Register allocation for programs in SSA-form

Sebastian Hack, Daniel Grund, and Gerhard Goos

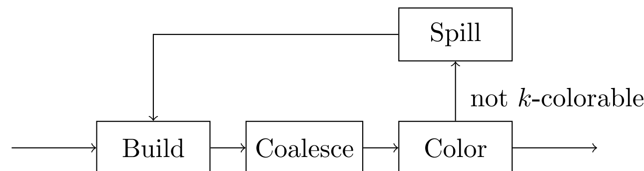
Fakultät für Informatik
Universität Karlsruhe
(hack|daniel|ggoos)@ipd.info.uni-karlsruhe.de

Abstract. As register allocation is one of the most important phases in optimizing compilers, much work has been done to improve its quality and speed. We present a novel register allocation architecture for programs in SSA-form which simplifies register allocation significantly. We investigate certain properties of SSA-programs and their interference graphs, showing that they belong to the class of chordal graphs. This leads to a quadratic-time optimal coloring algorithm and allows for decoupling the tasks of coloring, spilling and coalescing completely. After presenting heuristic methods for spilling and coalescing, we compare our coalescing heuristic to an optimal method based on integer linear programming.

1 Introduction

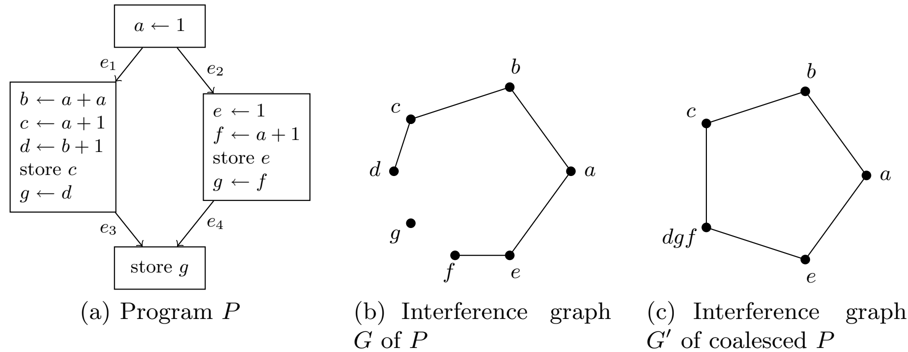
Graph coloring register allocation has been a successful approach for register allocation, mostly due to its very simple abstraction: Each variable in the program is mapped to a node in an undirected, so called *interference graph.* Whenever the compiler finds out that two variables cannot be held in the same register (they are *simultaneously live*), an edge is drawn between the two nodes in the interference graph representing the two variables. A k-coloring of the interference graph thus leads to a valid register allocation using at most k registers.

Chaitin [1] showed that for each undirected graph G, there is a program which has G as its interference graph. Since graph coloring is $\mathcal{NP}$-complete, so is register allocation. This leads to the well known iterative approach of graph coloring register allocators (here, we illustrate a simplified version of the allocator proposed by Briggs [2]):



Since determining the graph's chromatic number (the minimal number of colors needed for a valid coloring) is also $\mathcal{NP}$-complete, the impact of a modification of the graph (spilling and coalescing) on its colorability cannot be determined efficiently in general. This has two unappealing consequences:

1. Coalescing (the task of eliminating useless copies) may do more harm than good by increasing the chromatic number of the graph. Consider the example program P in figure 1(a) and its interference graph G in figure 1(b). G's chromatic number $\chi(G)$ equals 2. Aggressive coalescing would merge the nodes d, g, f into one producing the graph G' (shown in figure 1(c)) which is not 2-colorable anymore. Thus, the register demand of P is raised by merely removing some copies and thus possibly introducing spill code.



(a) Program P (b) Interference graph G of P (c) Interference graph G' of coalesced P

Fig. 1. Program P and its interference graph

2. Since it is not clear if the spilling of a node improved the colorability of the graph, the modifications of the program caused by spilling have to be materialized, the interference graph has to be rebuilt and coloring has to be attempted again. Thus, coloring is repeated until k colors suffice. Especially for a small number of available registers many iterations have to be expected, since the number of spills will be high. This is costly, since the interference graph is a large data structure which then has to be rebuilt over and over.

The situation drastically changes if we require the processed program to be in SSA-form. As we show in section 2, interference graphs of SSA-form programs are *chordal.* The two major properties of chordal graphs which make them so appealing for register allocation are:

1. Their chromatic number is equal to the size of the largest clique in the graph.
2. They can be optimally[1] colored in quadratic time (concerning the number of nodes, cf. Golumbic [3]).

