SSA-based Register Allocation with PBQP

Sebastian Buchwald, Andreas Zwinkau, and Thomas Bersch

Karlsruhe Institute of Technology (KIT)
{buchwald,zwinkau}@kit.edu thomas.bersch@student.kit.edu

Abstract. Recent research shows that maintaining SSA form allows to split register allocation into separate phases: spilling, register assignment and copy coalescing. After spilling, register assignment can be done in polynomial time, but copy coalescing is NP-complete. In this paper we present an assignment approach with integrated copy coalescing, which maps the problem to the Partitioned Boolean Quadratic Problem (PBQP). Compared to the state-of-the-art recoloring approach, this reduces the relative number of swap and copy instructions for the SPEC CINT2000 benchmark to 99.6% and 95.2%, respectively, while taking 19% less time for assignment and coalescing.

Keywords: register allocation, copy coalescing, PBQP

1 Introduction

Register allocation is an essential phase in any compiler generating native code. The goal is to map the program variables to the registers of the target architecture. Since there are only a limited number of registers, some variables may have to be spilled to memory and reloaded again when needed. Common register allocation algorithms like Graph Coloring [3,6] spill on demand during the allocation. This can result in an increased number of spills and reloads [11]. In contrast, register allocation based on static single assignment (SSA) form allows to completely decouple the spilling phase. This is due to the live-range splits induced by the ϕ-functions, which render the interference graph chordal and thus ensure that the interference graph is k-colorable, where k is the maximum register pressure.

The live-range splits may result in *shuffle code* that permutes the values (as variables are usually called in SSA form) on register level with copy or swap instructions. To model this circumstance, *affinities* indicate which values should be *coalesced*, i.e. assigned to the same register. If an affinity is fulfilled, inserting shuffle code for the corresponding values is not necessary anymore. Fulfilling such affinities is the challenge of copy coalescing and a central problem in SSA-based register allocation.

In this work we aim for a register assignment algorithm that is aware of affinities. In order to model affinities, we employ the Partitioned Boolean Quadratic Problem (PBQP), which is a generalization of the graph coloring problem.

In the following, we

- employ the chordal nature of the interference graph of a program in SSA form to obtain a linear PBQP solving algorithm, which does not spill registers, but uses a decoupled phase instead.
- integrate copy coalescing into the PBQP modelling, which makes a separate phase afterwards unnecessary.
- develop a new PBQP reduction to improve the solution quality by merging two nodes, if coloring one node implies a coloring of the other one.
- introduce an advanced technique to handle a wide class of register constraints during register assignment, which enlarges the solution space for the PBQP.
- show the competitiveness of our approach by evaluating our implementation for quality and speed. Additionally, some insight into our adaptations is gained by interpreting our measurements.

In Section 2 we describe related register allocation approaches using either SSA form or a PBQP-based algorithm, before we combine both ideas in Section 3, which shows the PBQP algorithm and our adaptations in detail. Section 4 presents our technique to handle register constraints. Afterwards, an evaluation of our implementation is given in Section 5. Finally, Section 6 describes future work and Section 7 our conclusions.

2 Related Work

2.1 Register allocation on SSA form

Most SSA-based compilers destruct SSA form in their intermediate representation after the optimization phase and before the code generation. However, maintaining SSA form provides an advantage for register allocation: Due to the live-range splits that are induced by the ϕ-functions, the interference graph of programs in SSA form is *chordal* [4,1,13], which means every induced subgraph that is a cycle, has length three. For chordal graphs the chromatic number is determined by the size of the largest clique. This means that the interference graph is k-colorable, if the spiller has reduced the register pressure to at most k at each program point. Thus, spilling can be decoupled from assignment [13], which means that the process is not iterated as with Graph Coloring.

To color the interference graph we employ the fact that there is a *perfect elimination order* (PEO) for each chordal graph [7]. A PEO defines an ordering $<$ of nodes of a graph G, such that each successively removed node is *simplicial*, which means it forms a clique with all remaining neighbors in G. After spilling, assigning the registers in reverse PEO ensures that each node is simplicial and thus has a free register available. Following Hack, we obtain a PEO by a post-order walk on the dominance tree [11].

In addition to ϕ-functions, live-range splits may originate from constrained instructions. For instance, if a value is located in register $R1$, but the constrained instruction needs this value in register $R2$, we can split the live-range of this

value. More generally, we split all live-ranges immediately before the constrained instruction to allow for unconstrained copying of values into the required registers and employ copy coalescing to remove the overhead afterwards. With this in mind, we add *affinity edges* to the interference graph, which represent that the incident nodes should be assigned to the same register. Each affinity has assigned costs, which can be weighted by the execution frequency of the potential copy instruction. The goal of *copy coalescing* is to find a coloring that minimizes the costs of unfulfilled affinities. Bouchez et al. showed that copy coalescing for chordal graphs is NP-complete [1], so one has to consider the usual tradeoff between speed and quality.

Grund and Hack [10] use integer linear programming to optimally solve the copy coalescing problem for programs in SSA form. To keep the solution time bearable some optimizations are used, but due to its high complexity the approach is too slow for practical purposes.

In contrast, the recoloring approach from Hack and Goos [12] features a heuristic solution in reasonable time. It improves an existing register assignment by recoloring the interference graph. Therefore, an initial coloring can be performed quickly without respect to quality. To achieve an improvement, the algorithm tries to assign the same color to affinity-related nodes. If changing the color of a node leads to a conflict with interfering neighbors, the algorithm tries to solve this conflict by changing the color of the conflicting neighbors. This can cause conflicts with other neighbors and recursively lead to a series of color changes. These changes are all done temporarily and will only be accepted, if a valid recoloring throughout the graph is found.

Another heuristic approach is the preference-guided register assignment introduced by Braun et al. [2]. This approach works in two phases: in the first phase the register preferences of instructions are determined and in the second phase registers are assigned in consideration of these preferences. The preferences serve as implicit copy coalescing, such that no extra phase is needed. Furthermore, the approach does not construct an interference graph. Preference-guided register assignment is inferior to recoloring in terms of quality, but significantly faster.

