Register Allocation After Classical SSA Elimination is NP-Complete

Fernando Magno Quintão Pereira and Jens Palsberg

UCLA, University of California, Los Angeles

Abstract. Chaitin proved that register allocation is equivalent to graph coloring and hence NP-complete. Recently, Bouchez, Brisk, and Hack have proved independently that the interference graph of a program in static single assignment (SSA) form is chordal and therefore colorable in linear time. Can we use the result of Bouchez et al. to do register allocation in polynomial time by first transforming the program to SSA form, then performing register allocation, and finally doing the classical SSA elimination that replaces ϕ-functions with copy instructions? In this paper we show that the answer is no, unless P = NP: register allocation after classical SSA elimination is NP-complete. Chaitin's proof technique does not work for programs after classical SSA elimination; instead we use a reduction from the graph coloring problem for circular arc graphs.

1 Introduction

In Section 1.1 we define three central notions that we will use in the paper: the core register allocation problem, static single assignment (SSA) form, and post-SSA programs. In Section 1.2 we explain why recent results on programs in SSA form might lead one to speculate that we can solve the core register allocation problem in polynomial time. Finally, in Section 1.3 we outline our result that register allocation is NP-complete for post-SSA programs produced by the classical approach that replaces ϕ-functions with copy instructions.

1.1 Background

Register Allocation. In a compiler, register allocation is the problem of mapping temporaries to machine registers. In this paper we will focus on:

> **Core register allocation problem:**
> Instance: a program P and a number K of available registers.
> Problem: can each of the temporaries of P be mapped to one of the K registers such that temporary variables with interfering live ranges are assigned to different registers?

Notice that K is part of the input to the problem. Fixing K would correspond to the register allocation problem solved by a compiler for a fixed architecture. Our core register allocation problem is related to the kind of register allocation problem solved by gcc; the problem does not make assumptions about the number of registers in the target architecture.

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 79–93, 2006.

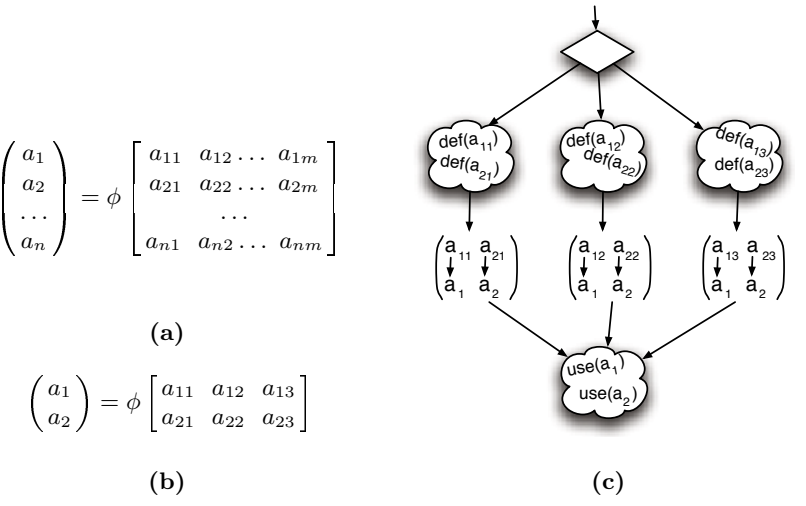


Fig. 1. (a) ϕ-functions represented as a matrix equation. (b) Matrix equation representing two ϕ-functions and three possible execution paths. (c) Control flow graph illustrating the semantics of ϕ-functions.

Chaitin et al. [8] showed that the core register allocation problem is NP-complete by a reduction from the graph coloring problem. The essence of Chaitin et al.'s proof is that every graph is the interference graph of some program.

SSA form. Static single assignment (SSA) form [21] is an intermediate representation used in many compilers, including gcc version 4. If a program is in SSA form, then every variable is assigned exactly once, and each use refers to exactly one definition. A compiler such as gcc version 4 translates a given program to SSA form and later to an executable form.

SSA form uses ϕ-functions to join the live ranges of different names that represent the same value. We will describe the syntax and semantics of ϕ-functions using the matrix notation introduced by Hack et al. [17]. Figure 1 (a) outlines the general representation of a ϕ-matrix. And Figure 1 (c) gives the intuitive semantics of the matrix shown in Figure 1 (b).

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 $a_i \leftarrow \phi(a_{i1}, a_{i2}, \dots a_{im})$. 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.

All the ϕ-functions are evaluated simultaneously at the beginning of the basic block where they are located. As noted by Hack et al. [17], the live ranges of temporaries in the same column of a ϕ-matrix overlap, while the live ranges of temporaries in the same row do not overlap. Therefore, we can allocate the same register to two temporaries in the same row. For example, Figure 2 shows a program, its SSA version, and the program after classical SSA elimination.

If the control flow reaches block 2 from block 1, $\phi(v_{11}, v_{12})$ will return v_{11}; v_{12} being returned otherwise. Variables i_2 and v_{11} do not interfere. In contrast, the variables v_{11} and i_1 interfere because both are alive at the end of block 1.