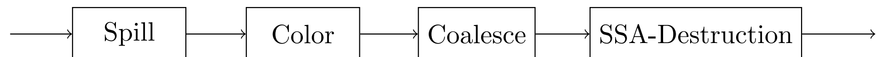
Furthermore, there are several relations between SSA-form programs and their interference graphs which allow us to circumvent the deficiencies of conventional graph coloring register allocators as mentioned above:

---

[1] Using as few colors as possible

– Cliques in the interference graph correspond to live sets in the program. This means after liveness analysis we know how many registers will be needed for the program in question. If we reduce the amount of variables live at each point in the program to at most k, the graph will be k-colorable, which eliminates the iteration. In section 4.1, we present a simple algorithm which splits the live ranges of the variables so that the register pressure is at most k at each point in the program.
– Dominance, a fundamental notion for SSA-form programs, induces an order of the interference graph's nodes which allows the interference graph $G = (V, E)$ of a SSA-form program to be colored optimally in $O(\chi(G) \cdot |V|)$ as shown in section 2.
– Finally, as shown in section 4.3, we coalesce useless copies in the shape of ϕ-operations not by modifying the graph but by finding a k-coloring which assigns as many sources and targets of copies the same register. This preserves the chordality of the interference graph and thus does not change its k-colorability. So coalescing a copy will never cause any additional spill.

This leads to a single pass register allocator architecture looking like



avoiding any iteration.

2 SSA-form Programs and their Interference Graphs

Before going into algorithmic details, let us discuss basic properties of SSA-form programs and their connection to relevant terms of register allocation like liveness and interference.

We consider a program as a standard CFG being a triple (*Labels*, *CF*, **start**). Each label $\ell \in \mathit{Labels}$ contains a single instruction

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

a set of control flow edges *CF* between the labels and one designated label **start** which has no control flow predecessors. As we only consider SSA-form programs from now on, each variable v has a unique label where it is defined. We will denote this label by $\mathcal{D}_v$.

A fundamental notion for SSA-form programs is the one of dominance:

Definition 1 (Dominance). *A label ℓ dominates a label ℓ' if all paths from* **start** *to ℓ' contain ℓ. We then write $\ell \preceq \ell'$.*

Essential for all later work is the notion of a strict program which was coined by Budimlić [4].

Definition 2 (Strict program). *A program is* strict*, if each usage of a variable v is dominated by $\mathcal{D}_v$.*

The interference graph $G = (V, E)$ of a program P contains all variables occurring in P as nodes. Two variables v and w are connected by an edge (we then write $vw \in E$) in G, iff they interfere:

Definition 3 (Interference). *We say, two variables* interfere *if there exists a label in the program where they are both live.*

In the same paper, Budimlić gave two lemmas which establish a fundamental relationship between dominance and interference:

Lemma 1. *If two variables v, w interfere either $\mathcal{D}_v \preceq \mathcal{D}_w$ or $\mathcal{D}_w \preceq \mathcal{D}_v$.*

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

Based on Budimlić's lemmas we can prove our first claim of the introduction:[2]

Theorem 1. *For each clique $C = \{c_1, \ldots, c_n\} \subseteq V$ in the interference graph $G = (V, E)$ of a SSA-form program P, there exists a label $\ell \in Labels_P$ where all $c_1, \ldots, c_n$ are live.*

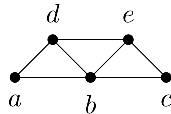
Proof. Since C is a clique, $(c_i, c_j) \in E$ for each $1 \leq i < j \leq n$. By lemma 1, the labels $\{\mathcal{D}_{c1}, \ldots, \mathcal{D}_{cn}\}$ form a totally ordered set. Thus there exists a permutation $\sigma : C \longrightarrow C$ for which $\mathcal{D}_{\sigma(c_1)} \preceq \cdots \preceq \mathcal{D}_{\sigma(c_n)}$. By lemma 2, $\sigma(c_1), \ldots, \sigma(c_n)$ are live at $\mathcal{D}_{\sigma(c_n)}$.

3 Coloring SSA Interference Graphs

Consider the following method to color a graph: Given an order $v_1, \ldots, v_n$ of the graph's nodes. Eliminate the v_i one by one from the graph. Then, re-insert the nodes in reverse order and give each v_i the first free color not used by its already re-inserted neighbors.

A well-known result from graph theory states that for each graph $G = (V, E)$ there is an ordering of all nodes in V for which this procedure leads to an optimal coloring of G (cf. the textbook of Diestel [6] for example). In general, as graph coloring is $\mathcal{NP}$-complete, determining such a sequence is also $\mathcal{NP}$-complete.