2.2 PBQP-based register allocation

The idea to map register allocation to PBQP was first implemented by Scholz and Eckstein [16] and features a linear heuristic. Since their implementation does not employ SSA form, they need to integrate spilling into their algorithm. Hames and Scholz [14] refine the approach by presenting a new heuristic and a branch-and-bound approach for optimal solutions. The essential advantage of the PBQP approach is its flexibility, which makes it suited for irregular architectures.

3 PBQP

3.1 PBQP in general

The PBQP is a special case of a Quadratic Assignment Problem and essentially consists of multiple interdependent choices with associated costs. While the problem can be expressed formally [8], a graph-based approach is more intuitive. Each graph node has an associated choice vector, which assigns each alternative its cost in $\mathbb{R}$. For each node only one alternative can be selected. Interdependencies are modeled by directed edges with an associated matrix, which assigns a cost to each combination of alternatives in $\mathbb{R} \cup \{\infty\}$. Naturally, a node with n choices and one with m choices have cost vectors of dimension n and m, respectively. An edge between them must have a corresponding matrix of size $n \times m$. If we select the i-th alternative at the source node and the j-th alternative at the target node, we implicitly select the entry at the i-th row and j-th column of the edge matrix.

A *selection* assigns each node one of its alternatives. This also implies a matrix entry for each edge. The cost of a selection is the sum of all chosen vector and matrix entries. If this cost is finite, the selection is called a *solution*. The goal of the PBQP is to find a solution with minimal cost.

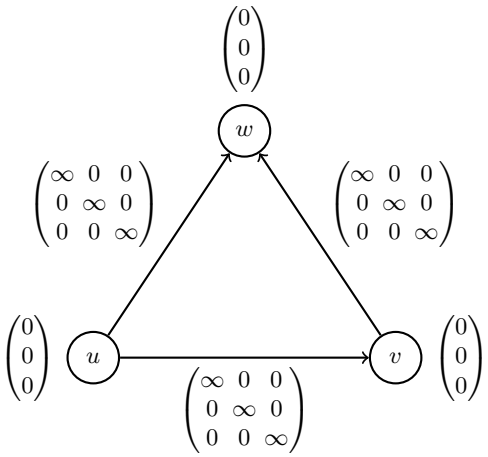


Fig. 1: PBQP instance for 3-coloring.

Figure 1 shows a PBQP instance for 3-coloring. Each node u, v, and w has three alternatives that represent its possible colors. Due to the edge matrices, each solution of the PBQP instance has to select different colors for adjacent nodes and thus represents a valid 3-coloring. The reduction from 3-coloring to PBQP renders PBQP NP-complete. Additionally, it shows that finding *any*

PBQP solution is NP-complete, since each solution is optimal. However, our algorithm can still solve all of our specific problems in linear time as we show in Section 3.4.

3.2 PBQP construction

As mentioned above, SSA-based register allocation allows to decouple spilling and register assignment, such that register assignment is essentially a graph coloring problem. In Figure 1 we show that PBQP can be considered as a generalization of graph coloring by employing color vectors and interference matrices:

Color vector The color vector contains one zero cost entry for each possible color.

$$\begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^{\mathrm{T}}$$

Interference matrix The matrix costs are set to ∞, if the corresponding colors are equal. Otherwise, the costs are set to zero.

$$\begin{pmatrix} \infty & 0 & 0 \\ 0 & \infty & 0 \\ 0 & 0 & \infty \end{pmatrix}$$

Register allocation can be considered as a graph coloring problem by identifying registers and colors. In order to integrate copy coalescing into register assignment, we add affinity edges for all potential copies, which should be prevented.

Affinity matrix The matrix costs are set to zero if the two corresponding registers are equal. Otherwise, the costs are set to a positive value that represents the cost of inserted shuffle code.

$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

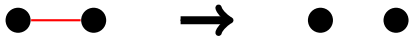
A PBQP instance is constructed like an interference graph by inspecting the live-ranges of all values. The values become nodes and get their corresponding color vectors assigned. For each pair of interfering nodes an interference matrix is assigned to the edge between them. Likewise, the affinity cost matrix is assigned to an edge between a pair of nodes, which should get the same register assigned. If a pair of nodes fulfills both conditions, then the sum of both cost matrices is assigned. While the copy cannot be avoided in this case, there may still be cost differences between the register combinations.

3.3 Solving PBQP instances

Solving the PBQP is done by iteratively reducing an instance to a smaller instance until all interdependencies vanish. Without edges every local optimum is

also globally optimal, so finding an optimal solution is trivial (unless there is none). By backpropagating the reductions, the selection of the smaller instance can be extended to a selection of the original PBQP instance. Originally, there are four reductions [5,9,16]:

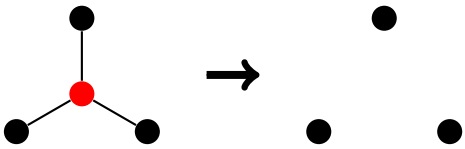
RE: Independent edges have a cost matrix that can be decomposed into two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$, i.e. each matrix entry C_{ij} has costs $u_i + v_j$. Such edges can be removed after adding $\boldsymbol{u}$ and $\boldsymbol{v}$ to the cost vector of the source and target node, respectively. If this would produce infinite vector costs, the corresponding alternative (including matrix rows/columns) is deleted.



R1: Nodes of degree one can be removed, after costs are accounted in the adjacent node.

R2: Nodes of degree two can be removed, after costs are accounted in the cost matrix of the edge between the two neighbors; if necessary, the edge is created first.

RN: Nodes of degree three or higher. For a node u of maximum degree we select a locally minimal alternative, which means we only consider u and its neighbors. After the alternative is selected, all other alternatives are deleted and the incident edges are independent, so they can be removed by using RE.



RE, R1, and R2 are *optimal* in the sense that they transform the PBQP instance to a smaller one with equal minimal costs, such that each solution of the small instance can be extended to a solution of the original instance with equal costs. For sparse graphs these reductions are very powerful, since they diminish the problem to its core. If the entire graph can be reduced by these reductions, then the resulting PBQP selection is optimal. If nodes of degree three or higher remain, the heuristic RN ensures linear time behavior.

