Preference-Guided Register Assignment

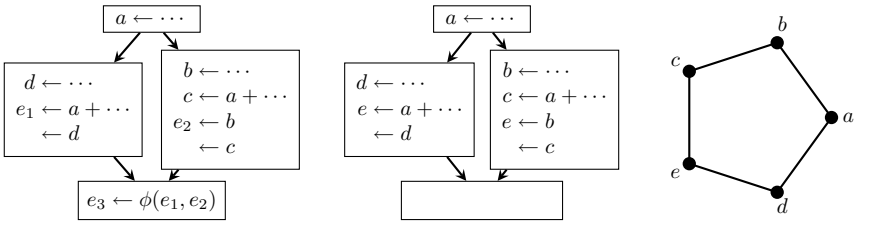
Matthias Braun[1], Christoph Mallon[2], and Sebastian Hack[2]

[1] Karlsruhe Institute of Technology
matthias.braun@kit.edu
[2] Computer Science Department
Saarland University
(mallon|hack)@cs.uni-saarland.de

Abstract. This paper deals with coalescing in SSA-based register allocation. Current coalescing techniques all require the interference graph to be built. This is generally considered to be too compile-time intensive for just-in-time compilation. In this paper, we present a biased coloring approach that gives results similar to standalone coalescers while significantly reducing compile time.

1 Introduction

The register allocation phase of a compiler maps the variables of a program to the registers of the processor. One important part of register allocation is coalescing. Coalescing is an optimization that tries to remove register-to-register move instructions by assigning the source and the target of the move the same register. One serious drawback of coalescing is that it can increase the register demand of the program. Consider the example in Figure 1. The register demand



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

Fig. 1: Coalescing a ϕ-function

in the SSA-form program P is 2 everywhere. If we perform classical SSA destruction and coalesce the move instructions represented by the ϕ-function, that is merge the live ranges of e_1, e_2, and e_3 into one (as shown in P'), we need

3 registers for a valid register assignment, as can be verified by coloring the interference graph G of P'.

Chaitin et al. [1] express register allocation by graph coloring and show that, if one makes no assumption about coalescing, every undirected *interference graph* G corresponds to a program P for which holds: An optimal register allocation for P is an optimal coloring of G. Although this approach is very popular, it has two undesirable properties:

- Because graph coloring is NP-complete, we need a heuristic to color such a graph. Hence, we might fail to color a graph with k colors although the graph is k colorable. For register allocation this means that we unnecessarily spill variables to memory.
- For any given $n \in \mathbb{N}$ there exists a graph that has a largest clique of size l but needs $l + n$ colors for an optimal coloring. The size of the largest clique in that graph corresponds to the register pressure in the program. Hence, as in the example above, we need $l + n$ registers although there are never more than l variables alive.

Consequently, recent register allocation approaches do not allow arbitrary coalescing of live ranges: In an SSA-form program, some live ranges are split by ϕ-functions. This splitting is sufficient to overcome both drawbacks mentioned above (see [2–4] for proofs):

1. An optimal register assignment can be computed in linear time.
2. The register pressure equals the minimum number of registers needed for the program.

Live-range splitting by ϕ-functions is not the only source of move instructions in a program. Treating register constraints as they are incurred by some architectures and application binary interfaces, also provokes the insertion of move instructions: Assume a variable v is an argument to a function call and the ABI dictates that it has to be in register `R1`. Then, we need to move v to `R1` in front of the call. On the other hand, if we assigned `R1` to v in the first place, we can save this move.

All these live-range splits result in move instructions. Usually, reducing the number of move instructions is the task of the *coalescing* phase of a register allocator. However, most of the existing coalescing techniques are very compile-time intensive: They all require the interference graph to be materialized as a data structure. Some of them even perform updates on that graph. However, in just-in-time compilation, constructing and updating the interference graph is considered too costly.

1.1 Contributions

In this paper, we pursue a new approach to coalescing: We assume that spilling already took place and the register pressure everywhere in the program is $\leq k$, where k is the number of available registers. Instead of delegating coalescing to

a separate phase, we make the assignment pass aware of move instructions by biasing the assignment: We try to assign sources and targets of move instructions the same register. To this end, we extend the conventional SSA register allocation algorithm by the following techniques:

- We compute *register preferences* for each variable. These preferences reflect the register constraints the variable is exposed to. Hence, instead of non-deterministically choosing a register during the assignment phase, we are able to make a more profound register choice. In doing so, we avoid many of the moves that are usually inserted due to register constraints. Section 3.1 discusses register preferences in more detail.
- When coloring the target of a move, e.g. the result of a ϕ-function, we propagate preferences for that color to the not-yet-colored sources, in this case the operands of the ϕ-function. Thus, when those variables are to be colored, we attempt to assign them the same register as the target of the ϕ-function. Section 3.3 gives a detailed discussion.
- When a variable is assigned to a register and the most preferable register is occupied by another variable, we allow for *optimistically* moving the occupying variable to a different register. Placing a variable in the preferred register from the start is often better than doing it right in front of the program point that caused the preference: If we assume that the register is occupied at that point we need two moves (one to free the register and one to move the variable to it) instead of the one needed to free the register upon the variable's definition. Details are discussed in Section 3.4.
- Based on profile data or estimated execution frequencies, we compute an order of the basic blocks in a control-flow graph that aids in removing more moves on frequently executed traces of the CFG (Section 4).

Our experimental evaluation (see Section 5) shows that coalescing in an SSA-based register allocator is important: The runtime of the benchmarks is decreased by 5% and the number of executed move instructions is decreased by 55% percent. Compared to our previous work based on graph recoloring [5], register allocation and coalescing is 2.27 times faster. Our compile-time measurements show a linear behavior of the presented algorithm.