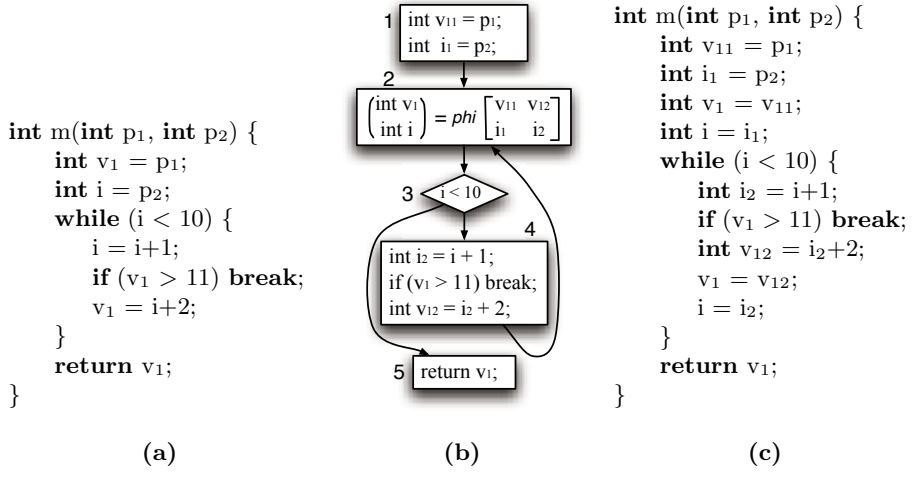


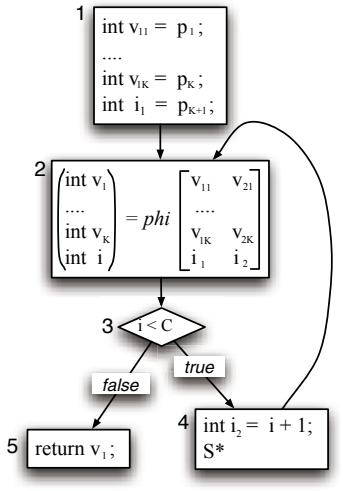
Fig. 2. (a) A program. (b) The same program in SSA form. (c) The program after classical SSA elimination.

Post-SSA programs. SSA form simplifies many analyses that are performed on the control flow graph of programs. However, traditional instruction sets do not implement ϕ-functions [10]. Thus, in order to generate executable code, compilers have a phase called *SSA-elimination* in which ϕ-functions are destroyed. Henceforth, we will refer to programs after SSA elimination as *post-SSA* programs.

The classical approach to SSA-elimination replaces the ϕ-functions with copy instructions [1, 5, 7, 10, 18, 20]. For example, consider $v_1 = \phi(v_{11}, ..., v_{1m})$ in block b. The algorithm explained by Appel [1] adds at the end of each block i that precedes b, one copy instruction such as $v_1 = v_{1i}$.

In this paper we concentrate on SSA programs whose control flow graphs have the structure outlined in Figure 3 (a). The equivalent post-SSA programs, generated by the classical approach to SSA-elimination, are given by the grammar in Figure 3 (b). We will say that a program generated by the grammar in Figure 3 (b) is a *simple* post-SSA program. For example, the program in Figure 2(c) is a simple post-SSA program. A simple post-SSA program contains a single loop. Just before the loop, and at the end of it (see Figure 3 (b)), the program contains copy instructions that correspond to the elimination of a ϕ-matrix such as:

$$\begin{pmatrix} v_1 \\ \cdots \\ v_K \\ i \end{pmatrix} = \phi \begin{bmatrix} v_{11} & v_{21} \\ & \cdots \\ v_{1K} & v_{2K} \\ i_1 & i_2 \end{bmatrix}$$



(a)

P ::= **int** m(**int** p_1, ..., **int** p_{K+1}) {
int $v_{11} = p_1$; ...; **int** $v_{1K} = p_K$;
int $i_1 = p_{K+1}$;
int $v_1 = v_{11}$; ...; **int** $v_K = v_{1K}$;
int $i = i_1$;
while ($i < \mathcal{C}$) {
int $i_2 = i+1$;
S^*
$v_1 = v_{21}$; ...; $v_K = v_{2K}$;
$i = i_2$;
}
return v_1;
}

S ::= **int** $v_j = i+\mathcal{C}$;
| $v_j = v_k$;
| **if** ($v_j > \mathcal{C}$) **break;**

$\mathcal{C}$ ranges over integer constants

(b)

Fig. 3. (a) Control flow representation of simple SSA programs. (b) The grammar for simple post-SSA programs.

1.2 Programs in SSA-Form Have Chordal Interference Graphs

The core register allocation problem is NP-complete and a compiler can transform a given program into SSA form in cubic time [9]. Thus we might expect that the core register allocation problem for programs in SSA form is NP-complete. However, that intuition would be wrong, unless P = NP, as demonstrated by the following result.

In 2005, Brisk et al. [6] proved that *strict* programs in SSA form have *perfect* interference graphs; independently, Bouchez [4] and Hack [16] proved the stronger result that strict programs in SSA form have *chordal* interference graphs. In a strict program, every path from the initial block to the use of a variable v passes through a definition of v [7]. The proofs presented in [4, 16] rely on two well-known facts: (i) the chordal graphs are the intersection graphs of subtrees in trees [14], and (ii) live ranges in an SSA program are subtrees of the dominance tree [16].