Hames et al. [14] introduced a variant of RN that removes the corresponding node from the PBQP graph without selecting an alternative. Thus, the decision is delayed until the backpropagation phase. We will refer to this approach as *late decision*. The other approach is *early decision*, which colors a node during RN. In this paper we follow both approaches and we will show which one is more suitable for SSA-based register allocation.

3.4 Adapting the PBQP solver for SSA-based Register Allocation

In the context of SSA-based register allocation, coloring can be guaranteed to succeed, if done in reverse PEO with respect to the interference graph. This seems to imply that our PBQP solver must assign registers in reverse PEO. However, we show in the following that this restriction is only necessary for heuristic reductions. Therefore, choosing a node for heuristic decision must use the last node of the PEO for early decision and the first node of the PEO for late decision. This different selection of nodes is needed, because the backpropagation phase inverts the order of the reduced nodes.

Early application of optimal reductions There is a conflict between the necessity to assign registers in reverse PEO and the PBQP strategy to favor RE, R1, and R2 until only RN is left to apply. Fortunately, we can show that the application of this reductions preserves the PEO property with Theorem 1 below.

Lemma 1. *Let P be a PEO of a graph $G = (V, E)$ and $H = (V', E')$ an induced subgraph of G, then $P|_{V'}$ is a PEO of H.*

Proof. For any node $v \in H$ let $V_v = \{u \in N_G(v) \mid u > v\}$ be the neighbor nodes in G behind in P and respectively $V'_v = \{u \in N_H(v) \mid u > v\}$. By definition $V'_v \subseteq V_v$. V_v is a clique in G, therefore V'_v is a clique in H and v is simplicial, when eliminated according to $P|_{V'}$. □

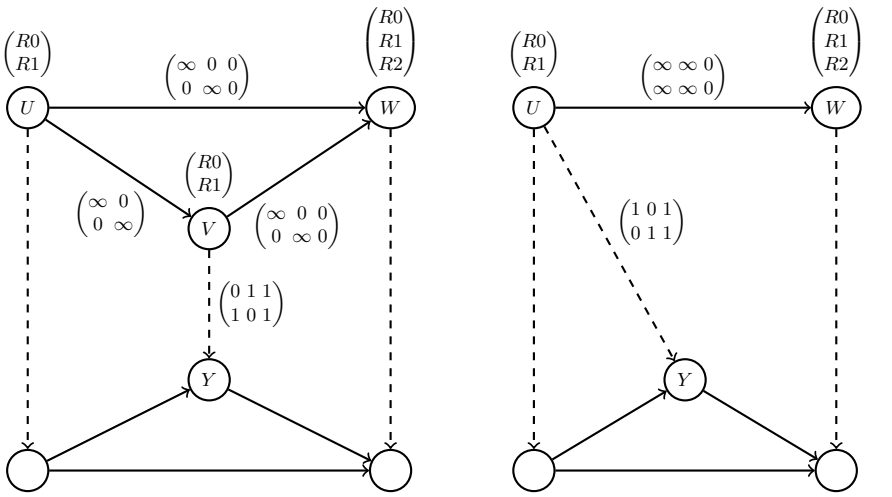
From Lemma 1 we know, that R1 preserves the PEO, since the resulting graph is always an induced subgraph. However, R2 may insert a new edge into the PBQP graph.

Lemma 2. *Let P be a PEO of a graph $G = (V, E)$ and $v \in V$ a vertex of degree two with neighbors $u, w \in V$. Further, let H=(V',E') be the subgraph induced by $V \setminus \{v\}$. Then $P|_{V'}$ is a PEO of $H' = (V', E' \cup \{\{u, w\}\})$.*

Proof. If $\{u, w\} \in E$ this follows directly from Lemma 1, since no new edge is introduced. In the other case, we assume without loss of generality $u < w$. Since $\{u, w\} \notin E$, v is not simplicial and we get $u < v$. Therefore, the only neighbor node of u in H behind in P must be v. Within H' the node w is the only neighbor of u behind in the PEO, hence u is simplicial. For the remaining nodes v and w the lemma follows directly from Lemma 1. □

With Lemma 2 the edge inserted by R2 is proven harmless, so we can derive the necessary theorem now. Remember that the PEO must be derived from the interference graph, while our PBQP graph may also include affinity edges.

Theorem 1. *Let $G = (V, E)$ be a PBQP graph, $i(E)$ the interference edges in E, i.e. edges that contain at least one infinite cost entry, P a PEO of $(V, i(E))$ and $H = (V', E')$ the PBQP graph G after exhaustive application of RE, R1 and R2. Further, let E_i be the interference edges that are reduced by RE. Then, $P|_{V'}$ is a PEO of $(V', i(E') \cup E_i)$.*



(a) PBQP instance that cannot be reduced further by RE, R1 or R2.

(b) PBQP instance after merging V into U.

Fig. 2: Example of an RM application.

Proof. If at least one affinity edge is involved in a reduction, there is no new interference edge and we can apply Lemma 1. If we consider only interference edges, Lemma 1 handles R1 and Lemma 2 handles R2. Applying RE for an interference edge moves the interference constraint into incident nodes. Thus, we have to consider such edges E_i for the PEO. □

Merging PBQP nodes The PBQP instances constructed for register assignment contain many interference cliques, which cannot be reduced by optimal reductions. This implies that the quality of the PBQP solution highly depends on the RN decisions. In this section we present *RM*, a new PBQP reduction that is designed to improve the quality of such decisions.

Figure 2a shows a PBQP instance that cannot be further reduced by application of RE, R1 or R2. Hence, we have to apply a heuristic reduction. We assume that the PEO selects U to be the heuristically reduced node. The new reduction is based on the following observation: If we select an alternative at U, there is only one alternative at V that yields finite costs. Thus, a selection at U implicitly selects an alternative at V. However, the affinities of V are not considered during the reduction of U. The idea of the new reduction is to merge the neighbor V into U. After the merge, U is also aware of V's affinities which may improve the heuristic decision.

To perform the merge, we apply the RM procedure of Algorithm 1 with arguments $v = V$ and $u = U$. In line 2 the algorithm chooses $w = Y$ as adjacent

Algorithm 1 RM merges the node v into the node u. The notation is adopted from [16].