2 SSA-based Register Allocation

This section reviews the basics of SSA-based register allocation and describes how register constraints are treated by an SSA-based allocator.

Register allocation on the SSA form uses the live-range splitting caused by ϕ-functions. The ϕ-functions of a basic block basically act as control-flow dependent *parallel* moves (see Figure 2). This splitting and the dominance property of the SSA form[3] cause the interference graphs for SSA-form programs to be chordal (see [2–4] for proofs). Chordal graphs have two properties that make them appealing for register allocation:

[3] The fact that each use of a variable is dominated by its definition.

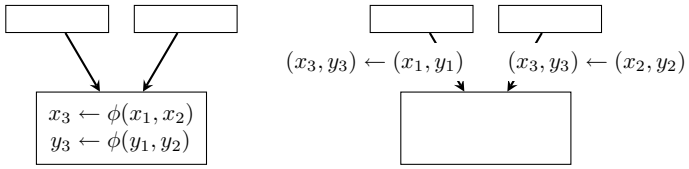


Fig. 2: ϕ-functions are parallel Moves

1. They are optimally colorable in time $O(\omega(G) \cdot |V|)$ where $\omega(G)$ is the size of the largest clique in G and V is the set of G's nodes.
2. The size of the largest clique in the graph is equal to the minimum number of colors needed for a coloring - the graph's chromatic number.

Furthermore, for each clique in the interference graph there is a location in the program where all the variables of the clique are alive. Thus, unlike conventional graph-coloring register allocation, lowering the register pressure to the number of available registers k results in a k-colorable interference graph. Hence, pressure-based spilling heuristics [6–8] already lead to k-colorable interference graphs.

2.1 Register Assignment

After the spilling phase has lowered the register pressure everywhere to at most k, registers can be assigned. While the interference graph is helpful to reason about, it actually never has to be built as a data structure when assigning registers. A SSA interference graph can be colored using a node elimination algorithm like the one used by Chaitin et al. [1] in their seminal paper. However, the advantage of SSA-based register allocation is that this elimination order coincides with dominance:

> Before a variable v can be eliminated, all variables that dominate v and interfere with v have to be eliminated.

Consequently, an order that colors a program point only after its dominators have been colored leads to an optimal coloring of the SSA interference graph.

Algorithm 1 shows the assignment pass for a single basic block B. This algorithm is then applied to every basic block such that the immediate dominator of B is processed before B itself (in Section 4 we propose a specific coloring order). We maintain a bit set occupied of registers used by currently live variables. We initialize this bitset with the registers of the values that are live-in at the beginning of B. Note that all live-in values already have a register assigned because:

1. The definition of a variable dominates all program points where it is alive.
2. All dominators of B have already been processed.

Then, all ϕ-functions of B are assigned. The arguments of the ϕ-functions are ignored in B because they correspond to move instructions in the predecessor blocks and hence don't represent live values in B.

The instructions inside the basic block are now processed in order: For every variable that dies at a program point, the register is put back into the pool of free registers. For every value which is defined by an instruction, a free register is chosen (function get_register) and put into the occupied set.

Algorithm 1 Coloring of a basic block

```
proc color_block(block):
    # Determine initial register occupation and color φ−nodes
    occupied ← ∅
    for val in block.live_in:
        occupied ← occupied ∪ { val.register }
    for phi in block.phi_nodes:
        phi.register ← get_register(phi, occupied)
        occupied ← occupied ∪ { phi.register }

    # Assign registers
    for insn in block.instructions:
        enforce_constraints(insn)
        for a in insn.arguments:
            if dies(a, insn):
                occupied ← occupied \ { a.register }
        for r in insn.results:
            r.register ← get_register(r, occupied)
            occupied ← occupied ∪ { r.register }

    block.processed ← true
    # Create φ−moves where necessary
    for pred in block.preds:
        if pred.processed:
            implement_phi_copies(pred, block)
    for succ in block.succs:
        if succ.processed:
            implement_phi_copies(block, block.succs[0])
```

2.2 Register Constraints

In practice, the instruction set architecture (ISA) and the application binary interface (ABI) impose several constraints on the registers that are allocatable

for a variable *at* a program point. Most prominent and omnipresent are caller- and callee-save registers across function calls. For example, the x86 ABIs state that the contents of the registers `eax`, `ecx`, and `edx` are destroyed after a function call. The return value of a function returning an `int` is delivered in `eax`.

Traditionally, such constraints are handled by splitting the live ranges of all variables alive across such a constrained instruction by inserting a parallel move instruction. In doing so, all registers become available in front of that instruction and the assignment pass can easily compute an assignment that fulfills these constraints. In Algorithm 1 this is expressed by the function enforce_constraints which we do not describe in further detail here. Figure 3 gives an example of a constrained call instruction and the inserted parallel move[4].

$$w\langle \texttt{R0} \rangle \leftarrow \texttt{call}(x\langle \texttt{R1} \rangle, y\langle \texttt{R2} \rangle)$$
$$\vdots$$
uses of x, y, z, w

(a) A constrained instruction

$$(x'\langle \texttt{R1} \rangle, y'\langle \texttt{R2} \rangle, z') \leftarrow (x, y, z)$$
$$w\langle \texttt{R0} \rangle \leftarrow \texttt{call}(x', y')$$
$$\vdots$$
uses of x', y', z', w