We can color a chordal graph in linear time [15] so we can solve the core register allocation problem for programs in SSA form in linear time. Thus, the transformation to SSA form seemingly maps an NP-complete problem to a polynomial-time problem in polynomial time! The key to understanding how such a transformation is possible lies in the following observation. Given a program P, its SSA-form version P', and a number of registers K, the core register allocation problem (P, K) is not equivalent to (P', K). While we can map a (P, K)-solution to a (P', K)-solution, we can not necessarily map a (P', K)-solution to a (P, K)-solution. The SSA transformation splits the live ranges of temporaries in P in such a way that P' may need fewer registers than P.

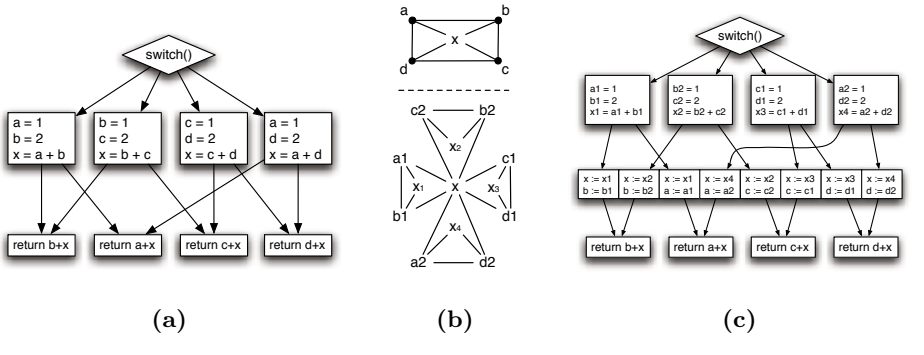
Given that the core register allocation problem for programs in SSA form can be solved in polynomial time and given that a compiler can do classical SSA elimination in linear time, we might expect that the core register allocation problem after classical SSA elimination is in polynomial time. In this paper we show that also that intuition would be wrong!

1.3 Our Result

We prove that the core register allocation problem for simple post-SSA programs is NP-complete. Our result has until now been a commonly-believed folk theorem without a published proof. The recent results on register allocation for programs in SSA form have increased the interest in a proof. Our result implies that the core register allocation problem for post-SSA programs is NP-complete for any language with loops or with jumps that can implement loops.

The proof technique used by Chaitin et al. [8] does not work for post-SSA programs. Chaitin et al.'s proof constructs programs from graphs, and if we transform those programs to SSA form and then post-SSA form, we can color the interference graph of each of the post-SSA programs with just three colors. For example, in order to represent C_4, their technique would generate the graph in the upper part of Figure 4 (b). The minimal coloring of such graph can be trivially mapped to a minimal coloring of C_4, by simply deleting node x. Figure 4 (a) shows the control flow graph of the program generated by Chaitin et al.'s proof technique, and Figure 4 (c) shows the program in post-SSA form. The interference graph of the transformed program is shown in the lower part of Figure 4 (b); that graph is chordal, as expected.

We prove our result using a reduction from the graph coloring problem for circular arc graphs, henceforth called simply *circular graphs*. A circular graph can be depicted as a set of intervals around a circle (e.g. Figure 5 (a)). The idea of our proof is that the live ranges of the variables in the loop in a simple post-SSA program form a circular graph. From a circular graph and an integer K we build a simple post-SSA program such that the graph is K-colorable if



(a) **(b)** **(c)**

Fig. 4. (a) Chaitin et al.'s program to represent C_4. (b) The interference graph of the original program (top) and of the program in SSA form (bottom). (c) The program of Chaitin et al. in SSA form.

and only if we can solve the core register allocation problem for the program and $K+1$ registers. Our reduction proceeds in two steps. In Section 2 we define the notion of SSA-circular graphs and we show that the coloring problem for SSA-circular graphs is NP-complete. In Section 3 we present a reduction from coloring of SSA-circular graphs to register allocation for simple post-SSA programs. An SSA-circular graph is a special case of a circular graph in which some of the intervals come in pairs that correspond to the copy instructions at the end of the loop in a simple post-SSA program. From a circular graph we build an SSA-circular graph by splitting some arcs. By adding new intervals at the end of the loop, we artificially increase the color pressure at that point, and ensure that two intervals that share an extreme point receive the same color. In Section 4 we give a brief survey of related work on complexity results for a variety of register allocation problems, and in Section 5 we conclude.

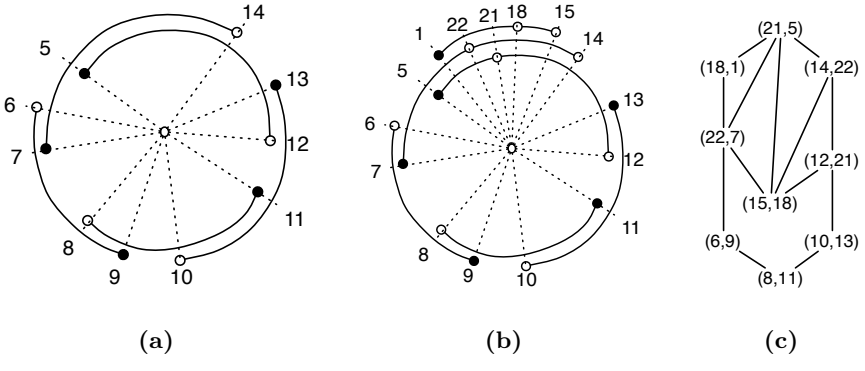


Fig. 5. (a) C_5 represented as a set of intervals. (b) The set of intervals that represent $W = \mathcal{F}(C_5, 3)$. (c) W represented as a graph.

