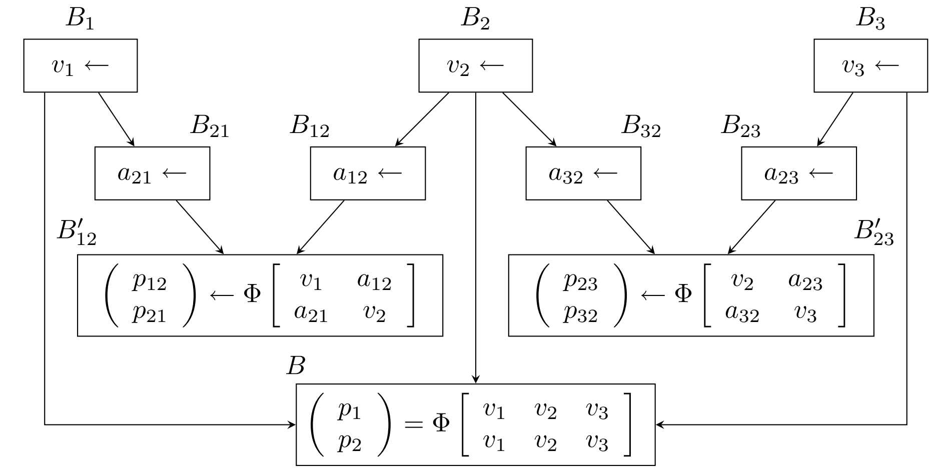
  $$\begin{pmatrix} p_{ij} \\ p_{ji} \end{pmatrix} \leftarrow \Phi \begin{bmatrix} v_i & a_{ij} \\ a_{ji} & v_j \end{bmatrix}$$

  is placed
- A Φ-operation

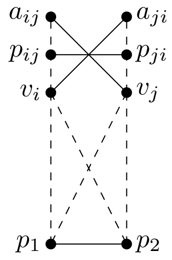
  $$\begin{pmatrix} p_1 \\ \vdots \\ p_k \end{pmatrix} \leftarrow \Phi \begin{bmatrix} v_1 & \cdots & v_{|V_H|} \\ \vdots & & \vdots \\ v_1 & \cdots & v_{|V_H|} \end{bmatrix}$$

  is placed in block B.

Thus, the example program fragment now looks like:


B_1
$v_1 \leftarrow$
B_2
$v_2 \leftarrow$
B_3
$v_3 \leftarrow$
B_{21}
$a_{21} \leftarrow$
B_{12}
$a_{12} \leftarrow$
B_{32}
$a_{32} \leftarrow$
B_{23}
$a_{23} \leftarrow$
B'_{12}
$\begin{pmatrix} p_{12} \\ p_{21} \end{pmatrix} \leftarrow \Phi \begin{bmatrix} v_1 & a_{12} \\ a_{21} & v_2 \end{bmatrix}$
B'_{23}
$\begin{pmatrix} p_{23} \\ p_{32} \end{pmatrix} \leftarrow \Phi \begin{bmatrix} v_2 & a_{23} \\ a_{32} & v_3 \end{bmatrix}$
B
$\begin{pmatrix} p_1 \\ p_2 \end{pmatrix} = \Phi \begin{bmatrix} v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \end{bmatrix}$

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 with different colors than v_i, resulting in costs of $k-1$ for v_i. So, each optimal solution incurs at least costs of $|V_H|(k-1)$.

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-Maximize-Fixed-Points.

A coloring f of G, which does not correspond to a k-coloring of H is no (optimal) solution of SSA-Maximize-Fixed-Points: 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-Maximize-Fixed-Points is $\mathcal{NP}$-complete with the number of Φ-operations. $\square$

5.3 A heuristic approach for Perm optimization

In this section, we present a heuristic approach for finding a k-coloring of the interference graph of a SSA-form program with preferably low costs, as defined in section 5.2.

The basic idea is to alter a coloring (as computed by the method described in section 4) in order to assign arguments of Φ-operations and their results the same color. We emphasize that a valid k-coloring of the interference graph G is always maintained. Generally, the effects of changing the color of a node are not local in the graph and require the alteration of several other nodes' colors. See appendix C for pseudo code of the algorithm.