(b) Splitting live ranges

Fig. 3: A call instruction with register constraints

To model register constraints, we annotate every program point ℓ with two partial functions (one for the defined and one for the used variables) that map a variable that has a register constraint at that program point to the register it is required to be in:

$$\mathit{constr}_\ell^{use} : \mathit{Var} \hookrightarrow \mathit{Reg} \qquad \mathit{constr}_\ell^{def} : \mathit{Var} \hookrightarrow \mathit{Reg}$$

where Var is the set of variables and $\mathit{Reg} \subset \mathbb{N}$ is the set of registers. For the example in Figure 3, we have:

$$\mathit{constr}_\ell^{use}(x) = \texttt{R1}, \mathit{constr}_\ell^{use}(y) = \texttt{R2} \qquad \mathit{constr}_\ell^{def}(w) = \texttt{R0}$$

2.3 Implementing Parallel Moves

The parallel move instructions are implemented *after* register assignment. Concerning the assigned registers, a parallel move corresponds to a register permutation that can be implemented with moves, swaps, xors, and so on [9]. For example, assume our architecture has four registers. Consider the following parallel move and a register allocation (indicated by the superscripts):

$$\text{Move: } (a^4, b^2, c^3) \leftarrow (d^3, e^1, f^4) \qquad \text{Permutation: } \begin{bmatrix} 2 & 3 & 4 \\ 1 & 4 & 3 \end{bmatrix}$$

[4] Register constraints are indicated in angle brackets.

This can be implemented with the following sequence of instructions:

```
move R2 ← R0
swap R3, R4
```

3 Coalescing with Register Preferences

In principle, Algorithm 1 can compute *any* legal register assignment for a CFG. The set of valid register allocations is basically characterized by the freedom of the function get_register: Whenever a register is assigned to a variable, get_register can choose among a set of free registers. However, regarding coalescing, not all valid allocations are equally preferable. An allocation in which many sources and targets of moves have the same color is *better* because it will result in less shuffle code in the program. Given an oracle telling us the best register for each variable, the algorithm would produce an optimal coalescing[5].

Algorithm 2 Choosing a register by preference

```
proc get_register(var, occupied):
    sort var.prefs by preference
    for (reg,pref) in var.prefs:
        if reg ∉ occupied:
            return reg
```

Unfortunately the coalescing problem is NP-hard even for programs in SSA-form [9, 3]. We thus rely on a heuristic approach that is guided by register preferences which are calculated before coloring and can be updated while allocating. To this end, we introduce a *preference analysis* that computes a preference vector for every variable. Such a vector has a component for every register. The higher the value of a component, the more preferable it is to assign the variable to the corresponding register. This vector is then used by get_register to select a "good" register (see Algorithm 2). The following sections describe the preference analysis and a mechanism to adjust the preferences while assigning registers for ϕ-functions.

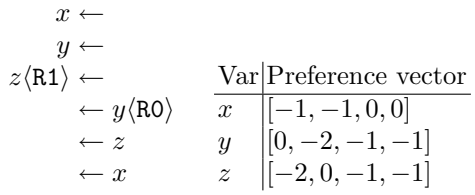
3.1 Register Preferences

Consider the example in Figure 4a. Assume the set of available registers when y is colored to be $\{\texttt{R0}, \texttt{R2}\}$ and assume the allocator (nondeterministically) chooses R2. Then, in front of its use, y has to be moved from R2 to R0 in order to fulfill

[5] Optimal for a given set of parallel moves. There are cases where a different placement of parallel moves can lead to a better overall result.

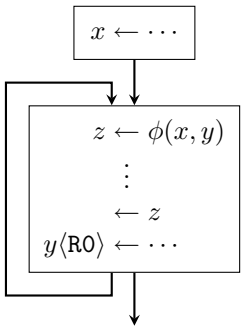
the register constraint. If the allocator knew that y is needed in R0, it could have selected it in the first place.

To make a sensible choice in the presence of register constraints, we need to propagate information from constrained uses of variables to the point where the color selection is done.



Var	Preference vector
x	$[-1,-1,0,0]$
y	$[0,-2,-1,-1]$
z	$[-2,0,-1,-1]$

(a) Constrained live ranges



(b) A ϕ-function with constrained arguments

Fig. 4: Examples for Preferences and constrained ϕ-functions

Reconsider the live ranges in Figure 4a. When assigning registers to the variables, we first assign a color to x. Since x interferes with two variables (y, z) which have constrained definitions or uses to R1 and R0, it would be good to choose one of the other registers: R2 or R3. If we would assign x to R0, we would have to move it aside to make room for y right in front of its constrained use. Correspondingly, the variables y and z should have a strong dislike for all registers other than the ones occurring in their constraints. Furthermore, they interfere with each other, so they have an even stronger dislike for each other's preferred register. Our analysis which is explained in the next section, computes the preference vectors shown in Figure 4a.

Thus, the allocator puts x in register R2 or R3 and leaves R0 and R1 untouched. y and z can then be directly allocated to R0 and R1 obviating any moves.

3.2 Preference Analysis

The register preference vector $pref(v)$ of a variable v is given by

$$pref(v) = \sum_{\{\ell | v \text{ is alive before } \ell\}} f_\ell \cdot \boldsymbol{c}_\ell^{use}(v) + \sum_{\{\ell | v \text{ is alive after } \ell\}} f_\ell \cdot \boldsymbol{c}_\ell^{def}(v)$$

where f_ℓ denotes the execution frequency of program point ℓ. This execution frequency can either be gathered from profile data or estimated (see e.g. [10]). For the sake of brevity, let $\square \in \{use, def\}$. $\boldsymbol{c}_\ell^\square(v)$ is the constraint vector concerning the used (defined) variables of program point ℓ for variable v:

$$\boldsymbol{c}_\ell^\square(v) := \begin{cases} \boldsymbol{e}_i - \mathbf{1} & \text{if } v \in \text{ dom } constr_\ell^\square \text{ and } i = constr_\ell^\square(v) \\ -\sum_{i \in R} \boldsymbol{e}_i & \text{else} \qquad \text{with } R = \text{ran } constr_\ell^\square \end{cases}$$

where $\boldsymbol{e}_i$ is the vector that is one at component i and zero everywhere else. $\mathbf{1}$ is the vector containing only ones.

