The Design and Implementation of a SSA-based Register Allocator

Fernando Magno Quintao Pereira

UCLA - University of California, Los Angeles

Abstract. A state-of-the-art register allocator is among the most complicated parts of a compiler, partly because register allocation is NP-complete in general. Surprisingly, Bouchez, Brisk et al., and Hack independently discovered in 2005 that if a program is in single static assignment (SSA) form, then a core register allocation problem can be solved in polynomial time. This result enables what we call SSA-based register allocation. In this report we present the first design and implementation of an SSA-based register allocator, integrated into LLVM. Compared to state of the art, our register allocator is much simpler and generates code of equivalent quality. We show the new static analyses we use for phi-lifting, spilling, and coalescing, and we explain that the choice of those analyses influence how we must do SSA deconstruction.

1 Introduction

Register allocation is the process of mapping a program that uses an unbounded number of *virtual variables* into a program that uses a fixed number of *physical registers*, in such a way that virtuals with overlapping live ranges are assigned different registers. If the number of registers is not enough to accommodate all the virtuals alive at some point in the control flow graph, some of these variables must be mapped into memory. These are called *spilled virtuals*.

The *Static Single Assignment* (SSA) form is an intermediate representation in which each variable is defined at most once in the program code [9, 19]. Prior work [3] has conjectured that the use of SSA Form could be beneficial to register allocation due to live range splitting; however, it was only in 2005 and 2006 that several research groups [2, 5, 13] have shown that interference graphs for *regular programs* [1] are chordal, and can be efficiently colored in polynomial time. This result is particularly surprising, given that Chaitin et al. [7] have shown that register allocation is NP-complete for general programs.

Most of the work on SSA-based register allocation remains purely theoretical. This paper presents what we believe is the first complete SSA-based register allocator. We describe the many phases that constitute the register allocation process, from the assignment of physical registers to variables to the generation of running code. We compare the proposed algorithm, and some of its variations,

[1] A regular program [6] is a program in SSA form with the additional property that each variable is defined before its first use.

against the state-of-the-art register allocator of LLVM [15], an industrial strength compiler. We have been able to compile and run a large range of benchmarks, such as `SPEC2000`, `MediaBench`, `FreeBench`, etc. In addition to the execution time of the compiled benchmarks, we give static metrics such as the number of loads and stores inserted by each allocator.

This paper also describes a collection of methods that we use in order to improve the quality of the code produced by our algorithm. These techniques include: (i - Sec. 4.5) heuristics for performing local and global register coalescing; (ii - Sec. 4.2) a transformation of the control flow graph that simplifies the register allocation process and (iii - Sec. 4.4) an aggressive static analysis that maps many spilled variables to the same memory address, thus reducing the number of loads, stores and the size of the stack frame in the generated code. Techniques (i) and (iii) can be added/removed from the algorithm in a very modular way, and, even without them, our implementation is very competitive with other register allocators.

2 Some Intuition on SSA-based Register Allocation

This section illustrates the simplicity and elegance of SSA-based allocation. The program in Figure 1 was taken from [17]. Figure 1 (b) is the same program, after the SSA elimination phase. Figure 1 (c) shows the sequence of assignments that would be performed by a traditional linear scan algorithm. This algorithm would traverse blocks 1, 2, 3 and finally block 4. When allocating registers in block 3, the allocator has to deal with temporaries `C` and `E`, which have already been assigned machine registers in block 2: a *self imposed conflict*. The graph in Figure 1 (c) cannot be colored with two colors. Its chromatic number is 3.

Figure 1 (d) outlines the dominance tree of our example program. Our allocator traverses the dominance tree in pre-order, assigning registers to temporaries. Because each variable in a SSA-form program is defined only once, self-imposed conflicts do not occur in SSA-based register allocation. That is, when the allocator first meets with the definition of a variable v, v cannot have been assigned a register. For a formal proof, see [14]. Figure 1 (e) shows the allocation process. The interference graph of the SSA-form program can be compiled with two registers: one register less than the minimum necessary in the program after SSA-elimination! Figure 1 (f) shows the final program. Notice that we use three `xor` instructions to transfer the values of `C1` and `E1` into `C` and `E`. This example is not a coincidence: the demand of registers in the SSA-form program is never greater than in the original program [13]. Furthermore, a greedy allocator that traverses the dominance tree in pre-order always finds an optimal coloring, if there is no aliasing of registers and no pre-colored registers.

2.1 A Note on the Representation of ϕ-functions

Following the notation established by Hack et al. [14], we represent the ϕ-functions existent in the beginning of a basic block as a matrix equation (see,

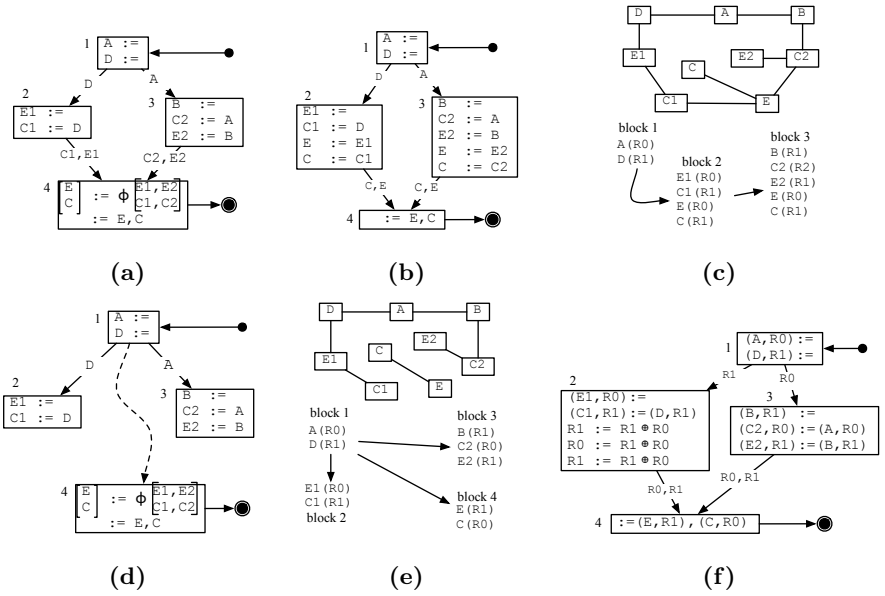

Fig. 1. (a) Example control flow graph in SSA-form [17]. (b) Program after the SSA elimination phase. (c) Interference graph and sequence of register assignments. (d) Dominance tree of the example program. (e) Interference graph of the SSA-form program, and assignment sequence. (f) Final program after SSA-based register allocation.

for instance, Figure 1 (a)). An equation such as $V = \phi M$, where V is a n-dimensional vector, and M is a $n \times m$ matrix, contains n ϕ-functions such as $a_i \leftarrow \phi(a_{i1}, a_{i2}, \ldots a_{im})$. Each possible execution path has a corresponding column in the ϕ-matrix, and adds one parameter to each ϕ-function. The ϕ symbol works as a multiplexer. It will assign to each element a_i of V an element a_{ij} of M, where j is determined by the actual path taken during the program's execution. This notation makes more explicit the fact that (i) ϕ-functions in a block are considered to execute in parallel and (ii) the use of two variables as arguments in a ϕ-function does not cause an interference between them.