For each row in a Φ, we build an *optimization unit* $OU = (p, a_1, \dots, a_k)$ consisting of

- The result p of the Φ in this row.
- The arguments $a_1, \dots, a_k$ of the Φ which do not interfere with p. An argument interfering with p can trivially never be assigned p's register.

For each OU a minimization of the costs is then tried separately. The minimization of an OU is not allowed to touch the results of an already processed OU. The processing of an $OU = (p, a_1, \dots, a_k)$ consists of three phases.

Init

For each allowed color $\mathbf{c}$ for p, we insert an entry $E_\mathbf{c} = (\mathbf{c}, C_\mathbf{c}, S_\mathbf{c})$ into a priority queue consisting of

- the color $\mathbf{c}$.
- a conflict graph $C_\mathbf{c}$. Initially, $C_\mathbf{c}$ equals the subgraph induced by $p, a_1, \dots, a_n$ in the interference graph.
- a maximum weighted stable set $S_\mathbf{c}$ of $C_\mathbf{c}$. Each a_i in the OU is assigned the weight w_{pa_i} as defined in the cost function in equation 1 in section 5.2. The weight of p is arbitrary, because p is contained in every maximum stable set by construction. This property is preserved throughout the optimization process.

This queue is ordered by decreasing weights of the $S_\mathbf{c}$. So, if one succeeds in altering the coloring of the interference graph so that all nodes in the first $S_\mathbf{c}$ have the same color, the OU causes minimal costs.

Test

While testing has not finished, the first entry $E_\mathbf{c}$ is removed from the priority queue. We then attempt to adjust the coloring of the interference graph in a way, that the nodes $p, a_1, \dots, a_k$ are assigned the color $\mathbf{c}$. Note, that until the testing phase is not completed for an OU, color changes are only virtual and rolled back if the optimization fails for the OU.

We try to change the color for each $u \in \{p, a_1, \dots, a_n\}$ to $\mathbf{c}$, in this order. If a neighbor n of u is also colored with $\mathbf{c}$, we annotate n with the former color of u. This may provoke other conflicts which are then resolved recursively. Swapping the color of a node v initiated by changing the color of u to $\mathbf{c}$ ends in one of the four cases:

1. Changing v's color does not generate new conflicts.
2. Register constraints forbid assigning v some color $\mathbf{d}$. This conflict, caused by the intention of changing u's color to $\mathbf{c}$, cannot be resolved. This is indicated by adding the edge uu to the conflict graph $C_{\mathbf{c}}$. Thus, u is excluded from every possible stable set of $C_{\mathbf{c}}$. So, $\mathbf{c}$ is not assignable to u. If $u = p$ then the entry is discarded (since trying to assign $\mathbf{c}$ to the a_i is not sensible if the color of p is unequal to $\mathbf{c}$). Otherwise, the entry is reinserted into the priority queue after recomputing $S_{\mathbf{c}}$.
3. v's color has already been pinned (see phase **Apply**) by the processing of another optimization unit. Then, changing v's color would increase the costs incurred by this other OU. uu is added to $C_{\mathbf{c}}$ and the entry is reinserted into the queue as described above.
4. If v is a pinning candidate for the current OU, u and v are somehow interdependent. The algorithm cannot assign $\mathbf{c}$ to u and v at the same time. As we require p to be always contained in each $C_{\mathbf{c}}$, if $v = p$ then we add the edge uu, otherwise the edge uv to $C_{\mathbf{c}}$. Afterwards, $S_{\mathbf{c}}$ is recomputed and the entry is re-inserted into the queue.

If all conflicts caused by changing u's color to $\mathbf{c}$ have been resolved (all ended in case 1), then u is marked as a *pinning candidate*, else all color annotations caused by re-coloring u are discarded.

If all $p, a_1, \ldots, a_n$ are marked as pinning candidates, testing ends for this OU.

Apply

If the testing phase produced at least two pinning candidates (some a_i and p could be colored with the same color), the pinning candidates become *pinned* and all color changes annotated by the pinning phase are applied to G.