Thus, the preference vector of a variable contains the sum of dislikes (negative preferences) caused by register constraints of program points where the variable is alive. To calculate the preferences, we perform a backward walk over the program's basic blocks so we can keep track of live values. When we encounter a constrained definition/use we add preferences to all other variables alive at that point. This is a simple flow-insensitive analysis and can be done in a single pass over the program.

3.3 Affinity Chunks

Besides register constraints, ϕ-functions are the second source of shuffle code. A "bad" register assignment can cause a cascade of move instructions to be inserted at the end of a ϕ predecessor block. In contrast to constrained instructions, the desirable register of an operand of a ϕ-function is not fixed a priori: It depends on which registers the other operands and the result variable of the ϕ are allocated to. Therefore, we do *not* consider ϕ-functions when performing the preference analysis *but* modify the preference vectors during the assignment process. When coloring a ϕ-function, a preference for the chosen color is added to the preference vectors of the still uncolored variables of the same affinity chunk.

A second observation is that the constraints of the arguments of a ϕ-function affect the ϕ-function as well. Consider the example in Figure 4b. One variable of the affinity chunk of the ϕ-function needs to be in `R0` upon its definition. Assigning z any other register than `R0` will cause a move on the loopback edge which needs to be avoided at all costs. Hence, we *propagate* the preference for `R0` to the whole affinity chunk of y and thus try to assign x and z to `R0` as well. In general, the preferences for all members of an affinity chunk are weighted by their execution frequencies and distributed among its members.

When coloring a ϕ-function, we want to assign that register to all not-yet colored variables of the ϕ's affinity component. However, such an affinity component can exhibit interferences within itself. Thus, one usually splits up the affinity components into interference-free *chunks* by *aggressive coalescing*. Aggressive coalescing itself is an NP-complete problem; it is an instance of a minimum multi-cut (see [9, 3] for example). In practice, one is content with a heuristic that greedily tries to merge chunks. Let C and D be two chunks that we want to merge. To merge the chunk, there must not exist an interference between

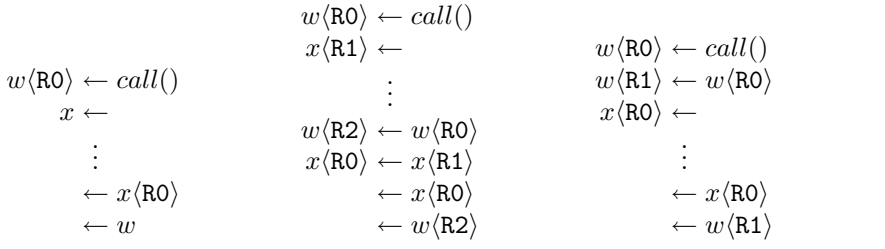
both chunks. If there is, the chunks cannot be merged and we "sacrifice" every affinity edge between both chunks. That means, that we no longer try to assign the same color to the variables of the move instruction, represented by the lost affinities. Of course, the order in which the chunks are merged decides on how good the results are, i.e. how many moves are introduced. This greedy heuristic requires an interference check between the two chunks. Naively, one could test each variable pair for interference, resulting in a quadratic algorithm. Recently, Boissinot et al. [11] gave a linear algorithm, exploiting SSA properties, to perform that check. However, this linear check still has to be performed whenever two chunks are to be merged.

To avoid this overhead, we do not split chunks up to the last interference edge but allow for remaining interferences within a chunk. This does not pose any correctness problems, as we use these chunks only to propagate register preferences when a ϕ-function is colored. In the worst case, we propagate preferences to a variable that interferes with that ϕ-function.

Our "approximated" chunks are computed using a union-find data structure. Whenever we encounter a ϕ-function, we check, whether the result variable of that ϕ-function and its operands interfere. This is can be done efficiently since we still have the set of live-in variables calculated by the liveness analysis. The chunk of an operand is merged with the ϕ's if the operand and the ϕ do not interfere. This can be done in hand with the preference analysis.

3.4 Optimistic Move Insertion

There is further room for reducing the number of move instructions: The fixed positions of the parallel moves aren't always optimal. A typical situation is shown in Figure 5a:



(a) Program P (b) Assignment for P (c) Assignment with optimistic move

Fig. 5: Candidate for optimistic move insertion

When the allocator reaches the assignment to variable x register R0 is already occupied by w. A classical allocator would assign the next free register

to x, say `R1`. A fixup would only occur before the constrained use of x. At this point however at least 2 move instructions are necessary: Variable w has to be moved away from `R0` and variable x into it. Instead, it is more beneficial to move variable w away from `R0` before the assignment to x as shown in Figure 5c compared to Figure 5c.

This situation is handled by optimistically inserting such early moves into the program: When the allocator finds that a desired output register is occupied by another variable then we determine the costs of moving that variable into another register. The cost is the sum of the preference differences when freeing the register by moving the occupying variable away and the preference differences when assigning the next possible register instead of the desired one. We compare these costs with the execution frequency of the current block. Higher costs are an indication that a move at the current position is cheaper than a later fixup. The move instruction is created optimistically. An improved version of get_register is shown in Algorithm 3.