3 Related Work

Many industrial compilers use the SSA form as an intermediate representation: Gcc 4.0 [12], Sun's HotSpot JVM [22], IBM's Jikes RVM [23] and LLVM [15]. However, these compilers do not perform register allocation on SSA-form programs: instead, they require a SSA elimination pass that replaces ϕ-functions with copy instructions before register allocation. The only SSA-based register allocator that has been described so far is a theoretical algorithm due to Hack et al. [14]. Some differences between our algorithm and Hack et al.'s are (i) the

global register coalescing heuristics, (ii) the handling of pre-colored variables, (iii) the spilling heuristics. However, the main difference is that we allow virtuals in ϕ-functions to be mapped to memory; what happens due to spilling. In Hack's case, if a parameter of a ϕ-function α is spilled, and it is not possible to re-load it in the basic block from where it flows into α, all other parameters and the variable defined by α must also be spilled. Our approach gives the compiler extra freedom to choose values to spill, but requires a more complex ϕ-deconstruction algorithm. Once a valid register assignment has been found, neither algorithm requires further spilling during the SSA-deconstruction phase.

The name *Linear Scan* designates a number of register allocation algorithms based on the work of Poletto and Sarkar [18]. The main objective of the original algorithm is to be fast. Subsequence versions attempt to produce code of better quality without seriously compromising the short compilation time. Traub et al.'s [24] and Wimmer et al.'s [25] versions handle holes in the live ranges of virtuals and split intervals to decrease spilling. Evlogimenos' [10] extensions coalesce move instructions and fold memory operands into instructions to save loads and stores. A key feature in this last work is the backtracking in face of spilling. Finally, the algorithm proposed by Sarkar et al. [20] gives another polynomial time exact solution to the register allocation problem. This technique does not rely on the SSA transformation to find an optimal register assignment. Instead, it uses copies and swaps to split the live ranges of variables whenever necessary. Sarkar's method and SSA-based register allocation require the same number of registers, which equals the size of the maximum number of live-ranges that cross any point of the control flow graph. The main difference between all these versions of linear scan and our SSA-based allocator is that, in the former case, register assignment occurs after ϕ-functions have been removed from the target code, whereas in the latter, it happens before.

The algorithm described by Cytron et al. [9] produces programs in *Conventional Single Static Assignment Form* (CSSA). In this flavor of SSA-form, the live ranges of virtuals that are part of the same ϕ-function do not overlap. If this is not the case, the program is said to be in *Transformed SSA* form (TSSA). The transformation techniques described in Sections 4.2 and 4.4 are similar to the analysis proposed by Sreedhar et al [21] to convert a program from TSSA to CSSA. In both cases copy instructions are used to split the live range of variables. The main difference is that Sreedhar et al.'s method incrementally inserts non-redundant copy instructions in the control flow graph, whereas we start with a completely partitioned program, and then proceed to the removal of redundant copy instructions until no redundancy remains. Furthermore, Sreedhar's transformation splits the live ranges of any variable v defined by a ϕ-function. In our case, this only happens if v is also used by some ϕ-function.

4 The Proposed Algorithm

Figure 2 outlines the main phases of our register allocation algorithm. Each of these phases is further detailed in this section. The grey boxes describe optional

extensions, which may improve the quality of the code produced, but may cause a degradation in compilation time. Figure 2 shows the SSA deconstruction phase preceding the insertion of spill code; however, swapping the order of these two phases is also possible.

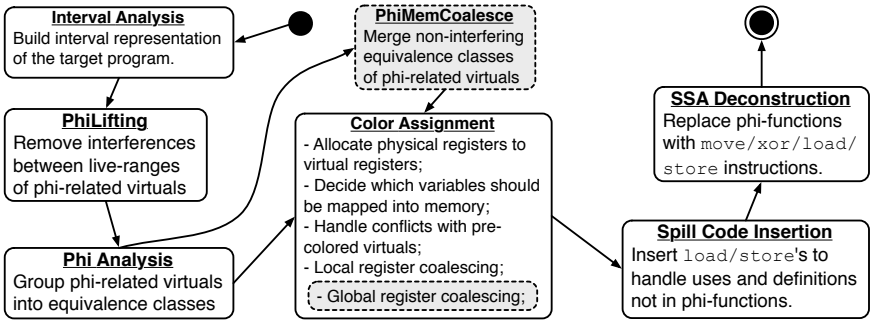

Fig. 2. Overview of the proposed algorithm. Grey boxes are optional extensions.

4.1 Interval analysis

This analysis is used in linear scan allocators in order to represent the live ranges of variables as a collection of ordered integer intervals. The interval representation allows to implement the algorithms in Sections 4.2 and 4.4 efficiently. Our interval representation ensures that ϕ-functions are treated as parallel copies. Given a ϕ-equation $V = \phi M$, all the virtuals in the vector V are marked as alive in the beginning of the basic block where they are defined. Let S be the set of virtuals from a column of the matrix M, and assume that these virtuals are fed into the ϕ-matrix from block B. All the virtuals in S are marked as alive at the end of B, but not beyond. For example, Figure 5 (d) shows the live ranges of variables alive in blocks 1 and 2 of the program in Figure 3 (c).

4.2 ϕ-Lifting: Static transformation of the control flow graph

Our algorithm allows the partial spilling of ϕ-functions, that is, some parameters of a ϕ-function can be mapped to memory, whereas others are located in registers. A question that must be addressed in this case is how to eliminate a ϕ-function whose definition v_d and at least one parameter v_u are stored in memory. A memory transfer would be necessary if the addresses of v_d and v_u were different. However, if these addresses are the same, no additional instruction is necessary: a store at the definition point of v_u provides also the correct value of v_d! In order to implement this variation of register coalescing, we ensure that if the

parameter and the definition of a ϕ-function are located in memory, then they are given the same memory address.