Recently, Hack et al. [17] presented an SSA-elimination algorithm that does not use move instructions to replace ϕ-functions. Instead, Hack et al.'s algorithm uses xor instructions to permute the values of the parameters of the ϕ-functions in a way that preserves both the semantics of the original program and the chordal structure of the interference graph, without demanding extra registers. As a result, register allocation after the Hack et al.'s approach to SSA elimination is in polynomial time. In contrast, register allocation after the classical approach to SSA elimination is NP-complete.

2 From Circular Graphs to SSA-Circular Graphs

Let N denote the set of positive, natural numbers $\{1, 2, 3, \ldots\}$. A *circular graph* is an undirected graph given by a finite set of vertices $V \subseteq N \times N$, such that $\forall d \in N : (d, d) \notin V$ and $\forall (d, u), (d', u') \in V : d = d' \Leftrightarrow u = u'$. We sometimes

refer to a vertex (d, u) as an *interval*, and we call d, u *extreme points*. The set of vertices of a circular graph defines the edges implicitly, as follows. Define

$$b \ : \ N \times N \rightarrow \text{ finite unions of intervals of the real numbers}$$
$$b(d,u) = \begin{cases}]d,u[& \text{if } d < u \\ ]0,u[\cup]d,\infty[& \text{if } d > u. \end{cases}$$

Two vertices $(d, u), (d', u')$ are connected by an edge if and only if $b(d, u) \cap b(d', u') \neq \emptyset$. We use $\mathcal{V}$ to denote the set of such representations of circular graphs. We use $\max(V)$ to denote the largest number used in V, and we use $\min(V)$ to denote the smallest number used in V. We distinguish three subsets of vertices of a circular graph, V_l, V_i and V_z:

$$V_i = \{\ (d,u) \in V \ \mid \ d < u\ \}$$
$$V_l = \{\ (d,u) \in V \ \mid \ d > u\ \}$$
$$V_z = \{\ (d,y) \in V_i \ \mid \ \exists (y,u) \in V_l\ \}$$

Notice that $V = V_i \cup V_l$, $V_i \cap V_l = \emptyset$, and $V_z \subseteq V_i$.

Figure 5 (a) shows a representation of $C_5 = (\{a, b, c, d, e\}, \{ab, bc, cd, de, ea\})$ as a collection of intervals, where $a = (14, 7)$, $b = (6, 9)$, $c = (8, 11)$, $d = (10, 13)$, $e = (12, 5)$, $V_i = \{b, c, d\}$, $V_l = \{a, e\}$, and $V_z = \emptyset$. Intuitively, when the intervals of the circular graph are arranged around a circle, overlapping intervals determine edges between the corresponding vertices.

An SSA-circular graph W is a circular graph with two additional properties:

$$\forall (y,u) \in W_l : \exists d \in N : (d,y) \in W_z \tag{1}$$
$$\forall (d,u) \in W_i \setminus W_z : \forall (d',u') \in W_l : u < d' \tag{2}$$

We use $\mathcal{W}$ to denote the set of SSA-circular graphs.

Let W be an SSA-circular graph. Property (1) says that for each interval in W_l there is an interval in W_z so that these intervals share an extreme point y. In Section 3 it will be shown that the y points represent copy instructions used to propagate the parameters of ϕ-functions. Henceforth, the y points will be called *copy points*. Figure 5 (b) shows $W \in \mathcal{W}$ as an example of SSA-circular graph. $W_l = \{(18, 1), (21, 5), (22, 7)\}$, $W_i = \{(6, 9), (8, 11), (10, 13)\} \cup W_z$, and $W_z = \{(15, 18), (12, 21), (14, 22)\}$. Notice that for every interval $(y, u) \in W_l$, there is an interval $(d, y) \in W_z$. Figure 5 (c) exhibits W using the traditional representation of graphs.

Let $n = |V_l|$. We will now define a mapping $\mathcal{F}$ on pairs (V, K):

$$\mathcal{F} \ : \ \mathcal{V} \times N \rightarrow \mathcal{W}$$
$$\mathcal{F}(V,K) = V_i \cup \mathcal{G}(V_l, K, \max(V))$$
$$\mathcal{G} \ : \ \mathcal{V} \times N \times N \rightarrow \mathcal{V}$$
$$\mathcal{G}(\{\ (d_i,u_i) \ \mid \ i \in 1..n\ \}, K, m) = \{\ (m+i, m+K+i) \ \mid \ i \in 1..K-n\ \} \tag{3}$$
$$\cup \{\ (m+K+i, i) \ \mid \ i \in 1..K-n\ \} \tag{4}$$
$$\cup \{\ (d_i, m+2K+i) \ \mid \ i \in 1..n\ \} \tag{5}$$
$$\cup \{\ (m+2K+i, u_i) \ \mid \ i \in 1..n\ \} \tag{6}$$

Given V, the function $\mathcal{F}$ splits each interval of V_l into two nonadjacent intervals that share an extreme point: $(d, y) \in W_z$, and $(y, u) \in W_l$. We call those intervals the *main* vertices. Given V, the function $\mathcal{F}$ also creates $2(K - n)$ new intervals, namely $K - n$ pairs of intervals such that the two intervals of each pair are nonadjacent and share an extreme point: $(d, y) \in W_z$, and $(y, u) \in W_l$. We call those intervals the *auxiliary* vertices. Figures 5 (b) and 5 (c) represent $\mathcal{F}(C_5, 3)$, and Figure 6 outlines the critical points between $m = \max(V)$ and $K - n$.