Algorithm 3 Choosing a register with optimistic move insertion

```
proc get_register(var, occupied):
    sort var.prefs by preference
    for (reg,pref) in var.prefs:
        if reg ∉ occupied:
            return reg

        # Determine costs for moving the variable
        # which occupies the register away
        ovar ← reg.current_variable
        sort ovar.prefs by preference
        for (oreg,opref) in ovar.prefs:
            if oreg ∉ occupied:
                other_win ← opref − oreg.current_pref
                break
        next_pref ← preference value for next register
        win ← next_pref − pref
        if win + other_win > block.execfreq:
            create move from reg to oreg
            return reg
```

4 Block Coloring Order

To retain the properties by SSA-based register allocation, we color basic blocks in dominance order. This still provides many valid visiting orders. We choose an

Algorithm 4 Determining the block coloring order

```
proc blockorder():
    for b in reverse_postorder(blocks):
        t ← 0
        for p in control_flow_predecessors(b):
            if t < trace[p]:
                t ← trace[p]
        trace[b] ← t + frequency(b)
    order ← ∅
    for b in sort(blocks, by: trace)
        order ← add_trace(order, block)
    return order

proc add_trace(order, block):
    if not block ∈ order:
        best_trace ← 0
        best_pred ← null
        for p in preds(block)
            if backedge(p, block): continue
            if best_trace < trace[p]:
                best_trace ← trace[p]
                best_pred ← p
        if not best_pred ← null:
            order ← add_trace(order, block)
        order ← order + block
    return order
```

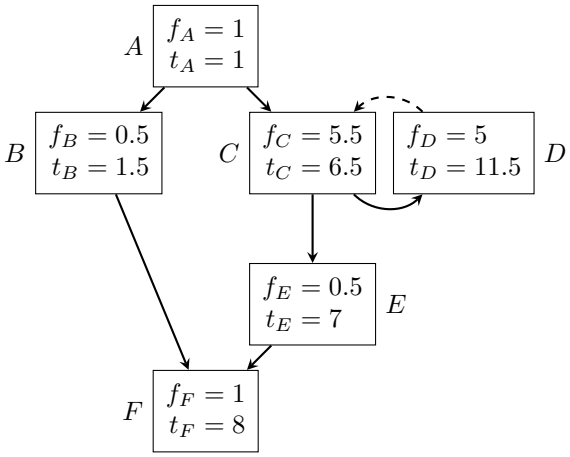


Fig. 6: A control-flow graph annotated with execution frequencies (f) and trace values (t)

order in which we color the most often executed basic blocks first while coloring paths beginning at the start block. By following the control flow along the "hot" paths, there is always one control flow predecessor colored already and we can assign ϕ-functions the same color as their operands in this predecessor.

To determine these paths in the control flow graph, we calculate a trace value for each basic block: First we gather execution frequencies for each basic block. This can be done heuristically (cf. Wagner et al. [10]) or they can be obtained from profiling information. Using the execution frequencies, we calculate the trace value of each block: The trace value of a block is the maximum of the trace values of its control flow predecessors (disregarding back edges) plus its own execution frequency. This approximates the amount of instructions executed from the start to each block while considering that a block can be executed multiple times.

Then we select the block with the highest trace value and determine a path to the start. Before this block is colored, we color its control flow predecessor (again ignoring back edges) which has the highest trace value. In turn, we repeat this until we reach the start block. This path then is colored in reverse order. After that, we select the block with the highest trace value from the remaining uncolored blocks and again construct a path towards the start block but this time stopping at some already colored block. Again, this new path is colored in reverse order and the process is repeated until all blocks are colored. Algorithm 4 shows the procedure as pseudo code.

In the example in Figure 6 the block with the highest trace value is D, therefore we first color the path A, C, D. Of the remaining, i.e. uncolored, blocks block F has the highest trace value, so we color its path E, F (A and C are already colored). B is colored last.

5 Experimental Evaluation

We implemented the presented coalescing algorithm in the libFirm [12] compiler. This compiler produces code for the x86 architecture and features a completely SSA-based register allocator as presented in [9]. All measurements were conducted on the integer part `CINT2000` of the `CPU2000` benchmark [13]. The program `252.eon` is missing because the compiler does not support C++. The time measurements were performed on a Core 2 Duo 2GHz PC with 2GB RAM running a Linux 2.6.24 kernel. The benchmarks mostly exercise the seven general-purpose registers of the x86. The execution frequencies were statically estimated using a Markov-chain model [10]. We compare the algorithm presented in this paper with our previous work performing colaescing by recoloring[5] after register allocation.

5.1 Compile Time

Figure 7 shows the runtime of the preference-guided assignment algorithm described in this paper running on the entire `CINT2000` benchmark set. We do not show CFGs larger than 2000 instructions because they are rare and unnecessarily scale the figure. The runtime behavior of the few CFGs not shown is consistent with those shown.

CFGs as large as 2000 instructions are processed well within 20 msecs ($\triangleq 10\mu$s per instruction) on the machine we experimented on. On average, an instruction took $6.2\mu s$ to allocate while the average speed of the recoloring approach is $14.1\mu s$. In comparison to the recoloring algorithm the approach presented here accelerates the allocation by a of factor 2.27.

5.2 Code Quality

We evaluate the quality of the produced code based on two experiments:

1. Counting the number of executed move/swap instructions in the benchmarks.
2. Measuring actual runtime of the benchmarks.

Counting moves and exchanges. By instrumenting the created binaries using Valgrind [14], we counted the number of move and swap instructions in the *runs* of the benchmarks. Table 1 shows the results of counting the move/exchange instructions.