In order to facilitate our exposition, we define the equivalence class of ϕ-*related virtuals* [2] as follows: (i - reflexivity) any virtual v is ϕ-related to itself; (ii - symmetry) if virtual v is ϕ-related to virtual u, then u is ϕ-related to v; (iii - transitivity) if v_1 is ϕ-related to v_2, and v_2 is ϕ-related to v_3, then v_1 is ϕ-related to v_3. (iv) given a ϕ-function such as $v_i = \phi(v_{i1}, v_{i2}, \ldots, v_{im})$, the virtuals $v_i, v_{i1}, v_{i2}, \ldots, v_{im}$ are ϕ-related; Notice that the set of all the ϕ-related equivalence classes completely partition the set of virtuals in a SSA-form program. For instance, Figure 3 (a) shows an example control flow graph containing 6 ϕ-function, and three equivalence classes of ϕ-related virtuals: $\{v_1, v_3, v_4, v_5, v_6, v_7, v_9\}$, $\{v_2, v_8\}$ and $\{v_{10}, v_{11}, v_{12}, v_{13}, v_{14}\}$.

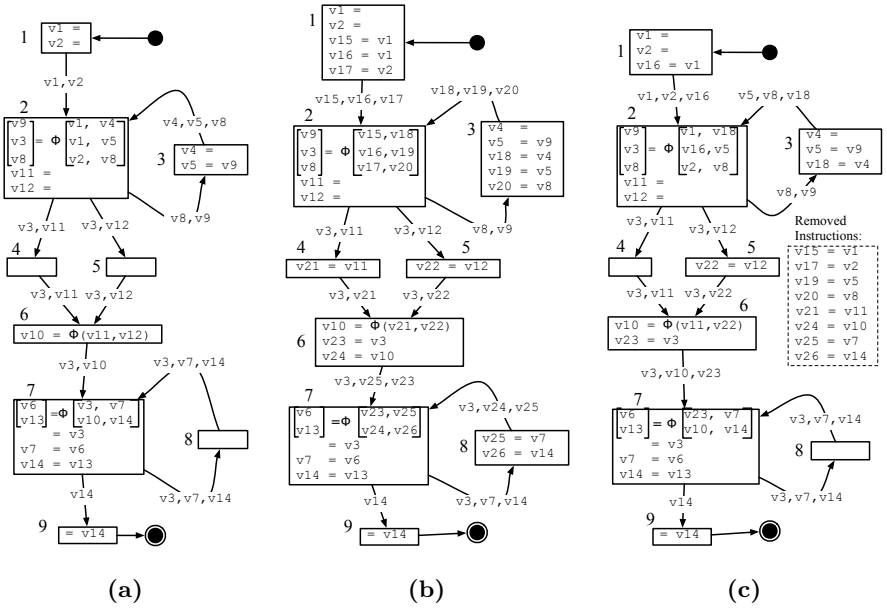

Fig. 3. (a) Control flow graph in TSSA-form. (b) Program transformed by **PhiLifting** (CSSA-form). (c) Program transformed by **PhiMemCoalesce** (CSSA-form).

Ideally, we would like to assign the same memory address to all the ϕ-related virtuals that have been spilled. However, this may introduce errors in a TSSA-form program, because the live ranges of ϕ-related virtuals might overlap. Interferences between the live ranges of ϕ-related virtuals are introduced by compiler optimizations such as global code motion [21] and copy folding during SSA construction [4]. For instance, assume that virtuals v_3 and v_6 have been spilled in

[2] ϕ-related equivalence classes are called ϕ-congruence classes in [21]

the program shown in Figure 3 (a). Although they are ϕ-related, it is not correct to store them in the same memory cell because their live ranges overlap in block 7. On the other hand, this problem does not exist if v_2 and v_8 are spilled, because they are never simultaneously alive. In order to convert the target program to CSSA-form, and so ensure that the live ranges of ϕ-related variables do not overlap, we resort to the **PhiLifting** [3] algorithm given in Figure 4.

PhiLifting
$(a_i = \phi(a_{i1} : B_1, a_{i2} : B_2, \ldots, a_{im} : B_m))$
  Create new virtual v_i
  Add copy instruction $I = \langle a_i := v_i \rangle$
  at the end of block B_j
  Replace a_i by v_i in ϕ
  For each virtual $a_{ij} \in \phi$,
    Create new virtual v_{ij}
    Add copy instruction $I = \langle v_{ij} := a_{ij} \rangle$
    at the end of block B_j
    Replace a_{ij} by v_{ij} in ϕ

PhiMemCoalesce($S_I = \{I_1, I_2, \ldots, I_q\}$,
$S_Q = \{Q_1, Q_2, \ldots, Q_r\}$)
  For each instruction $I = \langle v_{ij} := a_{ij} \rangle \in S_I$,
    $S_I := S_I \setminus \{I\}$
    Let Q_v be the equivalence class of v_{ij}
    Let Q_a be the equivalence class of a_{ij}
    If $Q_v \cap Q_v = \emptyset$
      $S_Q := S_Q \setminus \{Q_v\}$
      $S_Q := S_Q \setminus \{Q_a\}$
      $S_Q := S_Q \cup \{Q_a \cup Q_v\}$

Fig. 4. The algorithms **PhiLifting** and **PhiMemCoalesce**

.

Notice that if v_{ij} is a virtual created by the **PhiLifting** algorithm, then v_{ij} is alive only at the end of the block B_j that feeds it to its ϕ-function. Because each parameter of a ϕ-function comes from a different basic block, the control flow graph transformed by the **PhiLifting** algorithm has the following property: *if v_1 and v_2 are two ϕ-related virtuals, then their live ranges do not overlap.* The transformed control flow graph contains one ϕ-related equivalence class for each ϕ-function, and one equivalence class for each virtual v that does not participate in any ϕ-function. Following with our example, Figure 3 (b) outlines the result of applying **PhiLifting** on the control flow graph given in Figure 3 (b). The transformed program contains 14 equivalence classes: $\{v_1\}$, $\{v_2\}$, $\{v_4\}$, $\{v_5\}$, $\{v_7\}$, $\{v_{11}\}$, $\{v_{12}\}$, $\{v_{14}\}$, $\{v_9, v_{15}, v_{18}\}$, $\{v_3, v_{16}, v_{19}\}$, $\{v_8, v_{17}, v_{20}\}$, $\{v_{10}, v_{21}, v_{22}\}$, $\{v_6, v_{23}, v_{25}\}$ and $\{v_{13}, v_{24}, v_{26}\}$. It is important to notice that the extra variables created by **PhiLifting** do not increase the minimal number of registers necessary to compile the target program, as stated in Theorem 1 (For a proof, see this paper's companion technical report [16]).