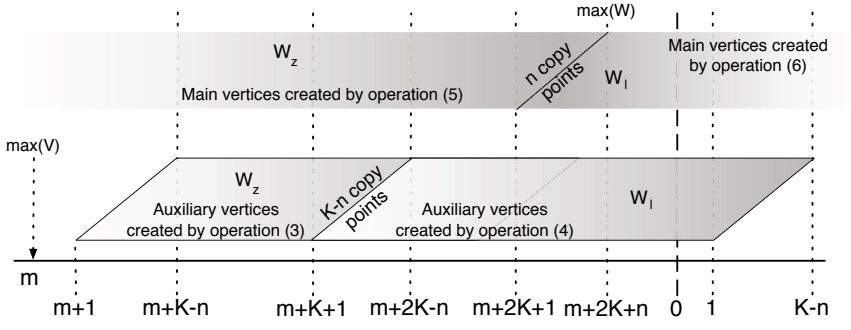


Fig. 6. The critical points created by $\mathcal{F}(V, K)$

Lemma 1. *If V is a circular graph and $\min(V) > K$, then $\mathcal{F}(V, K)$ is an SSA-circular graph.*

Proof. Let $W = \mathcal{F}(V, K)$. Notice first that W is a circular graph because the condition $\min(V) > K$ ensures that rules (4) and (6) define vertices that don't share any extreme points. To see that W is an SSA-circular graph let us consider in turn the two conditions (1) and (2). Regarding condition (1), W_l consists of the sets defined by (4), (6), while W_z consists of the sets defined by (3), (5), and for each $(y, u) \in W_l$, we can find $(d, y) \in W_z$. Regarding condition (2), we have that if $(d, u) \in W_i \setminus W_z$ and $(d', u') \in W_l$, then $u \leq \max(V) < \max(V) + K + 1 \leq d'$. □

Lemma 2. *If $W = \mathcal{F}(V, K)$ is K-colorable, then two intervals in $W_z \cup W_l$ that share an extreme point must be assigned the same color by any K-coloring of W.*

Proof. Let c be a K-coloring of W. Let $v_1 = (d, y)$ and $v_2 = (y, u)$ be a pair of intervals in $W_z \cup W_l$. From the definition of $\mathcal{F}(V, K)$ we have that those two intervals are not connected by an edge. The common copy point y is crossed by exactly $K - 1$ intervals (see Figure 6). Each of those $K - 1$ intervals must be assigned a different color by c so there remains just one color that c can assign to v_1 and v_2. Therefore, c has assigned the same color to v_1 and v_2. □

Lemma 3. *Suppose V is a circular graph and $\min(V) > K$. We have V is K-colorable if and only if $\mathcal{F}(V, K)$ is K-colorable.*

Proof. Let $V_l = \{\ (d_i, u_i)\ \mid\ i \in 1..n\ \}$ and let $m = \max(V)$.

First, suppose c is a K-coloring of V. The vertices of V_l form a clique so c must use $|V_l|$ colors to color V_l. Let $n = |V_l|$. Let $\{x_1, \ldots, x_{K-n}\}$ be the set of colors *not* used by c to color V_l. We now define a K-coloring c' of $\mathcal{F}(V, K)$:

$$c'(v) = \begin{cases} c(v) & \text{if } v \in V_i \\ x_i & \text{if } v = (m+i, m+K+i), i \in 1..K-n \\ x_i & \text{if } v = (m+K+i, i), i \in 1..K-n \\ c(d_i, u_i) & \text{if } v = (d_i, m+2K+i), i \in 1..n \\ c(d_i, u_i) & \text{if } v = (m+2K+i, u_i), i \in 1..n \end{cases}$$

To see that c' is indeed a K-coloring of $\mathcal{F}(V, K)$, first notice that the colors of the main vertices don't overlap with the colors of the auxiliary vertices. Second, notice that since $\min(V) > K$, no auxiliary edge is connected to a vertex in V_i. Third, notice that since c is a K-coloring of V, the main vertices have colors that don't conflict with their neighbors in V_i.