The column "No Coalescing" corresponds to not performing any sophisticated coalescing at all: For live-range splits that are due to register constraints, get_register will try to assign targets and corresponding sources at parallel moves the same register if possible. Else, no effort is made to coalesce copies.

The column "Pref. Guided" denotes the algorithm presented in this paper and "Recoloring" is the aforementioned recoloring approach. For every evaluated coalescing algorithm, we show the number of move/swap instructions and the

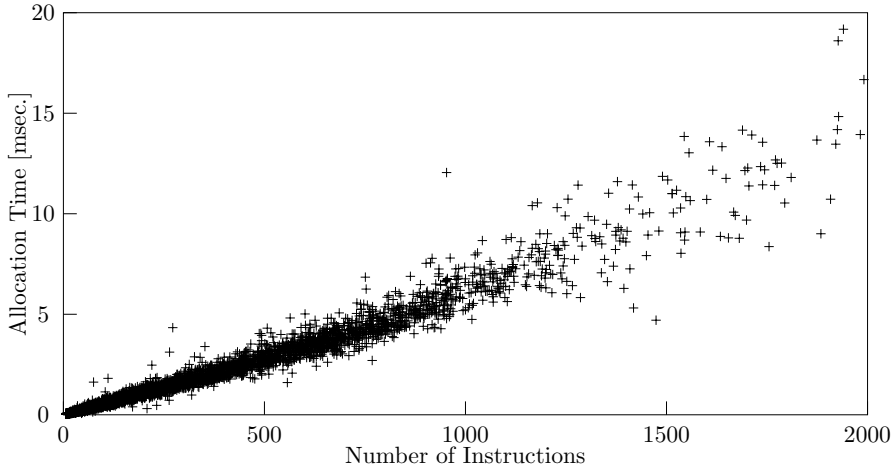


Fig. 7: Allocator runtime

Benchmark	No Coalescing			Pref. Guided			Recoloring		
	Copies	Swaps	Percent	Copies	Swaps	Percent	Copies	Swaps	Percent
164.gzip	24.1	18.3	11.76%	8.5	2.0	3.22%	5.8	0.3	1.88%
175.vpr	19.2	7.0	12.12%	11.7	1.1	6.28%	7.5	1.0	4.28%
176.gcc	16.9	7.9	14.02%	7.8	0.9	5.42%	6.5	0.4	4.37%
181.mcf	4.4	3.1	13.66%	3.3	0.0	6.67%	2.9	0.0	5.89%
186.crafty	29.0	4.9	16.30%	18.7	1.1	10.16%	17.5	1.0	9.58%
197.parser	34.4	11.9	13.19%	16.2	3.1	5.98%	13.9	1.8	4.93%
253.perlbmk	50.0	19.3	15.69%	23.3	1.4	6.23%	21.6	0.6	5.62%
254.gap	31.2	6.2	13.81%	17.2	1.4	7.40%	15.5	1.1	6.65%
255.vortex	44.0	3.8	13.11%	11.2	0.7	3.69%	9.5	0.3	3.03%
256.bzip2	34.6	9.8	14.14%	19.9	1.7	7.53%	17.0	3.1	7.01%
300.twolf	17.4	17.6	10.89%	10.3	5.6	5.25%	8.0	3.4	3.85%
Average	27.7	10.0	13.47%	13.5	1.7	5.93%	11.4	1.2	4.97%

Table 1: Number of executed move and swap operations in billions

percentage of all instructions being moves or exchanges. The preference-guided approach significantly reduces moves and swaps but does not reach the performance of the recoloring apporach. Performing almost no coalescing results in 13.46% of all executed instructions being moves or swaps. This number is decreased by our approach to 5.93% and to 4.97% by the recoloring technique. Hence, the code quality of the technique presented in this paper is very close to the recoloring approach which currently is one of the best conservative coalescers [5].

Runtime of the benchmarks. Figure 8 shows the runtime of the benchmarks normalized to "No Coalescing" as explained above. We see that performing coalescing *is* important and moves are not for free: The benchmark runtimes are decreased by 5%. Furthermore, the preference-guided approach is on par with the recoloring technique. Between those two, there is no clear winner. However, we suspect (without having verified this claim) that a smaller CPU with less pipelines and no out-of-order scheduling is more susceptible to register moves. Therefore, the recoloring approach might produce faster programs on such systems.

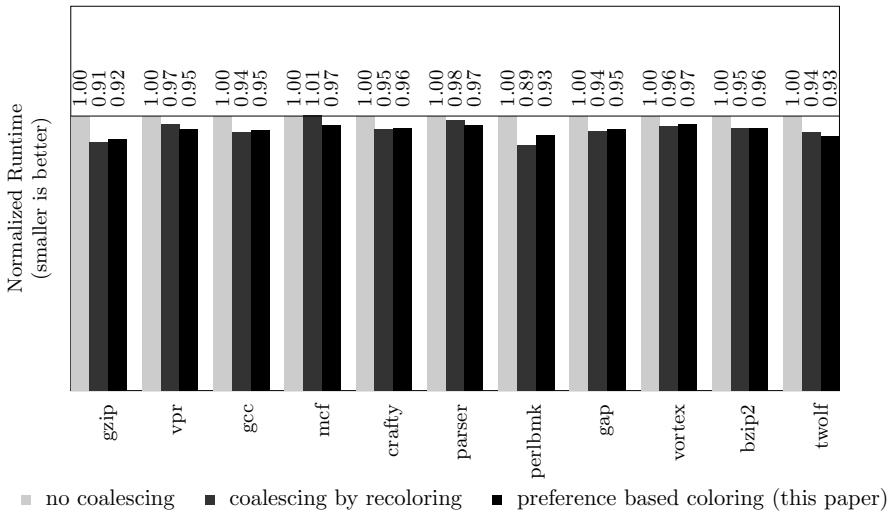


Fig. 8: SPEC `CINT2000` runtimes with different coalescing schemes

Finally, to show that our compiler produces high-quality results and the SSA-based register allocation technique is competetive, we compare the benchmark runtimes against those produced by GCC 4.2.4 and LLVM 2.5. libFirm has the smallest code base among these compilers and performs only a subset of the optimizations the others do. All compilers ran on maximum optimization level