Theorem 1. *Let P be a program whose control flow graph does not contain critical edges* [4]*.* **PhiLifting** *does not increase the register pressure in P.*

[3] occasionally we will denote ϕ-functions as $a_i = \phi(a_{i1} : B_1, a_{i2} : B_2, \ldots, a_{im} : B_m)$, meaning that variable a_{ij} comes from block B_j.

[4] A critical edge is defined as an edge between a block with multiple successors and a block with multiple predecessors [4].

4.3 ϕ-Analysis

The ϕ-analysis groups into equivalence classes the virtuals that are related by some ϕ-function. Because of the transformation performed by the **PhiLifiting** algorithm, the ϕ-analysis has a very efficient implementation: each ϕ-function $v = \phi(v_1, \ldots, v_m)$ already determines an equivalence class, e.g $\{v, v_1, \ldots, v_m\}$.

4.4 Aggressive Coalescing of ϕ-related virtuals

Although the **PhiLifting** algorithm produces correct programs, it is too conservative, because all the ϕ-related virtuals are used in the same ϕ-function. We now describe a coalescing technique that minimizes the number of ϕ-related equivalence classes while still keeping the target program in CSSA-form. In the algorithm **PhiMemCoalesce**, given in Figure 4, S_I is the set of instructions created by the procedure **PhiLifting**, and S_Q is the set of equivalence classes of the program transformed by **PhiLifting**.

In order to perform operations such as $Q_i \cap Q_j$ efficiently, we rely on the interval representation of live ranges commonly used in versions of the linear scan algorithm. Each virtual is represented as a collection of ordered intervals on the linearized control flow graph. Thus, a set Q of virtuals is a set of *ordered integer* intervals. In this way, the intersection of two ϕ-equivalence classes Q_i and Q_j can be found in time linear on the number of disjoint intervals in both sets. Because a ϕ-equivalence class can have $O(V)$ disjoint intervals, the final complexity of algorithm **PhiMemCoalesce** is $|L_i| \times V$.

Figure 3 (c) illustrates the application of algorithm **PhiMemCoalesce** on the program shown in Figure 3 (b). The procedure **PhiLifting** inserts 12 copy instructions into the target control flow graph. **PhiMemCoalesce** removes all but four of these instructions: $\langle v_{16} = v_1 \rangle$, $\langle v_{22} = v_{12} \rangle$, $\langle v_{18} = v_4 \rangle$ and $\langle v_{23} = v_3 \rangle$. If, for example, v_1 and v_{16} were coalesced, the interfering variables v_3 and v_9 would be placed in the same ϕ-equivalence class.

4.5 The Color Assignment Phase

Live ranges of variables in a SSA-form program are contiguous along any path on the dominance tree. For instance, Figure 5 (a) shows the dominance tree of the program given in Figure 3 (c). Figure 5 (b) outlines the live range of v_3 across the path formed by the blocks 2, 6 and 7, and Figure 5 (c) depicts the live range of v_{12} along the blocks 2 and 5. Due to this continuity, a single, non-iterative pass can be used to allocate physical registers to virtuals: the dominance tree is traversed in pre-order and registers are greedily assigned to live ranges. In the absence of pre-colored registers and aliasing this assignment is optimal [11].

Local Register coalescing Given a move instruction such as $v_d = v_u$, it is desirable that the same physical register be allocated to both v_d and v_u. In this case, the copy instruction can be removed without compromising the

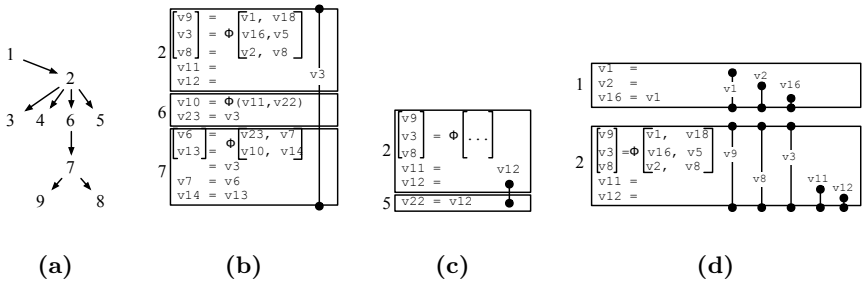

Fig. 5. (a) Dominance tree of program in Figure 3 (c). (b) Live range of v_3 along blocks 2, 6, 7. (c) Live range of v_{12} along blocks 2 and 5. (d) Live ranges of variables in blocks 1 and 2 of program in Figure 3 (c).

semantics of the program. We perform this assignment if v_u is not alive past the definition point of v_d. This type of coalescing is always safe, given Lemma 1 (for a proof, see [13]). Moreover, because there are no holes in the joint live range of v_u and v_d, it does not compromise the optimality of the color assignment algorithm. This simple optimization is important because it eliminates most of the redundant instructions added by the **PhiLifting** algorithm from Section 4.2, if the **PhiMemCoalesce** pass is not used.

Lemma 1. *Two virtuals v_1 and v_2 of a regular program interfere if, and only if, either v_1 is alive at the definition point of v_2, or vice-versa.*

Global Register coalescing It is desirable that the parameters and the definition of ϕ-functions be assigned the same physical register, because such assignments reduce the number of instructions necessary to eliminate the ϕ-functions. We say that a virtual v_{ij} is a *fixed point* in a ϕ-function such as $v_i = \phi(\ldots, v_{ij}, \ldots)$ if v_i and v_{ij} are assigned the same physical register during the coloring phase. The problem of maximizing the number of fixed points in a SSA-form program is NP-complete [14]. Thus, we use heuristics in order to increase the number of fixed points in the final register allocated code.

For each virtual that participates in a ϕ-function, we maintain a *preference list* of physical registers, which is computed by the procedure **ComputePreference**, given in Figure 6. In that figure, we let $l(v)$ be the location of v, which may be a physical register (r), a memory location (m), or an undefined value ($\perp$), if v has not yet been assigned a location by the register allocator. Before assigning a register to a virtual v, v's preference list is traversed in decreasing order, and the first available physical register is chosen.