Require: a selection at u implies a specific selection at v.

```
 1: procedure RM(v, u)
 2:     for all w ∈ adj(v) do
 3:         if w ≠ u then
 4:             for i ← 1 to |c_u| do
 5:                 c ← 0
 6:                 if c_u(i) ≠ ∞ then
 7:                     i_v ← i_min(C_uv(i, :) + c_v)
 8:                     c ← C_vw(i_v, :)
 9:                 Δ(i, :) ← c
10:             C_uw ← C_uw + Δ
11:             remove edge (v, w)
12:     ReduceI(v)
```

node. We now want to replace the affinity edge (v, w) by an edge (u, w). In lines 4–9 we create a new matrix Δ for the edge (u, w). To construct this matrix we use the fact that the selection of an alternative i at u also selects an alternative i_v at v. Thus, the i-th row of Δ is the i_v-th row of C_{uw}. For our example this means that we have to swap the rows for $R0$ and $R1$ in order to obtain Δ from C_{vw}. Since (u, w) does not exist, we create the edge with the matrix Δ. Afterwards we delete the old edge (v, w).

In the next iteration, the algorithm chooses $w = W$ as adjacent node. Similar to the previous iteration, we compute the matrix Δ. Since the edge (u, w) exists, we have to add Δ to the matrix C_{uw}. After the deletion of (v, w), the node v has degree one and can be reduced by employing R1. Figure 2b shows the resulting PBQP instance. Due to the merge the edge (U, W) is independent. After removing the edge the PBQP instance can be solved by applying R1 and R2, leading to an optimal solution.

Although RM is an optimal reduction, we only apply it immediately before a heuristic reduction at a node U. If there is a neighbor V of U that can be merged into U we apply RM for these two nodes. This process iterates until no such neighbor is left. In some cases—like our example—the RM allows further optimal reductions that supersede a heuristic reduction at U. If U still needs a heuristic reduction, the neighbors of the merged nodes are also considered by the heuristic and thus can improve the heuristic decision.

The reason for applying RM only immediately before a heuristic decision at a node u is that in this case each edge is reassigned only once, due to the fact that the node u (and all incident edges) will be deleted after the merge. Thus, each edge is considered at most twice: once for reassignment, and once during the reduction of u. As a result, the overall RM time complexity is in $\mathcal{O}(mk^2)$ where $m = |E|$ is the number of edges in the PBQP graph and k is the maximum number of alternatives at a node. The same argument can be used to show that the overall RN time complexity is in $\mathcal{O}(mk^2)$. Together with the existing

optimal reductions this leads to an overall time complexity in $\mathcal{O}(nk^3 + mk^2)$ where $n = |V|$ denotes the number of nodes in the PBQP graph [5,16]. Since k is the constant number of registers in our case, the solving algorithm has linear time complexity.

Similar to the other PBQP reductions, RM modifies the PBQP graph and thus we have to ensure that our PEO is still valid for the resulting graph.

Theorem 2. *Let $G = (V, E)$ be a PBQP graph, $i(E)$ the interference edges in E, P a PEO of $(V, i(E))$ and $H = (V', E')$ the PBQP graph G after exhaustive application of RM and the reduction of the greatest node u with respect to P. Further, let E_i be the interference edges that are reduced by RE. Then, $P|_{V'}$ is a PEO of $(V', i(E') \cup E_i)$.*

Proof. If u is reduced by RN or R1 this follows from Lemma 1. In the R2 case, let v be the last node whose merge changed u's degree. The theorem follows from applying Lemma 1 for u and all nodes that are merged before v and Lemma 2 for v and all nodes that are merged after v. Independent edges will be handled as in Theorem 1. □

4 Register Constraints

For SSA-based register allocation, naïve handling of register constraints can inhibit a coloring of the interference graph. For example, Figure 3a shows an instruction *instr* with three operands and two results. The operands I_1 and I_2 are live before the instruction, but are not used afterwards, so they *die* at the instruction. In constrast, operand I_3 is used afterwards (as indicated by the dashed live-range) and interferes with both results. We assume that a, b and c are the only available registers. The values are constrained to the annotated registers, for instance, the operand I_1 is constrained to registers $\{a, b\}$. However, a previous instruction may impose the constraint $\{c\}$ on I_1. Since both constraints are contradictory, there is no coloring of the interference graph. To prevent such situations, Hack splits all live-ranges before the constrained instruction [11]. For our example, this allows to fulfill the constraints by inserting a copy from register c to register a or b.

The next problem is that a PEO ensures a k-coloring only for unconstrained nodes. For example, we can derive the coloring order I_1, I_2, I_3, O_1, O_2 from a PEO of Figure 3a, but assigning c to I_2 inhibits a valid register assignment. To tackle this issue we employ the fact that if we obtain the PEO by a post-order walk of the dominance tree, the values live before and immediately after the constrained instruction are colored first. Thus, if we provide a coloring for these nodes, we can use our PEO to color the remaining nodes. Hack showed [11] how such a coloring can be found in the case of *simple constraints*, i.e. if each value is either constrained to one register or unconstrained. In case of a value with a non-simple constraint, the interference cliques before and after the statement are colored separately and values with non-simple constraints are pinned to the

chosen color. This may increase the register demand, but ensures a valid register allocation.

Simple constraints can easily be integrated into the PBQP solving algorithm, since we only have to ensure that operands which are live after the constrained instruction are colored first. However, pinning the values to a single register is very restrictive. In the following, we assume that the spiller enables a coloring by inserting possibly necessary copies of operands and present an algorithm that can deal with *hierarchic constraints*.

Definition 1 (hierarchic constraints). *Let $\mathcal{C}$ be a set of register constraints. $\mathcal{C}$ is hierarchic if for all constraints $C_1 \in \mathcal{C}$ and $C_2 \in \mathcal{C}$ holds:*

$$C_1 \cap C_2 \neq \emptyset \Rightarrow C_1 \subseteq C_2 \vee C_2 \subseteq C_1.$$

This definition excludes "partially overlapping" constraints, like $C_1 = \{a, b\}$ and $C_2 = \{b, c\}$. As a result, the constraints form a tree with respect to strict inclusion, which we call *constraint hierarchy*. For instance, the constraint hierarchy for the general purpose registers of the IA-32 architecture consists of $C_{all} = \{A, B, C, D, SI, DI\}$, a subset $C_s = \{A, B, C, D\}$ for instructions on 8- or 16-bit subregisters, and all constraints that consist of a single register.