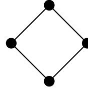
For the moment, let us consider the following approach to generate such an ordering: In each elimination step, search a node v whose neighbors form a clique in the current graph (such a node is also called *simplicial*). The idea is, that when the node is re-inserted, all neighbors which are already colored form a clique, and thus the number of colors used for the coloring is bound by the size of the largest clique in the graph. Such an elimination order is called a *perfect elimination order (PEO).* Consider the following example:



PEO: a, d, b, e, c
No PEO: b, a, c, d, e

[2] Bouchez [5] gave this theorem, independently from us, too.

Of course, not every graph allows to find such a node whose neighbors form a clique at each step in the elimination process. For instance, the diamond graph



used as an example by Briggs in [2] does not allow for perfect elimination order. It is a well-known theorem of the theory of perfect graphs, that if a graph possesses a perfect elimination order, the coloring procedure described above will generate an optimal coloring of the graph (cf. the textbook of Golumbic [3] for example).

Based on Budimlić's lemmas, we prove that the dominance relation of a program in SSA-form induces a perfect elimination order of its interference graph $G = (V, E)$.

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

Proof. By contradiction: due to lemma 1, either $\mathcal{D}_b \preceq \mathcal{D}_c$ or $\mathcal{D}_c \preceq \mathcal{D}_b$. Assume $\mathcal{D}_c \preceq \mathcal{D}_b$. Then (by lemma 2), c is live at $\mathcal{D}_b$. Since a and b also interfere and $\mathcal{D}_a \preceq \mathcal{D}_b$, a is also live at $\mathcal{D}_b$. So, a and c are live at $\mathcal{D}_b$ which cannot be by precondition.

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

Thus, a PEO of a SSA interference graph's nodes can be easily obtained by a post order walk over the program's dominance tree. Thus, we can optimally color the interference graphs of SSA-form programs in quadratic time.

The graphs, for which perfect elimination orders exist are called *chordal* graphs or sometimes triangulated or rigid-circuit graphs. Since chordal graphs are *perfect* (cf. to [3]) the characteristic property of perfect graphs also applies to chordal graphs:

Definition 4. *A graph H is perfect, iff for each induced subgraph H of G the chromatic number $\chi(H)$ is equal to the size of the largest clique $\omega(H)$.*

4 A Register Allocator for SSA-form Programs

Before giving a detailed description of spilling and coalescing techniques in the next subsections let us briefly outline how the theoretical results of the last

section can be exploited to derive a new architecture for register allocators in general.

Theorem 2 together with definition 4 state that the chromatic number of a SSA interference graph is determined by the largest clique in the graph. By theorem 1, for each clique in the interference graph, there is a label in the program, where all variables in the clique are live. Thus, spilling can make the interference graph k-colorable by reducing the number of live variables at each label to k. This enables us to consider the spilling problem separately from the other tasks of a register allocator since checking how many variables are live at all labels in the program is easy in contrast to determining the chromatic number of an arbitrary graph. In section 4.1 we demonstrate how a well known basic block oriented spilling technique can be extended to serve as a spilling method for the whole program.

By section 3, obtaining an optimal k-coloring is trivial. All one has to do is to obey the coloring sequence induced by the dominance relation. Section 4.2 shows how the ϕ-operations can be removed to obtain a non-SSA program having a valid register allocation with k registers.

We consider coalescing as the task of obtaining a *good* coloring with respect to ϕ-operations. Consider a ϕ-operation $y \leftarrow \phi(x_1, \ldots, x_n)$. If we can assign as many of the x_i the color of y, we save move operations on the respective edges to the ϕ's block. The advantage over merging the node of y with the nodes of the x_i in the interference graph is, that we do not modify the graph's structure (i.e. possibly rendering it non-chordal) which lets us still determine its chromatic number easily.

4.1 Spilling

In conventional global register allocation (like the register allocator by Briggs [2]), spilling is not activated until coloring fails. Thus, the spilling decision is tightly coupled to the way the graph is colored: If a node is popped from the coloring stack and there is no color left to assign since its neighbors use up all available colors, one of its neighbors is marked to be spilled, i.e. each use is preceded by a reload and each definition is succeeded by a store of its value. This breaks the live range apart making the variable only interfere with the variables live at the usages and definitions. So the node is spilled only because another one cannot be colored.

Since the interference graph represents the live ranges of variables, it hides relevant information concerning spilling:

- How often is a variable used?
- Where is a variable used?
- How far is the next use away from a given point?

Thus spilling in conventional global register allocation is *only* concerned with modifying the graph's structure in order to make it k-colorable. A lot of work has been done to make these register allocators more sensitive to the program structure (see e.g. the work by Bergner et al. [7] or by Chow and Hennessy [8]).