The worst case complexity of building the preference list of v is $O(V + |\phi|)$, where $|\phi|$ is the number of ϕ-functions in the target program and V is the number of variables. In practice this complexity is $O(V)$, because each virtual tends to appear in a constant number of ϕ functions. The complexity of performing the

```
ComputePreference v
    For each physical register r
        p[r] := 0
    If v is defined by ⟨v = φ(v_1, ..., v_p)⟩,
      For each v_i ∈ φ
          if l(v_i) = r
              p[r] := p[r] + 1
    For each I = ⟨v_d = φ(.., v, ..)⟩
      if l(v_d) = r
          p[r] := p[r] + 1
    return p sorted in decreasing order.
```

```
ComputeWeigth v, s
    w := 0
    Let I be the definition site of v
        If I ≠ φ
          w := w + 1 + 10^L(I)
        Else let I = ⟨v := φ(v_1 : B_1, ..., v_p : B_p)⟩
            ∀i, 1 ≤ i ≤ p
                If v ≠ v_i
                  w := w + 1 + 10^L(B_i)
    ∀ I ∈ use sites of v
        If I ≠ φ
          w := w + 1 + 10^L(I)
        Else let I = ⟨v_aux := φ(..., v : B, ...)⟩
            If v_aux ≠ v
              w := w + 1 + 10^L(B)
    return w/s
```

Fig. 6. The algorithms **ComputePreference** and **ComputeWeight**.

ordering is $O(R \times \log R)$, where R denotes the number of physical registers in the target program. Given that the maximum size of each cell in the preference list is V, it can also be ordered in $O(V)$ using the bucket sort method [8].

The Spilling Heuristics If, at any point in the dominance tree, the number of live ranges is larger than the number of available physical registers, then some virtual must be spilled. The problem of minimizing the number of registers sent to memory in an SSA-form program is NP-complete [26]. We decide which variables should be spilled during the coloring phase based on a simple heuristics. We use the algorithm in Figure 6 to compute the *weight* of variables, where $L(I)$ is the loop nesting depth of the basic block that contains instruction I, and s is the size of v's live range, which is given by the number of instructions and control flow edges that it crosses. The smaller the weight of a variable, the bigger the chance that it will be spilled. The weight of a variable can change during the color assignment phase. As we show in Section 4.6, it is beneficial to deduct from the weight of a variable defined by a ϕ-function the contribution of the parameters that have been already spilled.

Handling Pre-colored Virtual Registers Constraints imposed by the computer architecture and the compiler may require that specific registers be allocated to some virtuals. These virtuals are called *pre-colored* or *fixed* registers. The polynomial solution to SSA-based register allocation does not support pre-colored registers. Indeed, a general pre-coloring of a chordal graph cannot be extended to a full coloring in polynomial time, unless $P = NP$ [1][5]. With the objective of keeping our implementation simple, we adopt the approach of [10, 25] and avoid assigning to a virtual v any physical register r if the live range of

[5] This problem is polynomial if each color appears at most once in the pre-coloring [1]

v overlaps the live range of any other virtual pre-colored with r. We can use the interval representation of v and r to efficiently determine whether the assignment is safe.

4.6 The SSA-deconstruction Phase

Traditional instruction sets do not implement ϕ-functions. Thus, once every virtual in the target program has been assigned a location l, which is either a physical register or a memory slot, ϕ-functions must be replaced by concrete instructions. A ϕ equation $V = \phi M$, where M is a $m \times n$ matrix, represents n parallel copies $(l(a_1), l(a_2), \ldots, l(a_m)) = \phi(l(a_{1j}), l(a_{2j}), \ldots, l(a_{mj}))$. The SSA-elimination problem amounts to how to sequentially transfer the values from each $l(a_{ij})$ to its counterpart $l(a_i)$ while preserving the semantics of the parallel copy. This is, in essence, a scheduling problem [4]: the copy $l(a_i) = l(a_{ij})$ can be scheduled safely if any other copy that uses $l(a_{ij})$ as a destiny has already been scheduled; we let this operation to be called a *safe copy*. Safe copies are described in the table below [6]. We let Q_j be the set $\{a_{1j}, a_{2j}, \ldots, a_{nj}\}$ of parameters coming from block B_j, and we let Q be the set $\{a_1, a_2, \ldots, a_n\}$ of variables defined by the ϕ-equation. After inserting a safe copy at the end of B_j to resolve the transference $l(a_i) = l(a_{ij})$, we remove a_i from Q and we remove a_{ij} from Q_j. The safe copies are inserted until no longer possible:

	$l(a_i)$	$l(a_{ij})$	pre-condition	Operations
1	r	m		`store r in m`
2	m	r	$\nexists v \in Q_j, l(v) = r$	`load r from m`
3	r_x	r_y	$r_x = r_y$	do nothing
4	r_x	r_y	$\nexists v \in Q_j, l(v) = r_x$	`rx := ry`
5	m	m		do nothing

If no safe copy can be inserted, the copy operations that remain to be destroyed constitute a perfect permutation [16]. Thus, in the final step of the algorithm, we need to swap locations to break cycles; safe copies are no longer inserted after this point. We have implemented swaps of integer registers as sequences of three `xor` instructions. A backup register must be used to swap floating point values, as proposed by Briggs et al. [4]. Figure 7 illustrates our SSA-elimination algorithm. Notice that there is a permutation of the physical registers `R2, R3, R4` between the definition vector and the column fed by block X. Such permutation cannot use safe copies, and must be eliminated by a sequence of register swaps. The transferring of values between block Y and block Z can be completely implemented via safe copies.

4.7 Spill Code Placement

During this phase, load and store instructions are inserted into the target code to handle spilled virtuals used or defined in instructions other than ϕ-functions.

[6] The 'do nothing' operation can be seen as an instance of register coalescing.

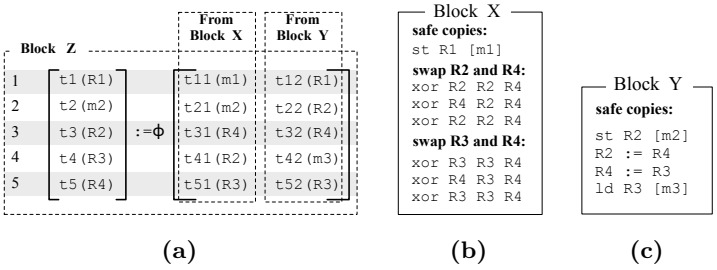

Fig. 7. (a) Five ϕ-functions joining two basic blocks. (b) Tail of block X, after ϕ-elimination. (c) Tail of block y, after ϕ-elimination.