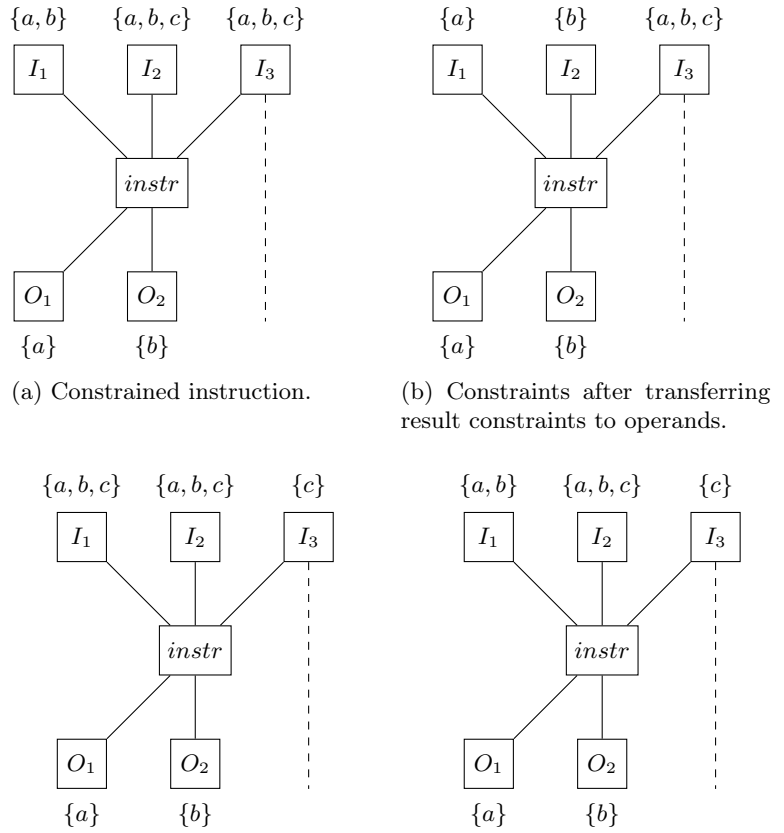
For hierarchic constraints we obtain a valid register assignment of an interference clique by successively coloring a most constrained node. However, for a constrained instruction we also have to ensure that after coloring the values, which are live before an instruction *instr*, we still can color the values live after *instr*. In Figure 3a O_1 and O_2 are constrained to a and b, respectively, and thus c must be assigned to I_3. Unfortunately, c may also be chosen for I_2 according to its constraints, if it is colored before I_3. To avoid such situations, we want to modify the constraints in a way that forces the first two operands to use the same registers as the results. This is done in three steps:

1. Add unconstrained pseudo operands/results until the register pressure equals the number of available registers.
2. Match results and dying operands to assign result constraints to the corresponding operand, if they are more restrictive.
3. Try to relax the introduced constraints of the previous step, to enable more affinities to be fulfilled.

The first step ensures that the number of dying operands and the number of results are equal, which is required by the second step. For our example in Figure 3a we have nothing to do, since the register pressure before and after *instr* is already equal to the number of available registers.

4.1 Restricting operands

We employ Algorithm 2 for the second step. The algorithm has two parameters: A multiset of input constraints and a multiset of output constraints. It iteratively pairs an input constraint with an output constraint. For this pairing we select



(a) Constrained instruction.

(b) Constraints after transferring result constraints to operands.

(c) Constraints after relaxation of operand constraints.

(d) Final constraints for PBQP construction.

Fig. 3: Handling of constrained instructions.

a minimal constraint (with respect to inclusion) C_{min}. Then we try to find a minimal partner $C_{partner}$, which is a constraint of the other parameter such that $C_{min} \subseteq C_{partner}$. If C_{min} is an output constraint we transfer the constraint to the partner. It is not necessary to restrict output constraints, since the inputs are colored first and restrictions propagate from there.

For our example in Figure 3a the algorithm input is $\{I_1, I_2\}$ and $\{O_1, O_2\}$. The constraints of O_1 and O_2 are both minimal. We assume that the function $getMinimalElement$ chooses O_1 and thus $C_{min} = \{a\}$. Since O_1 is a result, the corresponding partner must be an operand. We select the only minimal partner I_1 which leads to $C_{partner} = \{a, b\}$. We now have our first match (I_1, O_1) and remove both values from the value sets. Since the result constraint is more restrictive, we assign this constraint to the operand I_1. In the next iteration we

Algorithm 2 Restricting constraints of input operands.

```
1: procedure RESTRICTINPUTS(Ins, Outs)
2:     C ← Ins ∪ Outs
3:     while C ≠ ∅ do
4:         C_min ← getMinimalElement(C)
5:         C_partner ← getMinimalPartner(C, C_min)
6:         C ← C \ {C_min, C_partner}
7:         if C_partner from Ins then
8:             assign C_min to partner value
```

match I_2 and O_2 and restrict I_2 to $\{b\}$. The resulting constraints are shown in Figure 3b. Due to the introduced restrictions, the dying operands have to use the same registers as the results.

In the following, we prove that $getMinimalPartner$ always finds a minimal partner. Furthermore, we show that Algorithm 2 cannot restrict the operands constraints in a way that renders a coloring of the values, which are live before the instruction, impossible.