Conversely, suppose c' is a K-coloring of $\mathcal{F}(V, K)$. We now define a K-coloring c of V:

$$c(v) = \begin{cases} c'(v) & \text{if } v \in V_i \\ c'(d_i, m+2K+i) & \text{if } v = (d_i, u_i), i \in 1..n \end{cases}$$

To see that c is indeed a K-coloring of V, notice that from Lemma 2 we have that c' assigns the same color to the intervals $(d_i, m+2K+i), (m+2K+i, u_i)$ for each $i \in 1..n$. So, since c' is a K-coloring of $\mathcal{F}(V, K)$, the vertices in V_l have colors that don't conflict with their neighbors in V_i. □

Lemma 4. *Graph coloring for SSA-circular graphs is NP-complete.*

Proof. First notice that graph coloring for SSA-circular graphs is in NP because we can verify any color assignment in polynomial time. To prove NP-hardness, we will do a reduction from graph coloring for circular graphs, which is known to be NP-complete [12, 19]. Given a graph coloring problem instance (V, K) where V is a circular graph, we first transform V into an isomorphic graph V' by adding K to all the integers used in V. Notice that $\min(V') > K$. Next we produce the graph coloring problem instance $(\mathcal{F}(V', K), K)$, and, by Lemma 3, V' is K-colorable if and only if $\mathcal{F}(V', K)$ is K-colorable. □

3 From SSA-Circular Graphs to Post-SSA Programs

We now present a reduction from coloring of SSA-circular graphs to the core register allocation problem for simple post-SSA programs. In this section we use a representation of circular graphs which is different from the one used in Section 2. We represent a circular graph by a finite list of elements of the set $\mathcal{I} = \{\ \text{def}(j),\ \text{use}(j),\ \text{copy}(j,j'),\ \mid j, j' \in N\ \}$. Each j represents a temporary name in a program. We use ℓ to range over finite lists over $\mathcal{I}$. If ℓ is a finite list over $\mathcal{I}$ and the d-th element of ℓ is either $\text{def}(j)$ or $\text{copy}(j,j')$, then we say that

j is *defined* at index d of ℓ. Similarly, if the u'th element of ℓ is either use(j') or copy(j,j'), then we say that j' is *used* at index u of ℓ. We define $\mathcal{X}$ as follows:

$$\mathcal{X} = \{\ \ell \mid \text{for every } j \text{ mentioned in } \ell,\ j \text{ is defined exactly once and used exactly once} \wedge \text{for every copy}(j,j') \text{ in } \ell, \text{ we have } j \neq j'\ \}$$

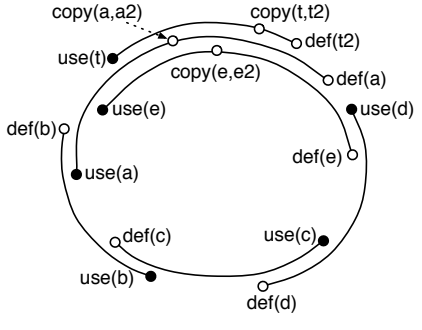
We will use X to range over $\mathcal{X}$. The sets $\mathcal{X}$ and $\mathcal{V}$ are isomorphic; the function α is an isomorphism which maps $\mathcal{X}$ to $\mathcal{V}$:

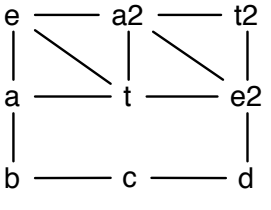
$$\alpha\ :\ \mathcal{X} \rightarrow \mathcal{V}$$
$$\alpha(X) = \{\ (d,u) \mid \exists j \in N : j \text{ is defined at index } d \text{ of } X \text{ and } j \text{ is used at index } u \text{ of } X\ \}$$

We define $\mathcal{Y} = \alpha^{-1}(\mathcal{W})$, and we use Y to range over $\mathcal{Y}$. The graph W shown in Figure 5 (b) is shown again in Figure 7 (a) in the new representation:

$$Y = \langle \text{use}(t), \text{use}(e), \text{def}(b), \text{use}(a), \text{def}(c), \text{use}(b), \text{def}(d), \text{use}(c), \text{def}(e), \text{use}(d), \text{def}(a), \text{def}(t_2), \text{copy}(t,t_2), \text{copy}(e,e_2), \text{copy}(a,a_2)\rangle.$$

Figure 8 presents a mapping $\mathcal{H}$ from $\mathcal{Y}$-representations of SSA-circular graphs to simple post-SSA programs. Given an interval (d,u) represented by $def(j)$ and





(a)

```
int m(int p1, int p2, int p3, int p4) {
  int i1 = p4;
  int a1 = p1;
  int e1 = p2;
  int t1 = p3;
  int a = a1;
  int e = e1;
  int t = t1;
  int i = i1;
  while(i > 10) {
    int i2 = i + 1;
    if(t > 9) break;
    if(e > 10) break;
    int b = i2 + 11;
    if (a > 11) break;
    int c = i2 + 12;
    if(b > 12) break;
    int d = i2 + 13;
    if(c > 13) break;
    int e2 = i2 + 14;
    if(d > 14) break;
    int a2 = i2 + 15;
    int t2 = i2 + 16;
    i = i2;
    a = a2;
    t = t2;
    e = e2;
  }
  return a;
}
```

(b)

Fig. 7. (a) $Y = \mathcal{F}(C_5, 3)$ represented as a sequence of instructions and as a graph. (b) $P = \mathcal{H}(Y, 3)$.