Because of the single assignment property, only one store is necessary to preserve the definition of a spilled variable. We attempt to recycle loads among different uses of the same spilled register whenever possible. This optimization is only applicable inside a basic block. Notice that further loads and stores are inserted during the SSA-elimination phase due to ϕ-related virtuals that have been spilled. Nevertheless, this phase is completely independent of the SSA-elimination step, and these two stages can be executed in any order. We do not spare registers for the insertion of spill code, e.g loads and stores. Still during the coloring phase, if a variable v is spilled, each use of v is replaced by a fresh virtual v', which is mapped to the same memory location as v. The register allocated to v' will be the target of the load instruction.

5 Experimental Results

We have implemented the proposed algorithm in C++, using the LLVM [15] platform, version 1.8. Our implementation does not use LLVM's data structures directly. Instead, we dump the compilers' intermediate representation in our framework, process it, and map the results back into LLVM's back end. We opted for this strategy in order to facilitate the development and debugging of our allocator. Our entire framework has about 2,600 lines of uncommented code, including code responsible for implementing liveness analysis, phi-deconstruction, removal of critical edges and auxiliary data structures (lists, trees, etc). We have used two collection of benchmarks in our experiments. The first set of programs comes from SPEC-CPU 2000. The second set is the LLVM's test suite: a collection of 214 integer and floating-point applications that include benchmarks such as `Fhourstones, FreeBench, MallocBench, Prolangs-C, ptrdist, mediabench, CoyoteBench`, etc. The LLVM test suite has provided us with 1,172 .c/.cpp files and 590 .h files, comprising 641,708 lines of C code. The hardware used for the tests is a Dual 1.25 GHz powerPC G4 with 2MB L3 cache and 1.25 GB DDR SDRAM running Mac OS X version 10.4.8. We have compiled and run our benchmarks using seven different register allocators:

`(Simple)`: the naive algorithm, that spills every temporary. `(Local)`: performs local allocation, that is, it attempts to keep values in registers along the same basic block, and spills the variables alive at the block boundaries. `(LLVM)`: the LLVM default algorithm. This is a modern version of linear scan. Before the register assignment phase, it executes an aggressive coalescing pass. Holes in the live ranges of variables are filled with other virtuals whenever possible. One particular characteristic of this algorithm is that it backtracks in the presence of spills: if a variable v is spilled, the allocation restarts from the beginning of v's interval. This optimization avoids reserving registers to insert spill code. As a further optimization, the algorithm tries to use spilled values directly from memory whenever it is possible, in order to minimize the number of loads/stores inserted due to spilling. For a detailed description of the algorithm, see [10]. `(Ch -O1)` the plain SSA-based register allocator without any of the optional extensions shown in Figure 2. `(Ch -O2)` SSA-based register allocation with global register coalescing. `(Ch -O3)` SSA-based allocator with register coalescing and coalescing of ϕ-related virtuals in memory, as described in Section 4.4. `(GCC -O2)` The gcc compiler. We include gcc for reference purposes, given that it is a well know compiler, although it uses a back-end different than the other algorithms.

We do not compare compilation times, because our algorithm has not been directly implemented in LLVM, i.e, the dumping of data slows the compilation substantially. We point that the asymptotical complexity of `Ch -O1` is the same as one iteration of `LLVM's alg`. The coalescing heuristics in `Ch -O2` has increased the compilation time in about 11%, and the merging of equivalence classes in `Ch -O3` has added 16% to the compilation time of `Ch -O1`. In our experiments we vary the number of registers available in the target architecture. The PowerPC register bank contains 32 integer registers, but 3 are reserved by the PowerPC ABI (they are used for heap and stack addressing). Figure 8 gives the absolute running time of the 214 applications from the LLVM test suite when compiled with 4, 8 and 29 (the maximum number) general purpose registers. On the bottom of the figure we distinguish the applications whose running time, when compiled with our algorithm with 7 registers (four general purpose), was greater than 5 seconds. This happened for 33 applications, which count for about 94% of the total execution time. We also display the five largest absolute running times that we found among the benchmarks.

Table 1 shows the results obtained during the compilation of programs from the SPEC-CPU 2000 with 11 integer and 32 floating-point registers available. We present running time and static data, which consists of the number of variables spilled (`spill`), the maximum amount of space (in four bite words) reserved on the stack frame to stored spilled values (`stack`), plus the number of stores, loads, moves and xors present in the final assembly code. Notice that some of these instructions have not been inserted by the register allocator, but are part of the program itself. The algorithms used are `Ch -O1`, `Ch -O3` and LLVM's implementation. The rows labeled `Ratio` contain the execution time of the code produced with the LLVM's algorithm divided by the execution time produced by the different versions of our algorithm.

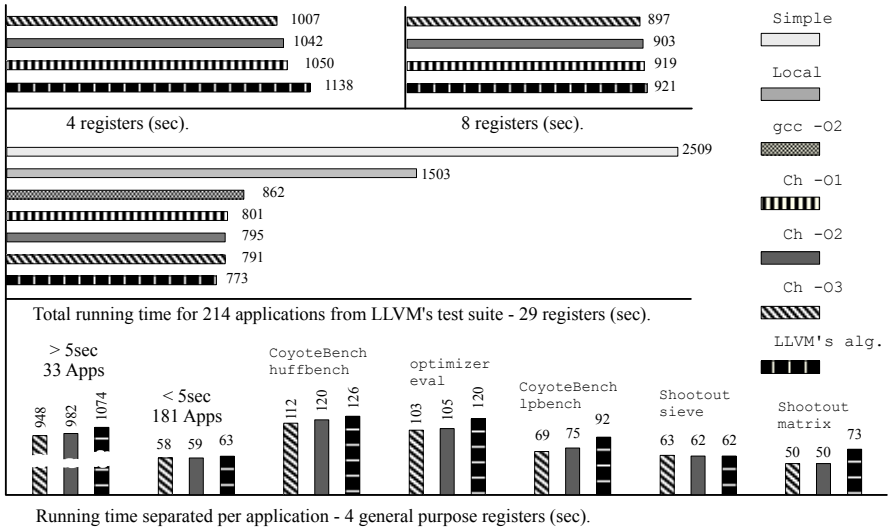

Fig. 8. Total execution time for 214 applications compiled with different register allcators and different number of general purpose registers.