Theorem 3. *Let $G = (V = \mathcal{I} \cup \mathcal{O}, E)$ a bipartite graph with $\mathcal{I} = \{I_1, \ldots, I_n\}$ and $\mathcal{O} = \{O_1, \ldots, O_n\}$. Further, let $\mathcal{R} = \{R_1, \ldots, R_n\}$ be a set of colors (registers) and $constr : V \to \mathcal{P}(\mathcal{R})$ a function that assigns each node its feasible colors. Moreover, let $c : v \mapsto R_v \in constr(v)$ be a coloring of V that assigns each color to exactly one element of $\mathcal{I}$ and one element of $\mathcal{O}$. Let $M \subseteq E$ be a perfect bipartite matching of G such that*

$$\{u, v\} \in M \Rightarrow c(u) = c(v).$$

Then, Algorithm 2 finds a perfect bipartite matching M' such that there is a coloring $c' : v \mapsto R'_v \in constr(v)$ that assigns each color to exactly one element of $\mathcal{I}$ and one element of $\mathcal{O}$ with

$$\{u, v\} \in M' \Rightarrow c'(u) = c'(v).$$

Proof. We prove the theorem by induction on n. For $n = 1$ there is only one perfect bipartite matching and since $c(I_1) = c(O_1) \in (constr(I_1) \cap constr(O_1))$ we have $constr(I_1) \subseteq constr(O_1)$ or $constr(O_1) \subseteq constr(I_1)$. Thus, Algorithm 2 finds the perfect bipartite matching which can be colored by $c' = c$.

For $n > 1$, without loss of generality, we can rename the nodes such that $\forall i : c(I_i) = c(O_i)$ and Algorithm 2 selects O_1 as node with minimal constraints. If the algorithm selects I_1 to be the minimal partner, we can remove I_1 and O_1 from the graph, the color $c(I_1)$ from the set of colors $\mathcal{R}$ and apply the induction assumption.

In case the algorithm does not select I_1 as minimal partner let I_p be the minimal partner. Our goal is to show that there is a coloring for

$$M'' = (M \setminus \{\{I_1, O_1\}, \{I_p, O_p\}\}) \cup \{\{I_1, O_p\}, \{I_p, O_1\}\}$$

and then apply the induction assumption. To obtain such a coloring we consider the corresponding constraints. Since O_1 has minimal constraints and $c(I_1) = c(O_1) \in (constr(I_1) \cap constr(O_1))$, we get $constr(O_1) \subseteq constr(I_1)$. Furthermore, we know that I_p is the minimal partner of O_1 which means $constr(O_1) \subseteq constr(I_p)$ by definition. Thus, we get $\emptyset \neq constr(O_1) \subseteq (constr(I_1) \cap constr(I_p))$ and since I_p is the minimal partner of O_1, we get $constr(I_p) \subseteq constr(I_1)$. Using these relations, we obtain

$$c(O_1) \in constr(O_1) \subseteq constr(I_p)$$

$$c(O_p) = c(I_p) \in constr(I_p) \subseteq constr(I_1).$$

Thus, $c'' = c[I_1 \mapsto c(O_p), I_p \mapsto c(O_1)]$ is a coloring for M''. We now remove $\{I_p, O_1\}$ from the graph and $c''(O_1)$ from the set of colors $\mathcal{R}$ and apply the induction assumption, resulting in a matching M''' and a coloring c'''. Since c''' does not use the color $c''(O_1)$, we can extend the matching M''' to

$$M' = M''' \cup \{I_p, O_1\}$$

and the corresponding coloring c''' to

$$c'(v) = \begin{cases} c''(O_1) & , v \in \{I_p, O_1\} \\ c'''(v) & , \text{otherwise} \end{cases}$$

so $\{u, v\} \in M' \Rightarrow c'(u) = c'(v)$ holds. □

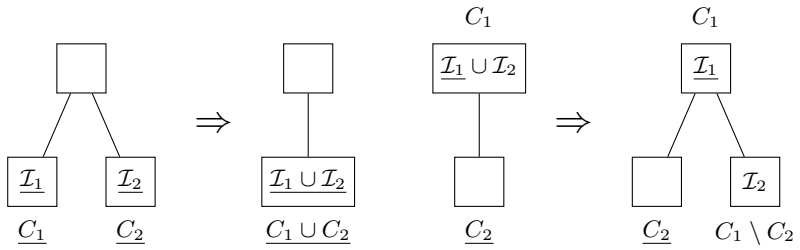
4.2 Relaxing constraints

The restriction of the operands ensures a feasible coloring. However, some of the operands may now be more restricted than necessary, so the third step relaxes their constraints again. For instance, in Figure 3b the operands I_1 and I_2 are pinned to register a and b, respectively, but assigning register b to I_1 and register a to I_2 is also feasible. To permit this additional solution, the constraints can be relaxed to $\{a, b\}$ for both operands. In the following, we provide some rules that modify the constraint hierarchy of the input operands in order to relax the previously restricted constraints. We introduce two predicates to determine whether a rule is applicable or not.

Dying A node is dying if the live-range of the operand ends at the instruction. Its assigned register is available for result values.

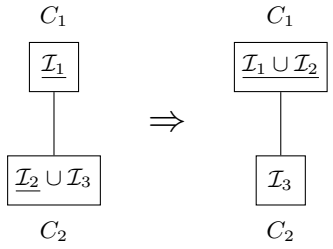
Saturated A constraint C is saturated, if it contains as many registers $|C|$ as there are nodes, which must get one of those registers assigned $|\{I \in \mathcal{I} \mid C_I \subseteq C\}|$. This means, every register in C will be assigned in the end.

Figure 4 shows the transformation rules for constraint hierarchies. The rules are applied greedily from left to right. A constraint C_i is underlined if it is saturated. Each constraint has a set of dying nodes $\underline{\mathcal{I}_i}$ and a set of non-dying nodes $\mathcal{I}_j$.

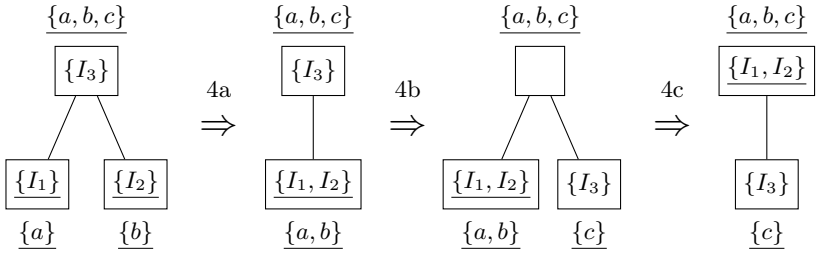


(a) Merging two saturated constraints consisting of dying nodes.

(b) Restricting non-dying nodes due to a saturated constraint.