Note, that the **Test**-Phase always terminates, since in each step an edge is added to the conflict graph, if testing was not successful. Thus, in the worst case, the stable set will finally consist of the Φ-result only and is *not* re-inserted into the priority queue. Thus, the whole algorithm terminates.

5.4 Φ-optimization using integer linear programming

In this section we describe a method yielding optimal solutions for SSA-Maximize-Fixed-Points. Let $G = (V, E, Q)$ be the interference graph augmented with edges Q that indicate a request for same colors of the adjacent nodes.

5.4.1 Formalization

Since every solution is also a valid coloring, we start with 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. x_{ic} shall be 1, iff node v_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. For chordal graphs this can be modelled efficiently with facet defining clique constraints, since a minimum clique cover is computable in $O(|V|^2)$ for chordal graphs (see [Gav72]). For each color c and each clique C in a given (minimal) clique cover $\mathcal{C}$, we add the following constraint:

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

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 same-color-requests. For each edge $v_i v_j \in Q$ the binary variable y_{ij} shall indicate, whether the incident nodes have the same color (0), or different colors (1). To model the positive costs ω_{ij} arising from different colors we add the term $\omega_{ij} y_{ij}$ to the objective function being minimized.

To interconnect the y with the x variables we add two constraints per color and edge in Q. Since the y_{ij} are automatically pulled to 0 by minimization, we only must enforce $y_{ij} = 1$, if v_i and v_j have different colors:

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

Though this is a complete formalization of the problem, one can improve it by reducing the model size, or by adding constraints pruning the search space.

5.4.2 Improvements

To reduce the number of variables and constraints, we perform another step before building the ILP. We remove the maximum number of simplicial nodes from G, which are not concerned with equal-color-requests (not adjacent to edges in Q). These nodes build the beginning of a PEO and thus can be colored after the solution of the ILP is applied. Now the ILP only has to deal with the "core" of the problem.

Finally, we want to present two classes of inequalities which can be used to restrict the search space: *Path*-inequalities and *Clique-Path*-inequalities. The basic idea for both classes is to provide lower bounds for the costs coming from the contrariness of equal-color-request and interference-edges.

Definition 2. *We call two nodes a, b equal-color-connected, $ecc(a, b)$, if there is a path of edges in Q connecting them, and no inner node of this path interferes with another node of the path. Formally: $\exists v_1, \ldots, v_n \in V$:*

- $a = v_1, b = v_n$
- $\forall 1 \leq i < n : v_i v_{i+1} \in Q$
- $\forall 1 \leq i < j \leq n : v_i v_j \in E \Rightarrow \{v_i, v_j\} = \{a, b\}$

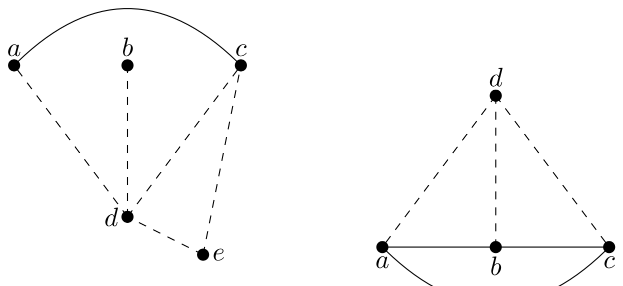
If $ab \in E$ and $ecc(a, b)$ with path $v_1, \dots, v_n$, the following *Path*-inequality is valid:

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

The second class uses a clique $C = \{v_1, \dots, v_n\}$ in the interference graph combined with a node $a \notin C$. If all $av_i \in Q$ the following *Clique-Path*-inequality is valid:

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

At first sight, the conditions of the second class seem to be very special, but especially in the case where an argument appears multiply in the same column of a Φ all of them are satisfied.



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

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

(a) Path-inequalities

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

(b) Clique-Path-inequality

Figure 6: Examples for the two classes of inequalities.

Register constraints can be integrated in the model by either setting the forbidden x_{ic} to 0, or by omitting the corresponding variables and adjusting some constraints.