However, theorems 1 and 2 and definition 4 allow for using more program-sensitive, basic block oriented spilling approaches like Hsu et al. [9] and combine their results to a solution for the whole procedure. Guo et al. [10] describe the power of Belady's MIN algorithm [11] for spilling in a basic block. Belady's algorithm does *not* minimize the number of loads or stores in a basic block. Though, as the measurements of Guo show, it is still a good heuristic. In the following, we present a method how Belady's algorithm can be extended to work on a whole procedure by using the results of section 2.

Belady's MIN Algorithm The main principle of Belady's MIN algorithm is to displace the variables from registers whose next use is farthest in the future (regarding the number of instructions). The algorithm starts at the entry of a basic block B and visits each label ℓ in the block once. Assume, that all operands of the instruction of ℓ are read/written from/to registers. If all registers are occupied, one variable has to be displaced from the registers to make room for the result of the instruction. If a label is reached whose instruction uses a value which has been displaced, a reload must be inserted for this variable and, since the reload loads the value in a register, another variable may have to be displaced from the register set.

For example, you have 4 registers which are currently occupied by the variables a, b, c, d. Reaching a label

$$\ell : f \leftarrow \tau(a, e)$$

one register has to be freed to reload the variable e. The algorithm of Belady selects the one of b, c, d whose next use is farthest away from ℓ. Two questions arise immediately:

1. How far away is the next use of a variable v which is live out at the block B of consideration but not used in that block anymore?
   Since v can be used on several different control flow paths from the block, it is not clear when v will be used next since this depends on the taken control flow successor of B. Therefore we use an estimation by taking the minimum of all next use distances.
2. What is the initial occupation of the registers?
   Let us consider the set I_B containing all values live in at B and the results of all ϕ-functions in B. All these values are passed to this block "from outside".[3]
   If $|I| > k$, we select k elements from I_B with the nearest next uses.
   Furthermore, if we find out that a variable v in I_B is displaced before it is used, it is not sensible to hold v in a register at the entry of the block, thus v is removed from I_B.

We record the occupation of the registers after the last instruction in the block B in the set O_B.

[3] Note that a ϕ is just a representative for a control flow dependent live in.

The final step is to combine the results of the algorithm applied to all basic blocks in the program into a solution for the whole procedure. Since the register pressure is nowhere larger than k, we only have to assure that all variables in I_B for some block B are in registers on each control flow edge leading to B. Thus we examine each predecessor block P of B: If $M := I_B \setminus O_P$ is not empty, we have to insert reloads for all variables in M on the control flow edge from P to B.[4]

Note that spilling a (SSA-)variable v and reloading it several times actually destroys the SSA-form of the program, since v has then multiple definitions, i.e. the reloads v. The SSA-form can be reconstructed by applying a SSA construction algorithm, e.g. the one by Cytron et al. [12].

4.2 SSA-Destruction

A ϕ-operation $y \leftarrow \phi(x_1, \ldots, x_n)$ works like a control flow dependent copy operation assigning x_i to y if the ϕ's label is reached via the i-th control flow predecessor. Furthermore, SSA semantics state that all ϕ-operations in a basic block have to be executed simultaneously before all other instructions in that basic block. Thus, all ϕ-operations

$$
\begin{aligned}
y_1 &\leftarrow \phi(x_{11}, \ldots, x_{1n}) \\
&\cdots \\
y_m &\leftarrow \phi(x_{m1}, \ldots, x_{mn})
\end{aligned}
$$

in a block work as a "bulk copy" copying the x_{ij} to the y_i *at once* if the block was entered via the j-th edge.

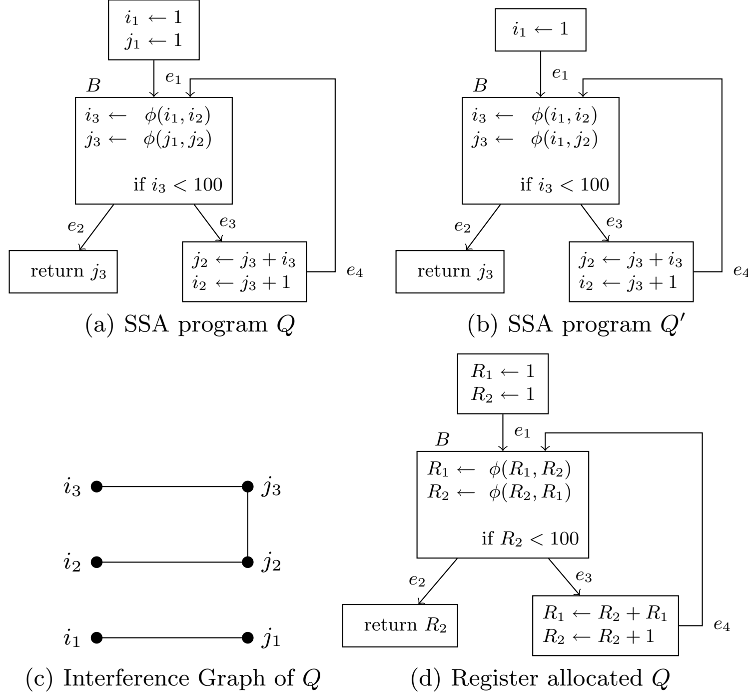
Conventionally, while translating out of the SSA-form, ϕ-operations are replaced by copy instructions. Despite some other problems like the swap-problem (see Briggs et al. [13]), this kind of ϕ removal may raise the register demand unnecessarily as demonstrated by the example program Q in figure 2(a): Replacing the ϕ-operations by inserting the copies