$$
\begin{aligned}
\text{gen}(\text{def}(j)) &= \textbf{int } \mathrm{v}_j = \mathrm{i}_2 + \mathcal{C};\\
\text{gen}(\text{use}(j)) &= \textbf{if } (\mathrm{v}_j > \mathcal{C}) \textbf{ break;}\\
\text{gen}(\text{copy}(j,j')) &= \mathrm{v}_j = \mathrm{v}_{j'};\\
\mathcal{H} &: \mathcal{Y} \times N \rightarrow \text{simple post-SSA program}\\
\mathcal{H}(Y,K) &= \textbf{int } \mathrm{m}(\textbf{int } \mathrm{p}_1, \ldots, \textbf{int } \mathrm{p}_{K+1}) \{\\
&\qquad \textbf{int } \mathrm{v}_{11} = \mathrm{p}_1; \ldots; \textbf{int } \mathrm{v}_{1K} = \mathrm{p}_K;\\
&\qquad \textbf{int } \mathrm{i}_1 = \mathrm{p}_{K+1};\\
&\qquad \textbf{int } \mathrm{v}_1 = \mathrm{v}_{11}; \ldots; \textbf{int } \mathrm{v}_K = \mathrm{v}_{1K};\\
&\qquad \textbf{int } \mathrm{i} = \mathrm{i}1;\\
&\qquad \textbf{while } (\mathrm{i} < \mathcal{C}) \{\\
&\qquad\quad \textbf{int } \mathrm{i}_2 = \mathrm{i}+1;\\
&\qquad\quad \mathrm{map}(Y, \mathrm{gen})\\
&\qquad\quad \mathrm{i} = \mathrm{i}_2;\\
&\qquad \}\\
&\qquad \textbf{return } \mathrm{v}_1;\\
&\}
\end{aligned}
$$

Fig. 8. The mapping of circular graphs to simple post-SSA programs

$use(j)$, we map the initial point d to a variable definition $\mathbf{v_j} = \mathbf{i_2} + \mathcal{C}$, where $\mathbf{i_2}$ is the variable that controls the loop. We assume that all the constants are chosen to be different. The final point u is mapped to a variable use, which we implement by means of the conditional statement **if** $(\mathbf{v_j} > \mathcal{C})$ **break**. We opted for mapping uses to conditional commands because they do not change the live ranges inside the loop, and their compilation do not add extra registers to the final code. An element of the form $copy(j, j')$, which is mapped to the assignment $j = j'$, is used to simulate the copy of one of the parameters of a ϕ-function, after classical SSA elimination. Figure 7 (b) shows the program $P = \mathcal{H}(Y, 3)$, where $Y = \mathcal{F}(C_5, 3)$.

Lemma 5. *We can color an SSA-circular graph Y with K colors if and only if we can solve the core register allocation problem for $\mathcal{H}(Y, K)$ and $K+1$ registers.*

Proof. First, assume Y has a K-coloring. The intervals in Y match the live ranges in the loop of $\mathcal{H}(Y, K)$, except for the control variables i, and i_2, which have nonoverlapping live ranges. Therefore, the interference graph for the loop can be colored with $K+1$ colors. The live ranges of the variables declared outside the loop form an interval graph of width $K+1$. We can extend the $K+1$-coloring of that interval graph to a $K+1$-coloring of the entire graph in linear time.

Now, assume that there is a solution of the core register allocation problem for $\mathcal{H}(Y, K)$ that uses $K+1$ registers. The intervals in Y represent the live ranges of the variables in the loop. The control variables i and i_2 demand one register, which cannot be used in the allocation of the other live ranges inside the loop. Therefore, the coloring of $\mathcal{H}(Y, K)$ can be mapped trivially to the nodes of Y. □

Theorem 1. *The core register allocation problem for simple post-SSA programs is NP-complete.*

Proof. Combine Lemmas 4 and 5. □

As an illustrative example, to color C_5 with three colors is equivalent to determining a 3-coloring to the graph Y in Figure 7 (a). Such colorings can be found if and only if the core register allocation problem for $P = \mathcal{H}(Y, 3)$ can be solved with 4 registers. In this example, a solution exists. One assignment of registers would be $\{a, a_1, a_2, c, p_1\} \rightarrow R1$, $\{b, d, t_1, t_2, t, p_3\} \rightarrow R2$, $\{e, e_1, e_2, p2\} \rightarrow R3$, and $\{i, i_1, i_2, p_4\} \rightarrow R4$. This corresponds to coloring the arcs a and c with the first color, arcs b and d with the second, and e with the third.

4 Related Work

The first NP-completeness proof of a register allocation related problem was published by Sethi [22]. Sethi showed that, given a program represented as a set of instructions in a directed acyclic graph and an integer K, it is an NP-complete problem to determine if there is a computation of the DAG that uses at most K registers. Essentially, Sethi proved that the placement of loads and stores during the generation of code for a straight line program is an NP-complete problem if the order in which instructions appear in the target code is not fixed.

Much of the literature concerning complexity results for register allocation deals with two basic questions. The first is the core register allocation problem, which we defined in Section 1. The second is the core spilling problem which generalizes the core register allocation problem:

> **Core spilling problem:**
> Instance: a program P, number K of available registers, and a number M of temporaries.
> Problem: can at least M of the temporaries of P be mapped to one of the K registers such that temporary variables with interfering live ranges are assigned to different registers?