and had machine-dependent optimizations for the benchmarking machine (see above) turned on[6]. As can be seen in Figure 9 the runtime of the benchmark programs produced by our compiler is on par with the others.

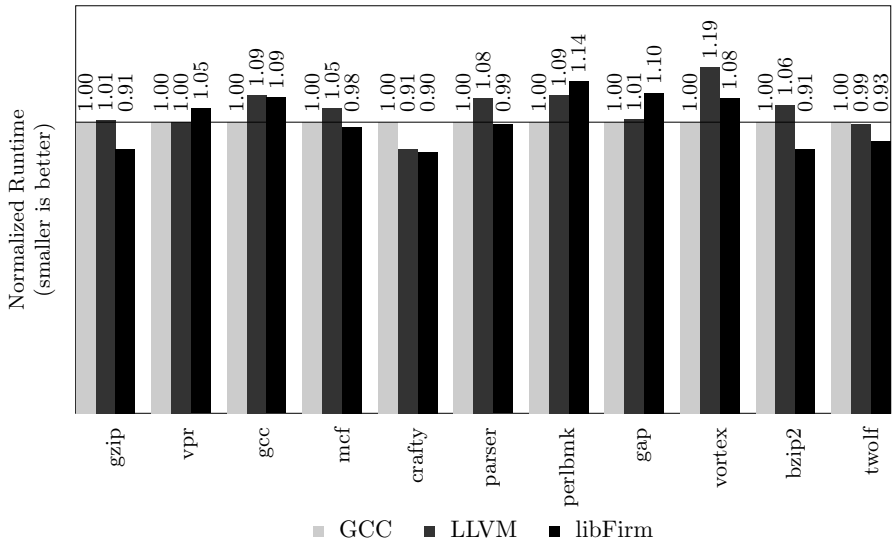


Fig. 9: SPEC `CINT2000` runtimes relative to GCC and LLVM

6 Related Work

Graph-based approaches. The first graph-coloring allocator due to Chaitin et al. [1] used *aggressive coalescing* and did not make any effort at all to respect the chromatic number of the graph. Since then, a lot of work was done on safe coalescing. Briggs et al. [15] introduced *conservative coalescing.* To decide whether an affinity can be coalesced, they considered the degree of the resulting coalesced node. Only if that node's degree was lower than k, the copy was coalesced. George and Appel's *iterated coalescing* [16] improves upon conservative coalescing by applying Briggs et al.'s criterion and a new one iteratively to the graph. Park and Moon [17] left the road of safe coalescing and improved upon the aggressive scheme.

Live-range splitting. Live-range splitting has often been proposed to aid coloring. To our knowledge, Fabri [18] was first to observe this. Appel and George [19] presented an ILP approach to reduce the register pressure everywhere to k by

[6] `-O3 -fomit-frame-pointer -march=native`

allowing every live range being split at every program point. Lueh et al.'s fusion based allocator [20] integrates live-range splitting into the register allocator. They start by building the interference graphs of certain regions (that can be basic blocks, loops, traces, etc.) that are not imposed by the allocator but can be chosen by the compiler writer. In a later step, the interference graphs are fused to form the complete interference graph. During this fusion process, live ranges can be split or spilled if the fused interference graph was no longer colorable. Recently, Nakaike et al. [21] proposed a dynamic approach that splits around basic blocks and uses coalescing to unify split live-ranges in hot code regions.

Linear-scan allocators. Wimmer and Mössenböck [22] give a highly tuned extension of Traub's version [23] of linear scan. Their *register hints* is a similar technique to our preference propagation for ϕ-functions. Furthermore, they can take register constraints into account. Recently, Sarkar and Barik [24] introduced more aggressive live-range splitting to linear scan allocators however without performing coalescing.

SSA-based register allocation. Budimlic et al. [25] pioneered in coalescing on SSA-form programs already using many properties that SSA-based register allocation relies on. However, they are only concerned with aggressive coalescing. In 2005, three groups [26, 2, 4] independently from each other discovered that the interference graphs of SSA-form programs are chordal. All yet published coalescing techniques tailored to SSA-based register allocation use interference graphs.

Bouchez et al. [3] investigate the theoretic background of coalescing. They show that coalescing is NP-complete concerning the number of affinities, also in the SSA-based setting. Later, Bouchez et al. proposed several extensions to conservative coalescing [27]. Brisk [28] presents a biased coloring algorithm for chordal graphs. Hack et al. [4, 5] present two approaches based on recoloring: First, the program is colored using the standard algorithm presented in Section 2. Then, the color assignment is changed by assigning move-related nodes the same color. Color clashes are resolved recursively through the graph.

Pereira and Palsberg [29] consider the problem of subregisters. In this setting, optimal allocation even inside a basic block is NP-complete. Therefore, they split live ranges after every program point and allocate each instruction separately. In doing so, they process the program points in dominance order and perform coalescing only along dominance order. Especially, moves on loop back edges are not coalesced.