Register allocation before SSA-elimination tends to produce lower number of spills. Consequently, the target code has fewer loads and stores, and requires less space on the memory stack. However, the splitting of live ranges causes the insertion of a large number of move and xor instructions in the produced assembly code. Because memory accesses are more expensive than copies, this tradeoff is advantageous in situations where the register pressure is high. Our academic implementation was able to outperform the industrial quality implementation of LLVM in most of the cases when the number of general purpose registers available was reduced to 8. Moreover, in the tests performed on the LLVM benchmarks with four general purpose registers, the performance gain of `Ch -O1` was approximately 9.5%, whereas the experiments in the SPEC2000 programs produced performance improvements as high as 38%, as in `bzip2`. Notice that in an architecture plenty of physical registers, spills seldom happen, yet the extra move/xor instructions remain as a burden in the SSA-based allocator. This fact suggests the existence of a *performance threshold* between register allocation before and after the elimination of ϕ-functions. Figure 9 compares our algorithm and the LLVM's implementation with four different numbers of general purpose registers: 4, 8, 12 and 16. Around 12 registers the fewer number of spills stops compensating for the large number of `move` and `xor` instructions that the SSA-based algorithm requires to keep the variables from been spilled.

		gzip	vpr	mcf	parser	bzip2	twolf	crafty	art	ammp	equake
1	Spill	247	1657	93	750	288	3326	3222	138	637	150
2	stack	84	394	100	96	202	385	421	104	74	169
3	Store	780	5274	364	2970	699	13479	6914	437	2252	362
4	Load	1173	12126	617	6363	1171	22333	15265	738	4746	782
5	Move	916	8118	257	4627	685	7158	6534	380	4046	482
6	Xor	356	1170	94	666	245	2245	2211	189	634	124
7	Time	89.93	375.38	541.87	41.79	453.12	16.27	44.46	330.97	18.20	17.51
8	Ratio	1.15	1.03	0.98	1.16	1.35	1.07	1.10	0.97	1.00	0.98
9	Spill	192	1507	73	671	246	2217	2647	124	613	146
10	stack	66	316	80	77	163	282	306	95	67	165
11	Store	698	4556	342	2874	637	7086	6185	400	2251	359
12	Load	1088	11410	595	6272	1109	15873	14513	700	4755	779
13	Move	808	7245	201	4229	600	5933	6006	352	3617	461
14	Xor	53	435	25	99	38	502	744	18	106	1
15	Time	88.28	374.87	537.94	41.12	443.81	16.04	45.06	305.77	18.04	17.02
16	Ratio	1.16	1.04	0.99	1.18	1.38	1.08	1.08	1.05	1.01	1.01
17	Spill	301	2241	120	990	369	3643	5260	197	717	252
18	stack	107	421	127	177	245	405	526	159	102	271
19	Store	855	5368	401	3315	802	8511	8673	507	2503	494
20	Load	1253	11146	622	6264	1312	16610	15996	733	4511	970
21	Move	219	3326	90	1575	180	2688	2190	164	1836	160
22	Xor	23	363	7	75	29	325	621	18	82	1
23	Time	103.13	388.05	533.27	48.71	611.15	17.36	48.83	322.28	18.15	17.18

Table 1. Static data collected for SPEC2000 with 11 registers (8 general purpose). (lines 1-8): `Ch -O1`. (lines 9-16): `Ch -O3`. (lines 17-23): `LLVM's alg`. `Time` given in secs, is the running time of the applications.

6 Conclusion

This paper has presented a SSA-based register allocator, which introduces novel approaches for global register coalescing, TSSA to CSSA transformation and SSA-elimination. Our experiments point that this algorithm is a good alternative for architectures with few registers, such as `x86`, or the `Thumb` subset of `ARM`, with eight general purpose registers. We are currently adapting our algorithm for JIT compilation. This version will be built directly into the LLVM core, and we expect that it will be as fast as the current algorithms available in that compiler. Further details about this project, and the implementation of our framework are available at `http://compilers.cs.ucla.edu/fernando/projects/soc/`.

References

1. M Biró, M Hujter, and Zs Tuza. Precoloring extension. i:interval graphs. In *Discrete Mathematics*, pages 267 – 279. ACM Press, 1992. Special volume (part 1) to mark the centennial of Julius Petersen's "Die theorie der regularen graphs".

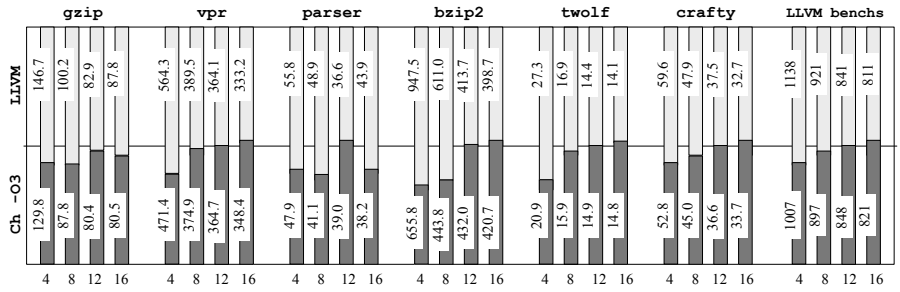

Fig. 9. Comparison between `Ch -O3` and LLVM's algorithm for architectures with 4, 8, 12 and 16 general purpose registers (plus three reserved registers). The numbers in the bars give the absolute execution time of the compiled applications (seconds).