6 Register constraints

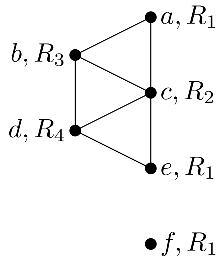
Almost all processor architectures impose register constraints to some of their instructions. Often, the register assignable to an operand of an instruction is fixed. Graph coloring register allocators usually solve this problem by

1. splitting live range of that variable at the location of constrained occurrence by inserting copies to let it change its register there.
2. adding pre-colored nodes for each register in the register class.
3. Connecting the node of the constrained variable with all pre-colored nodes but the one which represents the constraint color.

Note that omitting the live range splitting, even if two variables with the same constraint do not interfere, could raise the register demand unnecessarily as shown in figure 7.

$$\begin{array}{rcl}
a_{R_1} & \leftarrow & \dots \\
b & \leftarrow & \dots \\
c & \leftarrow & b+1 \\
d & \leftarrow & a+1 \\
e_{R_1} & \leftarrow & b+c \\
f & \leftarrow & c+d \\
 & & \dots
\end{array}$$

(a) Program R with constraints



(b) Colored interference graph

Figure 7: Program with constraints and its interference graph

Unfortunately, coloring chordal graphs with pre-colored nodes is only in $\mathcal{P}$, iff each color is used only once in a pre-coloring of the graph (see [Mar04] and also [BHT92]). However, in register allocation, constraint colors often are used multiple times. Therefore we delegate this problem to the *Perm*-Optimizer which cares about minimizing the transpositions in *Perm*s inserted for Φ-operations as described in section 5:

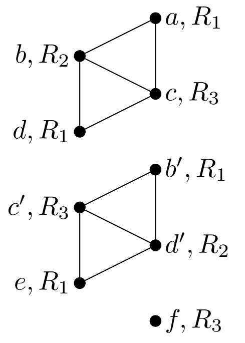
We insert a Φ with a single argument column right in front of each instruction imposing register constraints on its operands or results. This expresses, that all variables live in front of the constrained instruction are able to change their register there. Thus, the interference graph breaks in two completely unconnected components (see figure 8). This ensures, that in each component, each color occurs only once as a pre-coloring. Along the way, this also solves the problem of two interfering nodes which must reside in the same register and never causes the register demand to exceed the number of registers available.

Coloring the operands of the instruction imposing the constraints is now easy. Consider algorithm 1: Arriving at a Φ inserted due to register constraints, the set of used colors is empty, since each value stops living at the Φ. The values which get colored next are the results of that Φ followed by the results of the constrained instruction. Since all colors are available, one can select the coloring of these variables freely. Finally, we mark the color of the constrained variable as not changeable to pass this information to the *Perm*-Optimizer.

It is then up to the *Perm*-Optimizer to find a coloring in which as many operands and results as possible are assigned the same color.

$$\begin{array}{rcl}
a_{R_1} & \leftarrow & \dots \\
b & \leftarrow & \dots \\
c & \leftarrow & b+1 \\
d & \leftarrow & a+1 \\
\begin{pmatrix} b' \\ c' \\ d' \end{pmatrix} & \leftarrow & \Phi \begin{bmatrix} b \\ c \\ d \end{bmatrix} \\
e_{R_1} & \leftarrow & b'+c' \\
f & \leftarrow & c'+d' \\
 & & \dots
\end{array}$$

(a) Program R' with constraints



(b) Colored interference graph of R'

Figure 8: Program with constraints and its interference graph

7 Conclusions

SSA-form programs allow for a completely new architecture of register allocators. Due to the chordality of their interference graphs, the different phases of register allocation (spilling, coloring, coalescing) can be completely decoupled, thus avoiding the feedback based approach in common graph coloring register allocators. In this report we gave an outline of such a new register allocator architecture and proposed novel solution methods for each of the three phases mentioned above.

Based on the direct correspondence of the live sets in the program to the cliques in the program's interference graph, we showed how an already existing, heuristic method for spilling in basic blocks can be extended to work on a whole procedure. Additionally, we presented an integer linear programming formulation providing optimal solutions for the spilling problem. Both methods work *without* constructing the interference graph.

Furthermore, we showed that an optimal coloring of the interference graph can be obtained in linear time (assuming dominance and live ranges have already been computed), also without constructing the interference graph itself.