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

on the edge e_4 introduces an interference between i_3 and j_2 which was not present in the SSA interference graph shown in figure 2(c). This edge creates the clique i_3, j_2, j_3 which raises the graph's chromatic number to 3.

So let us reconsider the bulk copy property of the ϕ-operations in a basic block. Consider the register allocation of Q shown in figure 2(d). If the block B is entered via e_1, R_1 is assigned R_1 and R_2 is assigned R_2, so the ϕs do nothing on this edge. However, if B is reached via e_4, R_1 is assigned R_2 and vice versa, *at once:* The registers R_1 and R_2 are swapped.

[4] Instructions can be placed on a control flow edge by eliminating critical edges and putting the instruction in the respective block.



(a) SSA program Q (b) SSA program Q'

(c) Interference Graph of Q (d) Register allocated Q

Fig. 2. Example programs Q and Q'

So generally ϕ-operations work like permutations on registers and not like a set of copies. It depends on the registers allocated for the results of the ϕ and its operands how the permutation will look like. Thus, in this setting, coalescing is the task of finding a register allocation in which the permutations will have as many *fixed points* (registers that are mapped to themselves) as possible.

As known from basic linear algebra, each permutation of size n can be written as a sequence of *transpositions* (swaps) and thus is implementable using n registers, using *no* extra register. For example, the ϕ-operations at some label ℓ

$$\begin{aligned} R_2 &\leftarrow \phi(\ldots, R_1, \ldots) \\ R_3 &\leftarrow \phi(\ldots, R_2, \ldots) \\ R_1 &\leftarrow \phi(\ldots, R_3, \ldots) \\ R_4 &\leftarrow \phi(\ldots, R_4, \ldots) \end{aligned}$$

can be implemented by inserting the sequence

```
swap R2, R3
swap R1, R2
```

on the corresponding control flow edge entering ℓ.

If the processor provides a swap instruction (like `xchg` on the x86), ϕ-operations can be directly be implemented by a sequence of these. If not, one can use three exclusive ors. However, if one register is spare at the ϕs, we can use it to implement the permutation with moves. Note that due to theorem 1 and definition 4 we exactly know how many registers are in use at ℓ.

Finally, there is one subtle point: The ϕ-operations of a basic block can use a variable multiply concerning the same control flow edge like the ϕ-operations in figure 2(b) both use i_1 concerning edge e_1. So, arriving at B from e_1, i_1 must be written to i_3 and j_3. Thus, a copy from i_1 to either i_3 or j_3 must be inserted on edge e_1. This copy is inevitable since the value of i_1 must be present in *two* registers upon entering B. The decision which destination the copy has (in this example either i_3 or j_3) is deferred to coalescing since at this point in time it is not clear whether i_1 and i_3 or i_1 and j_3 can be assigned the same color.

Note that the same situation also occurs if an operand x of a ϕ is live-in at the ϕ's block. Then x and the ϕ's result interfere and cannot be given the same color. Thus a copy has to be inserted also.

4.3 Coalescing

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 program with ϕ-operations removed. In order to lower the number of transpositions needed for a ϕ-operation we investigate the problem of maximizing the number of fixed points of a ϕ.[5]

Concerning a coloring f, variable x is a fixed point of a ϕ-operation $y \leftarrow \phi(\dots, x, \dots)$ if x and y have been assigned the same register, i.e. $f(y) = f(x)$. Clearly, for fixed points no code has to be generated. Even more, if all ϕ-operands are fixed points, no code has to be generated for the ϕ at all.