(c) Moving dying nodes along the constraint hierarchy.



(d) Application of the transformation rules to relax the operand constraints shown in Figure 3b.

Fig. 4: Rules to relax constraints and a usage example.

The rule shown in Figure 4a combines two saturated constraints that contain only dying nodes. Applying this rule to the constraint hierarchy of our example in Figure 3b lifts the constraints of I_1 and I_2 to $\{a, b\}$. Since both values die, it is not important which one is assigned to register a and which one to register b.

We now apply the rule of Figure 4b. This rule removes registers from a node constraint if we know that these registers are occupied by other nodes, i.e. the constraints consisting of these registers are saturated. Reconsidering the example shown in Figure 4d, the nodes I_1 and I_2 can only be assigned to register a and b. Thus, I_3 cannot be assigned to one of these registers and we remove them from the constraint of I_3. The transformation removes all nodes from the upper constraint. Usually, we delete such an empty node after reconnecting all children to its parent, because an empty node serves no purpose and the removal may enable further rule applications. However, since $\{a, b, c\}$ is the root of our tree—holding only unconstrained nodes—we keep it.

Since we want to relax the constraints of dying nodes as much as possible, the rule shown in Figure 4c moves dying nodes upwards in the constraint hierarchy. This is only allowed, if the constraint C_1 does not contain non-dying nodes. For our example of Figure 4d we relax the constraints of I_1 and I_2 further to $\{a, b, c\}$. This would be disallowed without the application of 4b, because a or b could then be assigned to I_3, which would render a coloring of the results impossible.

4.3 Obtaining a coloring order

After exhaustive application of the transformation rules, we obtain an ordering of the constraints by a post-order traversal of the constraint hierarchy, so more constrained nodes are colored first. For example, in Figure 4d the node I_3 must be colored first due to this order. Within each constraint of the hierarchy, the associated values are further ordered with respect to a post-order traversal over the original constraint hierarchy. The second traversal ensures that "over-relaxed" values, i.e. values with a constraint that is less restrictive than their original constraint, are colored first. For our example in Figure 4d this means that we have to color I_1 before I_2, although their relaxed constraints are equal. The final node order is I_3, I_1, I_2. For the PBQP, we intersect the original (Figure 3a) and the relaxed constraints (Figure 3c); resulting in the constraints shown in Figure 3d. We now have an order to color the values live immediately before the constrained instruction. Likewise, we obtain an order for the results by coloring the more constrained values first. Finally, we obtain a coloring order for the whole PBQP graph by employing the PEO for the remaining (unconstrained) nodes. This order ensures that our PBQP solver finds a register assignment even in presence of constrained instructions.

5 Evaluation

In this section we evaluate the impact of our adaptations. First, the late decision is compared to early decision making. Also, we investigate the effects of RM.

	RM disabled		RM enabled	
Reduction	Applications	Ratio	Applications	Ratio
R0	2,047,038	—	2,013,003	—
RE	126,002	—	33,759	—
R1	106,828	13.9%	94,529	11.9%
R2	363,221	47.2%	382,705	48.0%
RN	298,928	38.9%	292,872	36.7%
RM		0.0%	26,850	3.4%

Table 1: Percentages of reduction types.

Finally, our approach is compared to the current libFirm allocator in terms of speed and result quality.

5.1 Early vs. late decision

As mentioned in Section 3.3 we implemented early decision as well as late decision. We evaluated both approaches using the C programs of the SPEC CINT2000 benchmark suite. The programs compiled with late decision do not reach the performance of the programs compiled with early decision for any benchmark, showing a slowdown of 3.9% on average. Especially the 253.perlbmk benchmark performs nearly 20% slower.

We think that the quality gap stems from the different handling of affinities. An early decision takes account of the surrounding affinity costs and propagates them during the reduction. For a late decision a node and incident affinity edges are removed from the PBQP graph first; then the decisions at adjacent nodes are made without accounting the affinity costs. When the late decision is made, the affinities may not be fulfilled due to decisions at interference neighbors that were not aware of these affinities.

5.2 Effects of RM

We added RM to our PBQP solver and Table 1 shows that 3.4% of the PBQP reductions during a SPEC compilation are RM. The number of nodes, remaining after the graph is completely reduced, is given in the R0 row, but technically these nodes are not "reduced" by the solver, so they are excluded from the ratio calculation. RE is also excluded, since it reduces edges instead of nodes. The heuristic RN makes up 36.7% of the reductions, so these decisions are significant. The number of independent edge reductions decreases to nearly a fourth in total, which suggests that a significant number of RE stem from nodes, whose assignment is determined by a heuristic reduction of a neighbor. In case of RM, those edges are "redirected" to this neighbor instead. Another effect is that the number of heuristic decisions decreases by 2%. This reflects nodes that can be

Benchmark	Recoloring	PBQP	Ratio
164.gzip	345	350	101.4%
175.vpr	446	444	99.7%
176.gcc	179	179	99.8%
181.mcf	336	335	99.6%
186.crafty	233	231	99.4%
197.parser	468	467	99.7%
253.perlbmk	355	354	99.8%
254.gap	252	253	100.4%
255.vortex	417	418	100.1%
256.bzip2	374	371	99.4%
300.twolf	684	680	99.4%
Average			99.9%

Table 2: Comparison of execution time in seconds with recoloring and PBQP.

optimally reduced after merging the neighbors into them. Altogether, the costs of the PBQP solutions decreased by nearly 1% on average, which shows that RM successfully improved the heuristic decisions.

5.3 Speed evaluation

To evaluate the speed of the compilation process with PBQP-based copy coalescing, we compare our approach to the recoloring approach [12]. Both approaches are implemented within the libFirm compiler backend, so all other optimizations are identical. The SPEC CINT2000 programs ran on an 1.60GHz Intel Atom 330 processor on top of an Ubuntu 10.04.1 system. We timed the relevant phases within both register allocators and compare the total time taken for a compilation of all benchmark programs. The recoloring approach uses 11.6 seconds for coloring and 27.3 seconds for copy coalescing, which is 38.9 seconds in total. In contrast, the PBQP approach integrates copy coalescing into the coloring, so the coloring time equals the total time. Here, the total time amounts to 31.5 seconds, which means it takes 7.4 seconds less. Effectively, register assignment and copy coalescing are 19% faster when using the PBQP approach instead of recoloring.