Finally, we investigated the problem of copy coalescing, which we find is identical to the removal of Φ-operations. We proved this problem to be $\mathcal{NP}$-complete and gave a heuristic, as well as an optimal solution method. Furthermore, we showed how common register constraints can easily be expressed in this setting.

8 Related Work

Besides our work, the power of chordal graphs for register allocation has been discovered by several people independently.

Brisk [Bri05] independently developed a proof of the chordality of SSA-form program's interference graphs, which is quite similar to ours. He utilizes standard techniques from chordal graph theory to obtain an optimal coloring of the interference graph. He does not consider spilling and coalescing.

Bouchez [Bou05] also recognizes the chordality of SSA-form programs' interference graphs and gives an extensive analysis of the complexity of the spilling problem in his master thesis.

Pereira and Palsberg [PP05] investigate interference graphs of non-SSA programs in a Java compiler and find that 95% of them are chordal. They furthermore present heuristics for spilling and coalescing which, since they work on non-SSA programs, cannot exploit the special properties of SSA-form programs and their interference graphs.

Although not considering chordal graphs, Appel and George [AG01] propose a non-feedback based approach for register allocation of non-SSA programs using integer linear programming in which spilling, coloring and coalescing are decoupled. They explicitly address processors with a small number of registers to keep the linear programs solvable in a reasonable time.

9 Acknowledgements

We want to thank our colleagues Michael Beck, Marco Gaertler, Rubino Geiß and Götz Lindenmayer for many fruitful discussions. We also thank Alan Mycroft and Simon Peyton Jones for many helpful comments.

A Graph theory terminology

Let $G = (V, E)$ be an undirected graph. We call a graph G *complete,* iff for each $v, w \in V_G$, there is an edge $vw \in E_G$ and denote it by K^n, $n = |V_G|$. We call $H = G[V_H]$ an induced subgraph of G, if $V_H \subseteq V_G$ and for all nodes $v, w \in V_H$, $vw \in E_G \iff vw \in E_H$ holds. H is called a clique if H is complete and $H \subseteq G$ for some G. $\omega(G)$ is the size of the largest clique in G. A graph $G = (V, E)$ with $V = \{v_1, \dots, v_n\}$ and $E = \{v_1v_2, \dots, v_{n-1}v_n, v_nv_1\}$ is called a cycle and is denoted by C^n. A set of nodes $\{v_1, \dots, v_n\} \subseteq V_G$ is called a stable set, iff for each v_i, v_j there is $v_iv_j \notin E_G$.

A *coloring* is a partition of V_G into subsets $C_1, \dots, C_k$ whereas $v, w \in C_m$ implies that $vw \notin E_G$. The *chromatic number* $\chi(G)$ is the smallest k for which $C_1, \dots, C_k$ is a coloring of G.

Definition 3. *A graph G is called* perfect, *iff $\omega(H) = \chi(H)$ for each $H \subseteq G$.*

Definition 4. *A graph G is called* chordal *iff it does not contain any induced C^n for $n \geq 4$.*

B Permutations

A bijective mapping σ over a set X is called a *permutation.* An $i \in X$ for which $\sigma(i) = i$, is called a *fixed point* of σ. A cyclic permutation ζ is a permutation for which there is an $i \in X$ and a $|\zeta| \in \mathbb{N}$ for which $\zeta^{|\zeta|}(i) = i$ and for each $j \in X \setminus \{i, \zeta(i), \dots, \zeta^{|\zeta|}(i)\}$, $\zeta(j) = j$. A cyclic permutation is written by giving the members of the cycle:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 3 & 4 & 2 \end{pmatrix} \quad \Rightarrow \quad (2\ 3\ 4)$$

Each cycle can be decomposed into cycles of length two, called *transpositions,* in the following way:

$$(i_1\ i_2\ \cdots\ i_k) \quad \Rightarrow \quad (i_1\ i_2) \cdot (i_2\ i_3) \cdots (i_{k-1}\ i_k)$$

A fixed point is a cycle of length 1. Each permutation is uniquely determined by a product of disjoint cycles, up to isomorphy between the cycles since (1 2 3), (2 3 1), (3 1 2) denote the same cycle.