Given a SSA-form program P, its interference graph $G = (V, E)$ and the set $\mathbf{\Phi}$ of all ϕ-operations in P. For a valid k-coloring $f : V \longrightarrow \{1, \dots, k\}$ of G, we define the costs of a ϕ-operation $p : y \leftarrow \phi(x_1, \dots, x_n)$ as follows:

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

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

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

[5] 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 5 (SSA-Maximize-Fixed-Points). *Given a SSA-form program P and its interference graph G. Find a coloring f of G for which $c_f(P)$ is minimal.*

Theorem 3. SSA-Maximize-Fixed-Points *is $\mathcal{NP}$-complete depending on the number of Φ-operations. For a proof see [14].*

A Heuristic Approach for SSA-Maximize-Fixed-Points In contrast to existing approaches we do *not* merge nodes in the interference graph but try to alter the coloring (as obtained with theorem 2) in order to assign operands of ϕ-operations and their results the same color. So, instead of changing the graph's structure, we search for a "better" k-coloring wrt. the cost function defined in equation 1. Thus, it will never happen that additional spill code is caused by assigning two nodes the same color, in contrast to the example in figure 1. Unlike other techniques, our method is not limited to the immediate neighborhood of the node pair to base its decision whether to coalesce or not.

The algorithm considers each ϕ-operation separately. The aim is to color as many operands of the ϕ equally to the ϕ's result. Therefore we consider an excerpt (later called conflict graph) from the interference graph containing the ϕ's result and its operands. Then we try to assign these nodes the same color. As this may lead to conflicts (as this color may already be in use by neighbors), we try to resolve these conflicts by recursively adjusting the conflicting nodes' colors. If we cannot resolve the conflicts for a node, we mark this node as incompatible.

For each ϕ-operation, we build an *optimization unit (OU)* $\omega = (y, x_1, \ldots, x_m)$ consisting of the ϕ's result y and the arguments $x_1, \ldots, x_m$ of the ϕ which do not interfere with y. An argument interfering with y can trivially never be assigned y's color. 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 all already processed OU. The processing of every $\omega = (y, x_1, \ldots, x_m)$ consists of three phases:

Init For each allowed color $\mathbf{c}$ for y, we insert an entry $E_{\mathbf{c}} = (\mathbf{c}, C_{\mathbf{c}}, S_{\mathbf{c}})$ into a priority queue. An entry consists of:

- a color $\mathbf{c}$.
- a conflict graph $C_{\mathbf{c}}$. Initially, $C_{\mathbf{c}}$ equals to the subgraph of the interference graph induced by $y, x_1, \ldots, x_m$.
- a maximum weighted stable set $S_{\mathbf{c}}$ of $C_{\mathbf{c}}$.[6] $S_{\mathbf{c}}$ represents all nodes in the conflict graph which shall be assigned the color $\mathbf{c}$. Each x_i in the OU is assigned the weight w_{yx_i} as defined in the cost function in equation 1. The weight of y is arbitrary, because y is contained in every maximum stable set by construction. This property is preserved throughout the optimization process.

The gain of $E_{\mathbf{c}}$ is the sum of the weights of the nodes contained in $S_{\mathbf{c}}$. The priority queue is ordered decreasingly by the gain of the entries. Thus, the

[6] A weighted stable set is a set of nodes equipped with weights for which no node is connected to the other.

first entry in the queue represents a coloring which provides the largest gain (or causes the fewest costs).

Test 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 in $S_{\mathbf{c}}$ 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 $u \in \{y, x_1, \ldots, x_m\}$ to $\mathbf{c}$. If a neighbor n of u is also colored with $\mathbf{c}$, we annotate n with the former color of u. This may provoke further conflicts which are then resolved recursively. Swapping the color of a node v originally initiated by changing the color of u to $\mathbf{c}$ ends in one of the three cases:

1. Changing v's color does not generate new conflicts.
2. 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 $C_{\mathbf{c}}$. Thus, u is excluded from every possible stable set of $C_{\mathbf{c}}$. Then, $S_{\mathbf{c}}$ is recomputed and the entry is reinserted into the queue.
3. 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 y to be always contained in each $S_{\mathbf{c}}$, if $v = y$, we add the edge uu, otherwise the edge uv to $C_{\mathbf{c}}$. Afterwards, $S_{\mathbf{c}}$ is recomputed and the entry is reinserted 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 $y, x_1, \ldots, x_m$ are marked as pinning candidates, testing ends for this OU.

Apply If the testing phase produced at least two pinning candidates (some x_i and y could be colored with the same color), the pinning candidates become *pinned* and all color changes annotated by the testing 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.