2. Florent Bouchez. Allocation de registres et vidage en mémoire. Master's thesis, ENS Lyon, 2005.
3. Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, 1992.
4. Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software Practice and Experience*, 28(8):859–881, 1998.
5. Philip Brisk, Foad Dabiri, Roozbeh Jafari, and Majid Sarrafzadeh. Optimal register sharing for high-level synthesis of SSA form programs. *IEEE Transactions on Computer-Aided Design, Special Issue on IWLS*, 2006.
6. Zoran Budimlic, Keith D. Cooper, Timothy J. Harvey, Ken Kennedy, Timothy S. Oberg, and Steven W. Reeves. Fast copy coalescing and live-range identification. In *PLDI*, pages 25–32. ACM Press, 2002.
7. Gregory J. Chaitin, Mark A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.
8. Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Cliff Stein. *Introduction to Algorithms*. McGraw-Hill, 2nd edition, 2001.
9. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.
10. Alkis Evlogimenos. Improvements to linear scan register allocation. Technical report, University of Illinois, Urbana-Champaign, 2004.
11. Fanica Gavril. The intersection graphs of subtrees of a tree are exactly the chordal graphs. *Journal of Combinatoric*, B(16):46 – 56, 1974.
12. Brian J. Gough. *An Introduction to GCC*. Network Theory Ltd, 1st edition, 2005.
13. Sebastian Hack and Gerhard Goos. Optimal register allocation for SSA-form programs in polynomial time. *Information Processing Letters*, 98(4):150–155, 2006.
14. Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in SSA-form. In *CC*, pages 247–262. Springer-Verlag, 2006.

15. Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Symposium on Code Generation and Optimization*, pages 75–88, 2004.
16. Fernando Magno Quintao Pereira. A SSA-based register allocator. Technical report, University of California, Los Angeles, 2007.
17. Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In *APLAS*, pages 315–329. Springer, 2005.
18. Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):895–913, 1999.
19. B. K. Rosen, F. K. Zadeck, and M. N. Wegman. Global value numbers and redundant computations. In *POPL*, pages 12–27. ACM Press, 1988.
20. Vivek Sarkar and Rajkishore Barik. Extended linear scan: an alternate foundation for global register allocation. In *LCTES/CC*, pages X–X. ACM, 2007.
21. Vugranam C. Sreedhar, Roy Dz ching Ju, David M. Gillies, and Vatsa Santhanam. Translating out of static single assignment form. In *SAS*, pages 194–210. Springer-Verlag, 1999.
22. JVM Team. The java HotSpot virtual machine. Technical Report Technical White Paper, Sun Microsystems, 2006.
23. The Jikes Team. Jikes RVM home page, 2007. http://jikesrvm.sourceforge.net/.
24. Omri Traub, Glenn H. Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *Conference on Prgramming Language Design and Implementation (PLDI)*, pages 142–151, 1998.
25. Christian Wimmer and Hanspeter Mossenbock. Optimized interval splitting in a linear scan register allocator. In *VEE*, pages 132–141. ACM, 2005.
26. Mihalis Yannakakis and Fanica Gavril. The maximum k-colorable subgraph problem for chordal graphs. *Information Processing Letters*, 24(2):133 – 137, 1987.

A Lemmas and Theorems

Lemma 2. *Consider the ϕ-equation $V = \phi M$, where $V = (a_1, \ldots, a_n)$. Let the j^{th} column of M be $V_j = (a_{1j}, \ldots, a_{nj})$. We let B be the basic block where V is defined, and we let B_j be the basic block that feeds the j^{th} column of M. The register pressure at the end of B_j can never exceed the register pressure after the definition of the variables in V.*

Proof. Let $out(B_j)$ be the set of variables alive at the end of B_j. Let $S_j = \{a_{1j}, \ldots, a_{nj}\}$ be the set constituted by the elements of V_j, and let $T_j = \{t_1, \ldots, t_m\}$ be the set of variables that are alive after the definition of V, but that are not in V. We see that the set of variables alive at the end of block B_j is $out(B_j) = S_j \cup T_j$. Let $S = \{a_1, \ldots, a_n\}$. $|S| \geq |S_j|$ because $|V| = |V_j|$, and all the positions of V are filled with distinct elements, what is not necessarily the case in V_j. The set of variables alive past the definition of V is $A = S \cup T$, and by the definition of T, we have that $S \cap T = \emptyset$. We conclude that $A \geq out(B_j)$.

Theorem 2. *Let P be a program whose control flow graph does not contain critical edges.* **PhiLifting** *does not increase the global register pressure in P.*

Proof. This lemma amounts to saying that if P could be compiled with K registers before the control flow transformation, it still can be compiled with K registers after it. From Lemma 2 we know that the register pressure at the end of a basic block that feeds a ϕ-equation $V = \phi M$ is never greater than the register pressure after the definition of the virtuals in V. Because our target control flow graphs have no critical edges, **PhiLifting** only changes the register pressure at the end of basic blocks that feed ϕ-functions, which will be bounded by the register pressure past the definition point of those ϕ-functions.

Lemma 3. *If no safe operation applies, then there is no virtual a_i such that $l(a_i) = m$. .*

Proof. Immediate the from inspection of the table of preferences. □

Lemma 4. *If no safe copies can be used to transfer values from Q_j to Q, and if $l(a_i) \neq a_{ij}$, then there exists $a_x \in Q$ and $a_{yz} \in Q_j$ such that $l(a_x) = l(a_{ij})$ and $l(a_{yz}) = l(a_i)$.*

Proof. If there were $a_i \leftarrow a_{ij}$ such that $l(a_i)$ is not used by any virtual of Q_j, then either safe operation 2 or safe operation 4 would apply. From Lemma 3 we have that $\forall a_x \in Q, l(a_x) \neq m$. If $l(a_{ij})$ is not used by any virtual in Q, then there exists at least one register R used by some virtual from Q that is not used by any virtual from Q_j, and either safe operation 2 or 4 applies. □

Lemma 5. *If Q and Q_j share n pairs $l(a_i) := l(a_{ij}$, such that $l(a_i) \neq l(a_{ij})$, then the transference of values from Q_j to Q can be performed with at most n operations.*

Proof. The proof is by induction on n.

Basis: $n = 1$. In this case, there exists a virtual a_i in Q such that $l(a_i) = R_i$ and there exists a virtual a_{ij} such that $l(a_{ij}) = R_j$. Because the only miscolored variable in Q_j is a_{ij}, the register R_i cannot be in use by any other register of Q_j. Therefore, we can transfer the coloring with either safe operation 2 or 4.

Induction Hypothesis: the lemma is true for up to n miscolored registers.

Inductive Step: assume that there exists $n+1$ miscolored pairs of variables between Q and Q_j. Let $a_i \leftarrow a_{ij}$ be one of these pairs, so that $l(a_i) = R_i$ and $l(a_{ij}) = R_j$. If R_i is not in use by any miscolored variable of Q_j, we can perform the transfer via either a copy or a load instruction. Otherwise R_i is miscoloring another virtual a_{xy}, and we can transfer the coloring by means of a swap sequence. After the transfer of a_{ij}'s color, this virtual can be removed from the set of miscolored variables, and the result follows by the induction hypothesis. □