C Pseudocode of the Perm-Optimizer

```
procedure COALESCE($G$)
    for $\begin{pmatrix} p_1 \\ \vdots \\ p_m \end{pmatrix} = \Phi \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}$ do
        for i=1,...,m do
            Create an $OU$ consisting of $p_i$ and all $a_{ij}$ not interfering with $p_i$
        od
    od

    $pinned \leftarrow \emptyset$
    for optimization unit $OU = (p, a_1, \ldots, a_k)$ do
        create priority queue $Q$                                    ▷ Init
        for colors $\mathbf{c}$ assignable to $p$ do
            $C_\mathbf{c} \leftarrow G[p, a_1, \ldots, a_k]$
            $S_\mathbf{c} \leftarrow$ maximum weighted independent set of $C_\mathbf{c}$
            Insert entry $(\mathbf{c}, C_\mathbf{c}, S_\mathbf{c})$ into $Q$
        od

        repeat                                                       ▷ Test
            $candidates \leftarrow \emptyset$
            $g \leftarrow f$                          ▷ Copy current coloring $f$ to $g$
            pop $(\mathbf{c}, C, S)$ from $Q$
            $C' \leftarrow$ TEST($\mathbf{c}$, $C$, $S$)
            if $C' \neq$ nil then
                $S' \leftarrow$ maximum weighted independent set of $C'$
                Insert entry $(\mathbf{c}, C', S')$ into $Q$
            fi
        until $C' =$ nil

        if $|candidates| > 1$ then                                    ▷ Apply
            $pinned \leftarrow pinned \cup candidates$
            $f \leftarrow g$                    ▷ Put temporary coloring $g$ into effect
        fi
    od
end
```

```
function Test($\mathbf{c}$, $C$, $S$)
    for $u \in S$ do
        $(s, v)$ ←TryColor($u$, nil, $\mathbf{c}$)
        if $s =$ ok then
            $candidates \leftarrow candidates \cup u$
        else
            if $s =$ candidate and $v \neq p$ then
                $C' = (V_C, E_C \cup vu)$
            else
                $C' = (V_C, E_C \cup uu)$
            fi
            return $C'$
        fi
    od
    return nil
end
```

```
function TryColor($v \in V_G$, $u \in V_G$, $\mathbf{c}$)
    $\mathbf{c}_v \leftarrow g(v)$
    if $\mathbf{c} = \mathbf{c}_v$ then                        ▷ The color of $v$ is already $\mathbf{c}$
        return (ok, nil)
    else if $v \in pinned$ then                   ▷ $v$ has been pinned by another $OU$
        return (pinned, $v$)
    else if $v \in candidates$ then                    ▷ $v$ has already been tested
        return (candidate, $v$)
    else if $\mathbf{c}$ is not allowed for $v$ then           ▷ Due to register constraints
        return (forbidden, $v$)
    fi

    for $\{n \mid vn \in E_G, n \neq u, \mathbf{c} = g(n)\}$ do          ▷ Look at all conflicting
                                                          neighbors of $v$
        $(s, v')$ ←TryColor($n$,$v$,$\mathbf{c}_v$)             ▷ Try to give a neighbor $v$'s color
        if $s \neq$ ok then
            return $(s, v')$
        fi
    od
    $g(v) \leftarrow \mathbf{c}$
    return (ok, nil)
end
```