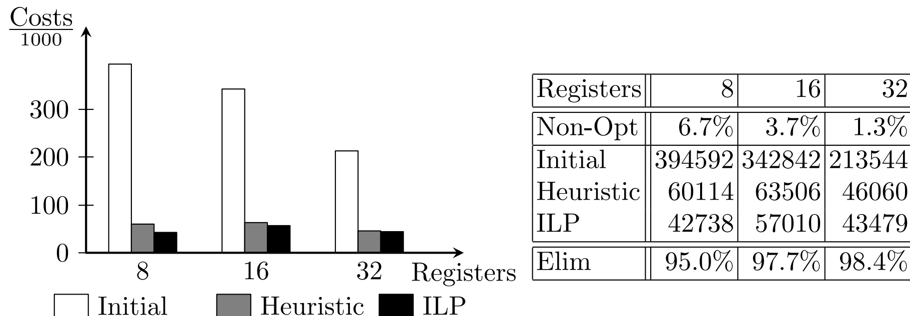
5 Measurements

We implemented our coalescing heuristic into our research compiler system Firm [15] and ran the complete C/C++-subset of the SPEC2000 benchmark suite through it. The architecture compiled for is a virtual RISC machine, to determine the effect of our approach on different register file sizes. Therefore, we did not measure the execution times of the compiled programs but investigated the quality of the heuristic's solutions in terms of costs of the target function as

defined in equation 1 in section 4.3. The weights w_{ij} are determined by the loop nesting depth to the power of two.

To assess the quality of the heuristic, we implemented an ILP formulation of SSA-MAXIMIZE-FIXED-POINTS (for details on the implementation, see [14]). Since ILP solving occasionally takes very long, the ILP solver was stopped after one minute of computation and thus did sometimes *not* produce an optimal solution.[7] As this happened in only 7% of all cases, we consider the solutions of the ILP solver as the best ones we could get and call the remaining costs after applying the ILP solver *unoptimizable.*

The measurements were conducted as follows: We compiled all C/C++-functions in the SPEC2000 benchmark suite for 8, 16 and 32 registers. We measured the costs incurred by the ϕ-functions at three stages in the compiler: After coloring with no coalescing done, after performing the heuristic and after applying the ILP solver. The solution of the heuristic was fed into the ILP solver as a start solution.[8] The row Non-Opt gives the percentage of functions for which the ILP solutions were not proven optimal. The results of the three measurements are reflected by the rows Initial, Heuristic and ILP in the table below. The Elim row shows the quotient (Initial − Heuristic)/(Initial − ILP) representing the fraction of optimizable costs the heuristic has eliminated. One can see that the heuristic eliminates always more than 95.0% of all optimizable costs.



Registers	8	16	32
Non-Opt	6.7%	3.7%	1.3%
Initial	394592	342842	213544
Heuristic	60114	63506	46060
ILP	42738	57010	43479
Elim	95.0%	97.7%	98.4%

6 Conclusions and Further Work

SSA-form programs allow for a new architecture of register allocators. Due to the chordality of their interference graphs, spilling and coalescing can be completely decoupled, thus avoiding the iterative approach in common graph coloring register allocators.

Based on the direct correspondence between the variables live at a label in the program and the cliques in its interference graph, we showed how an already

[7] To be precise, the solver could not prove the optimality of its best known feasible solution within time.

[8] Thus the solution of the ILP solver is always feasible and as good as or better than the heuristic's one.

existing, heuristic method for spilling in basic blocks can be extended to work on a whole procedure. Furthermore, we showed that an optimal coloring of the interference graph $G = (V, E)$ can be obtained in $O(\chi(G) \cdot |V|)$.

We investigated the $\mathcal{NP}$-complete problem of copy coalescing, presented a heuristic method for its solution and compared its quality to (near-) optimal solutions computed by an integer linear program. As our measurements show, the proposed coalescing heuristic eliminated more than 95% of all optimizable costs for 8, 16 and 32 registers.

Finally, we showed how a k-register allocation of a SSA program can be immediately turned into a k-register allocation of a non-SSA program. Thus, *optimizing* SSA-destruction is no longer necessary since it is handled by the coalescing phase.

As using SSA-form for register allocation demands a complete new backend architecture, we only had a prototype implementation running at the point in time this paper was written. The implementation of conventional register allocators is also work in progress and has not been completed.

As anticipated in section 4.3 optimal copy minimization is *not* achieved by maximizing the fixed points of a ϕ-operation but by minimizing the number of transpositions of the register permutation the ϕ stands for. Future work could investigate this problem.

7 Related Work

The first coalescing technique concerning graph coloring register allocation, called *aggressive coalescing,* was given by Chaitin et al. [1]. It recklessly coalesced all copies if the source and target did not interfere. It thus often introduced additional spill code by degrading the graph's colorability. Since then, a lot of work has been done on developing coalescing techniques which do not degrade the colorability of the graph, basically by stating criteria under which coalescing two nodes never will introduce a spill. Briggs et al. [2] introduced *conservative coalescing* which refuses to merge two nodes if the merged node will have more or equal than k neighbors. George and Appel [16] developed *iterated coalescing* which is able to remove more copies than conservative coalescing by interleaving it with the simplification phase of the register allocator. Park and Moon [17] present *optimistic coalescing* which adapts aggressive coalescing and integrate it into the Briggs allocator, allowing to undo coalescing if the coalesced node is selected to be spilled.