5.4 Quality evaluation

To evaluate the quality of our approach, we compare the best execution time out of five runs of the SPEC CPU2000 benchmark programs with the recoloring approach. The results in Table 2 show a slight improvement of 0.1% on average.

In addition, we assess the quality of our copy minimization approach by counting the inserted copies due to unfulfilled register affinities. We instrumented the Valgrind tool [15] to count these instructions during a SPEC run. Despite

	PBQP			Recoloring			Ratio	
Benchmark	Instr.	Swaps	Copies	Instr.	Swaps	Copies	Swaps	Copies
164.gzip	332	3.14%	0.71%	326	2.01%	0.19%	156.2%	374.7%
175.vpr	202	4.38%	0.29%	201	4.32%	0.25%	101.4%	113.8%
176.gcc	165	4.20%	0.30%	165	3.95%	0.28%	106.4%	108.9%
181.mcf	50	4.22%	0.00%	50	4.72%	0.00%	89.4%	1047207.5%
186.crafty	208	8.14%	0.56%	209	8.36%	0.64%	97.4%	87.7%
197.parser	365	4.13%	0.56%	366	4.48%	0.28%	92.2%	198.5%
253.perlbmk	396	4.64%	0.28%	408	5.97%	0.14%	77.7%	202.1%
254.gap	259	7.02%	0.08%	259	6.86%	0.40%	102.4%	20.1%
255.vortex	379	3.08%	0.53%	377	3.12%	0.16%	98.9%	339.2%
256.bzip2	295	6.30%	0.14%	298	6.15%	0.93%	102.4%	14.9%
300.twolf	306	4.80%	0.90%	306	4.33%	1.30%	110.6%	69.1%
Average	269	4.91%	0.40%	269	4.93%	0.42%	99.6%	95.2%

Table 3: Dynamic copy instructions in a SPEC run (in billions).

dynamic measuring, the results in Table 3 are static, because the input of the benchmark programs is static. Since the number of instructions varies between programs, we examine the percentage of copies. We observe that nearly 5% of the executed instructions are swaps and around 0.4% are copies on average. Because of the small number of copies a difference seems much higher, which results in the seemingly dramatic increase of 1047208% swaps for 181.mcf. On average the percentages decrease by 0.4% and 4.8%, respectively.

6 Future Work

Some architectures feature irregularities which are not considered in the context of SSA-based register allocation. The PBQP has been successfully used to model a wide range of these irregularities by appropriate cost matrices [16]. While the modelling can be adopted for SSA-based register assignment, guaranteeing a polynomial time solution is still an open problem.

7 Conclusion

This work combines SSA-based with PBQP-based register allocation and integrates copy coalescing into the assignment process. We introduced a novel PBQP reduction, which improves the quality of the heuristic decisions by merging nodes. Additionally, we presented a technique to handle hierarchic register constraints, which enables a wider range of options within the PBQP. Our implementation achieves an improvement over the SSA-based recoloring approach. On average, the relative number of swap and copy instructions for the SPEC

CINT2000 benchmark was reduced to 99.6% and 95.2%, respectively, while taking 19% less time for assignment and coalescing.

References

1. Bouchez, F., Darte, A., Rastello, F.: On the complexity of register coalescing. In: CGO '07: Proceedings of the International Symposium on Code Generation and Optimization. pp. 102–114 (2007)
2. Braun, M., Mallon, C., Hack, S.: Preference-guided register assignment. In: Compiler Construction, Lecture Notes in Computer Science, vol. 6011, chap. 12, pp. 205–223. Springer Berlin / Heidelberg (2010)
3. Briggs, P., Cooper, K.D., Torczon, L.: Improvements to graph coloring register allocation. ACM Trans. Program. Lang. Syst. 16(3), 428–455 (May 1994)
4. Brisk, P., Dabiri, F., Macbeth, J., Sarrafzadeh, M.: Polynomial time graph coloring register allocation. In: 14th International Workshop on Logic and Synthesis. ACM Press (2005)
5. Buchwald, S., Zwinkau, A.: Instruction selection by graph transformation. In: Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems. pp. 31–40. CASES '10, ACM, New York, NY, USA (2010)
6. Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.W.: Register allocation via coloring. Computer Languages 6(1), 47–57 (1981)
7. Dirac, G.: On rigid circuit graphs. Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg 25, 71–76 (1961)
8. Ebner, D., Brandner, F., Scholz, B., Krall, A., Wiedermann, P., Kadlec, A.: Generalized instruction selection using SSA-graphs. In: LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems. pp. 31–40. ACM, New York, NY, USA (2008)
9. Eckstein, E., König, O., Scholz, B.: Code instruction selection based on SSA-graphs. In: Software and Compilers for Embedded Systems, Lecture Notes in Computer Science, vol. 2826, pp. 49–65. Springer Berlin / Heidelberg (2003)
10. Grund, D., Hack, S.: A fast cutting-plane algorithm for optimal coalescing. In: Compiler Construction, chap. 8, pp. 111–125. Lecture Notes in Computer Science, Springer Berlin / Heidelberg (2007)
11. Hack, S.: Register allocation for programs in SSA form. Ph.D. thesis, Universität Karlsruhe (October 2007)
12. Hack, S., Goos, G.: Copy coalescing by graph recoloring. In: PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation (2008)
13. Hack, S., Grund, D., Goos, G.: Register allocation for programs in SSA-form. In: Compiler Construction, Lecture Notes in Computer Science, vol. 3923, pp. 247–262. Springer Berlin / Heidelberg (2006)
14. Hames, L., Scholz, B.: Nearly optimal register allocation with PBQP. In: Modular Programming Languages, Lecture Notes in Computer Science, vol. 4228, chap. 21, pp. 346–361. Springer Berlin / Heidelberg (2006)
15. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. SIGPLAN Not. 42(6), 89–100 (June 2007)
16. Scholz, B., Eckstein, E.: Register allocation for irregular architectures. In: LCTES-SCOPES. pp. 139–148 (2002)