D Pseudocode of spilling with Belady's method

Algorithm 2 Belady's algorithm for a basic block

```
function DISPLACE(Set $Q$, Label $\ell$, Function $nextuse$, Variables $V$ )
    $R \leftarrow V \setminus Q$                               ▷ $R$ contains all elements of $V$ not in $Q$
    $X \leftarrow \emptyset$                                   ▷ Use $X$ to record all variables displaced from $Q$
    for $i = 1 \ldots \max\{|R| + |Q| - k, 0\}$ do             ▷ Remove as many variables from $Q$ so that $R$ can be added to $Q$ preserving $|Q| \leq k$
        $w \leftarrow \arg\max_{v \in Q} nextuse(\ell, v)$
        $X \leftarrow X \cup w$
        $Q \leftarrow Q \setminus w$
    od
    $Q \leftarrow Q \cup V$
    return $X$
end

procedure BELADY-BLOCK(Basic block $B$)
    $I \leftarrow livein_B \cup$ results of $\Phi$ in $B$
    $I \leftarrow k$ smallest members of $I$ concerning $nextuse$
    $in_B \leftarrow I$                                        ▷ $in_B$ shall contain all variables which are already in registers upon entering $B$.
    $Q \leftarrow I$
    $U \leftarrow \emptyset$                                   ▷ If a variable is used at some label, it is put into $U$

    for each label $\ell : (y_1, \ldots, y_m) \leftarrow \tau(x_1, \ldots, x_n)$ in $B$ do
        Insert reloads for all $x \in \{x_1, \ldots, x_n\} \setminus Q$
        $X \leftarrow$ DISPLACE($Q, \ell, nextuse, \{x_1, \ldots, x_n\}$)
        $in_B \leftarrow in_B \setminus [X \setminus U]$       ▷ If a value live in is dis-. placed before it is used, it can be removed from $in_B$ since it is not necessary to hold it in a register upon entering the block
        $U \leftarrow U \cup \{x_1, \ldots, x_n\}$
        DISPLACE($Q, \ell, nextuse', \{y_1, \ldots, y_m\}$)
    od
    $out_B \leftarrow$ variables in $Q$
end
```

Algorithm 3 Belady's algorithm for spilling

```
procedure SPILL-BELADY(Procedure $\pi$, Number of registers $k$)
    for each basic block $B$ of $\pi$ do
        BELADY-BLOCK($B$)
    od
    for each basic block $B$ of $\pi$ do
        for $P \in pred_B$ do
            Insert reloads on control flow edge $P \to B$ for all $v \in in_B \setminus out_P$
        od
    od
end
```

References

[AG01] Andrew W. Appel and Lal 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.

[App98] Andrew W. Appel. SSA is Functional Programming. *ACM SIGPLAN Notices*, 33(4):17–20, April 1998.

[BCH$^+$02] Zoran Budimlić, Keith D. Cooper, Timothy J. Harvey, Ken Kennedy, Timothy S. Oberg, and Steven 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.

[BDEO97] Peter Bergner, Peter Dahl, David Engebretsen, and Matthew O'Keefe. Spill code minimization via interference region spilling. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 287–295, New York, NY, USA, 1997. ACM Press.

[Bel66] L. Belady. A Study of Replacement of Algorithms for a Virtual Storage Computer. *IBM Systems Journal*, 5:78–101, 1966.

[BHT92] M. Biró, M. Hujter, and Zs. Tuza. Precoloring extension. I. Interval graphs. *Discrete Mathematics*, 100:267–279, 1992.

[Bou05] Florent Bouchez. Allocation de registres et vidage en mémoire. Master's thesis, ÉNS Lyon, 2005.

[Bri05] Philip Brisk. Personal communication. 2005.

[CAC$^+$81] 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.

[CFR$^+$91] 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, October 1991.

[CH90] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.*, 12(4):501–536, 1990.

[Gav72] Fănică 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.

[GGP03] Jia Guo, Maria Jesus Garzaran, and David Padua. The Power of Belady's Algorithm in Register Allocation for Long Basic Blocks. The 16th International Workshop on Languages and Compilers for Parallel Computing, 2003.

[Gol80] Martin Charles Golumbic. *Algorithmic Graph Theory And Perfect Graphs*. Academic Press, 1980.

[Hac05] Sebastian Hack. Interference Graphs of Programs in SSA-form. Technical report, Universität Karlsruhe, June 2005.

[Kel95] Richard A. Kelsey. A Correspondence Between Continuation Passing Style and Static Single Assignment Form. *ACM SIGPLAN Notices*, 30(3):13–22, 1995.

[Mar04] Dániel Marx. *Graph Coloring with Local and Global Constraints*. PhD thesis, Budapest University of Technology and Economics, 2004.

[PP05] Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In *Proceedings of APLAS'05*, November 2005.

[RdFG03] Fabrice Rastello, Francois de Ferrire, and Christophe Guillon. Optimizing translation out of ssa with renaming constraints. Technical Report RR-03-35, LIP, ENS Lyon, June 2003.