In his inspiring paper [18] Andersson tested a huge amount of interference graphs from the SML/NJ compiler published by Appel and George for the so called 1-perfectness property and found for all graphs he investigated, $\omega(G)$ was equal to $\chi(G)$. Following Andersson's work, Pereira and Palsberg [19] tested the interference graphs of the Java standard library compiled with the JoeQ compiler for chordality and found that 95% of them were chordal. They propose a register allocator without iteration for non-SSA programs and give heuristics both for spilling and coalescing. Since their approach works with non-SSA programs and

even non-chordal interference graphs, they cannot utilize the theoretic properties presented in section 2.

A more technical proof (without using perfect elimination orders) of the chordality of SSA interference graphs by one of the authors can be found in [**?**]. Brisk [20] gives a proof for the perfectness of the interference graphs of SSA-form programs. Bouchez [5] extensively studies the complexity of the spilling problem for SSA-form programs. He proves the problem of reducing the number of live variables for each label to k while minimizing the number of reloads to be $\mathcal{NP}$-complete wrt. to the chromatic number of the interference graph of the SSA program.

8 Acknowledgements

We want to thank our colleagues Michael Beck, Rubino Geiß, Götz Lindenmaier for many fruitful discussions. Fernando Pereira, Jens Palsberg, Philip Brisk and Daniel Berlin provided many helpful insights. Also, this paper greatly benefitted from discussions arising from a seminar at the Computer Laboratory of Cambridge University.

References

1. Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.W.: Register allocation via graph coloring. Journal of Computer Languages **6** (1981) 45–57
2. Briggs, P., Cooper, K.D., Torczon, L.: Improvements to graph coloring register allocation. ACM Trans. Program. Lang. Syst. **16** (1994) 428–455
3. Golumbic, M.C.: Algorithmic Graph Theory And Perfect Graphs. Academic Press (1980)
4. Budimlić, Z., Cooper, K.D., Harvey, T.J., Kennedy, K., Oberg, T.S., Reeves, S.W.: Fast copy coalescing and live-range identification. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, ACM Press (2002) 25–32
5. Bouchez, F.: Allocation de registres et vidage en mémoire. Master's thesis, ÉNS Lyon (2005)
6. Diestel, R.: Graph Theory. 3 edn. Volume 173 of Graduate Texts in Mathematics. Springer (2005)
7. Bergner, P., Dahl, P., Engebretsen, D., O'Keefe, M.: Spill code minimization via interference region spilling. In: PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation, New York, NY, USA, ACM Press (1997) 287–295
8. Chow, F.C., Hennessy, J.L.: The priority-based coloring approach to register allocation. ACM Trans. Program. Lang. Syst. **12** (1990) 501–536
9. Hsu, W.C., Fisher, C.N., Goodman, J.R.: On the Minimization of Loads/Stores in Local Register Allocation. IEEE Trans. Softw. Eng. **15** (1989) 1252–1260
10. Guo, J., Garzaran, M.J., Padua, D.: 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)

11. Belady, L.: A Study of Replacement of Algorithms for a Virtual Storage Computer. IBM Systems Journal **5** (1966) 78–101
12. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadek, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems **13** (1991) 451–490
13. Briggs, P., D.Cooper, K., Harvey, T.J., Simpson, L.T.: Practical Improvements to the Construction and Destruction of Static Single Assignment Form. Software: Practice and Experience **28** (1998) 859–881
14. Hack, S., Grund, D., Goos, G.: Towards Register Allocation for Programs in SSA-form. Technical report (2005)
15. Lindenmaier, G., Beck, M., Boesler, B., Geiš, R.: Firm, an intermediate language for compiler research. Technical Report 2005-8 (2005)
16. George, L., Appel, A.W.: Iterated register coalescing. ACM Trans. Program. Lang. Syst. **18** (1996) 300–324
17. Park, J., Moon, S.M.: Optimistic register coalescing. ACM Trans. Program. Lang. Syst. **26** (2004) 735–765
18. Andersson, C.: Register Allocation By Optimal Graph Coloring. In Hedin, G., ed.: CC 2003. Volume 2622 of LNCS., Heidelberg, Springer-Verlag (2003) 33–45
19. Pereira, F.M.Q., Palsberg, J.: Register allocation via coloring of chordal graphs. In: Proceedings of APLAS'05. (2005)
20. Brisk, P., Dabiri, F., Macbeth, J., Sarrafzadeh, M.: Polynomial time graph coloring register allocation. In: In 14th International Workshop on Logic and Synthesis, ACM Press (2005)