7 Conclusions

In this paper, we presented an SSA-based register assignment algorithm that uses register preferences to bias the register assignment in order to reduce shuffle code. In doing so, we do not need a separate coalescing pass in the register allocator. Furthermore, building the interference graph, which is considered a red rag for

just-in-time compilation, is no longer necessary. Compared to a state-of-the art coalescing technique, our algorithm gives competitive results while reducing the runtime of the register allocation by a factor of 2.27.

Acknowledgements. We thank Michael Beck, Alain Darte, Gerhard Goos, Daniel Grund, Fabrice Rastello, Jan Reineke, and Christian Würdig for several insightful discussions. Furthermore, we thank the anonymous reviewers for their valuable comments.

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. Brisk, P., Dabiri, F., Jafari, R., Sarrafzadeh, M.: Optimal Register Sharing for High-Level Synthesis of SSA Form Programs. IEEE Trans. on CAD of Integrated Circuits and Systems **25**(5) (2006) 772–779
3. Bouchez, F., Darte, A., Rastello, F.: On the Complexity of Register Coalescing. In: CGO, San Jose, USA, IEEE Computer Society Press (March 2007)
4. Hack, S., Grund, D., Goos, G.: Register Allocation for Programs in SSA Form. In Zeller, A., Mycroft, A., eds.: CC. Volume 3923., Springer (March 2006) 247–262
5. Hack, S., Goos, G.: Copy Coalescing by Graph Recoloring. In: PLDI, New York, NY, USA, ACM (2008) 227–237
6. Braun, M., Hack, S.: Register Spilling and Live-Range Splitting for SSA-Form Programs. In: CC. Volume 5501 of LNCS., Springer (2009) 174–189
7. Morgan, R.: Building an Optimizing Compiler. Digital Press, Newton, MA, USA (1998)
8. Paleczny, M., Vick, C., Click, C.: The Java HotSpot$^{\text{TM}}$Server Compiler. In: Proceedings of the Java$^{\text{TM}}$ Virtual Machine Research and Technology Symposium (JVM '01). (April 2001)
9. Hack, S.: Register Allocation for Programs in SSA Form. PhD thesis, Universität Karlsruhe (October 2007)
10. Wagner, T.A., Maverick, V., Graham, S.L., Harrison, M.A.: Accurate Static Estimators for Program Optimization. In: PLDI, New York, NY, USA, ACM (1994) 85–96
11. Boissinot, B., Darte, A., Dupont de Dinechin, B., Guillon, C., Rastello, F.: Revisiting out-of-SSA translation for correctness, code quality, and efficiency. In: CGO, IEEE Computer Society Press (2009) 114–125 Best paper award.
12. The libFirm Compiler. http://www.libfirm.org
13. Standard Performance Evaluation Corporation: SPEC CPU2000 V1.3
14. Valgrind Instrumentation Framework for Building Dynamic Analysis Tools. http://www.valgrind.org
15. Briggs, P., Cooper, K.D., Torczon, L.: Improvements to Graph Coloring Register Allocation. TOPLAS **16**(3) (1994) 428–455
16. George, L., Appel, A.W.: Iterated Register Coalescing. TOPLAS **18**(3) (1996) 300–324
17. Park, J., Moon, S.M.: Optimistic Register Coalescing. ACM Transactions on Programming Languages and Systems **26**(4) (2004) 735–765

18. Fabri, J.: Automatic Storage Optimization. In: SIGPLAN '79: Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction, New York, NY, USA, ACM Press (1979) 83–91
19. Appel, A.W., George, L.: Optimal Spilling for CISC Machines with Few Registers. In: ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation. (June 2001) 243–253
20. Lueh, G.Y., Gross, T., Adl-Tabatabai, A.R.: Fusion-based Register Allocation. ACM Transactions on Programming Languages and Systems **22**(3) (2000) 431–470
21. Nakaike, T., Inagaki, T., Komatsu, H., Nakatani, T.: Profile-based Global Live-Range Splitting. In: PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, New York, NY, USA, ACM (2006) 216–227
22. Wimmer, C., Mössenböck, H.: Optimized interval splitting in a linear scan register allocator. In: VEE '05: Proceedings of the 1st ACM/USENIX international Conference on Virtual Execution Environments, New York, NY, USA, ACM Press (2005) 132–141
23. Traub, O., Holloway, G., Smith, M.D.: Quality and speed in linear-scan register allocation. In: PLDI '98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, New York, NY, USA, ACM Press (1998) 142–151
24. Sarkar, V., Barik, R.: Extended linear scan: An alternate foundation for global register allocation. In: Compiler Construction 2007. Volume 4420., Springer (2007) 141–155
25. Budimlić, Z., Cooper, K.D., Harvey, T.J., Kennedy, K., Oberg, T.S., Reeves, S.W.: Fast copy coalescing and live-range identification. In: PLDI, ACM Press (2002) 25–32
26. Bouchez, F., Darte, A., Guillon, C., Rastello, F.: Register Allocation: What does the NP-Completeness Proof of Chaitin et al. really prove? or revisiting Register Allocation: Why and How? In: LCPC, New Orleans, USA (Nov 2006)
27. Bouchez, F., Darte, A., Rastello, F.: Advanced Conservative and Optimistic Register Coalescing. In: CASES. (2008) 147–156
28. Brisk, P., Verma, A.K., Ienne, P.: An Optimistic and Conservative Register Assignment Heuristic for Chordal Graphs. In: CASES. (2007) 209–217
29. Pereira, F., Palsberg, J.: Register Allocation by Puzzle Solving. In: PLDI, New York, NY, USA, ACM (2008) 216–226