Farach and Liberatore [11] proved that the core spilling problem is NP-complete even for straight line code and even if rescheduling of instructions is not allowed. Their proof uses a reduction from set covering.

For a straight line program, the core register allocation problem is linear in the size of the interference graph. However, if the straight line program contains pre-colored registers that can appear more than once, then the core register allocation problem is NP-complete. In this case, register allocation is equivalent to pre-coloring extensions of interval graphs, which is NP-complete [2].

In the core register allocation problem, the number of registers K is not fixed. Indeed, the problem used in our reduction, namely the coloring of circular graphs, has a polynomial-time solution if the number of colors is fixed. Given n circular arcs determining a graph G, and a fixed number K of colors, Garey et al. [12] have given an $O(n \cdot K! \cdot K \cdot \log K)$ time algorithm for coloring G if such a coloring exists. Regarding general graphs, the coloring problem is NP-complete for every fixed value of $K > 2$ [13].

Bodlaender et al. [3] presented a linear-time algorithm for the core register allocation problem with a fixed number of registers for structured programs.

Their result holds even if rescheduling of instructions is allowed. If registers of different types are allowed, such as integer registers and floating point registers, for example, then the problem is no longer linear, although it is still polynomial.

Researchers have proposed different algorithms for inserting copy instructions, particularly for reducing the number of copy instructions [7, 5, 10, 18]. Rastello et al. [10] have proved that the optimum replacement of ϕ-functions by copy instructions is NP-complete. Their proof uses a reduction from the maximum independent set problem.

5 Conclusion

We have proved that the core register allocation problem is NP-complete for post-SSA programs generated by the classical approach to SSA-elimination that replaces ϕ-functions with copy instructions. In contrast, Hack et al.'s recent approach to SSA-elimination [17] generates programs for which the core register allocation problem is in polynomial time. We conclude that the choice of SSA-elimination algorithm matters.

We claim that compiler optimizations such as copy propagation and constant propagation cannot improve the complexity of the core register allocation problem for simple post-SSA programs. Inspecting the code in Figure 8 we perceive that the number of loop iterations cannot be easily predicted by a local analysis because all the control variables are given as function parameters. In the statement **int** $\mathrm{v}_j = \mathrm{i}+\mathcal{C}$; the variable i limits the effect of constant propagation and the use of different constants $\mathcal{C}$ limits the effect of copy propagation. Because all the $K+1$ variables alive at the end of the loop have different values, live ranges cannot be merged at that point. In contrast, rescheduling of instructions might improve the complexity of the core register allocation problem for simple post-SSA programs. However, rescheduling combined with register allocation is an NP-complete problem even for straight line programs [22].

Theorem 1 continues to hold independent on the ordering in which copy instructions are inserted, because the function $\mathcal{G}$, defined in Section 2, can be modified to accommodate any ordering of the copy points. In more detail, let $W = \mathcal{F}(V, K)$ be a SSA-circular graph, let $n \in [0 \cdots \max(W)]$, and let $\mathrm{ovl}(n)$ be the number of intervals that overlap at point n.

$$\forall n \in \ ]max(V) \cdots max(W)] : \mathrm{ovl}(n) = K \tag{7}$$

Any ordering that ensures property 7 suffices for the proof of Lemma 2. Figure 6 shows the region around the point 0 of a SSA-circular graph. Given $W = \mathcal{F}(V, K)$, exactly K copy points are inserted in the interval between $\max(V)$ and $\max(W)$.

Our proof is based on a reduction from the coloring of circular graphs. We proved our result for programs with a loop because the core of the interference graph of such programs is a circular graph. The existence of a back edge in the control flow graph is not a requirement for Theorem 1 to be true. For example, SSA-circular graphs can be obtained from a language with a single **if**-statement.

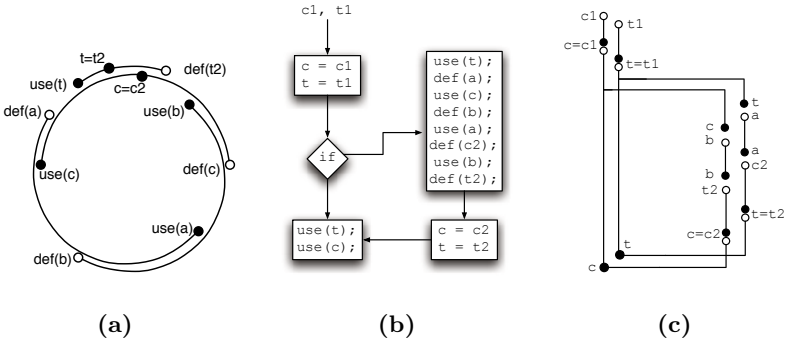


Fig. 9. (a) SSA graph W that represents $\mathcal{F}(C_3, K)$. (b) A program P that represents W with a single **if**-statement. (c) Schematic view of the live ranges of P.

Figure 9 (a) shows a SSA-circular graph that represents C_3, when $K = 2$, and Figure 9 (b) shows a program whose live ranges represent such graph. The live ranges are outlined in Figure 9 (c).

Acknowledgments. We thank Fabrice Rastello, Christian Grothoff and the anonymous referees for helpful comments on a draft of the paper. Fernando Pereira is sponsored by the Brazilian Ministry of Education under grant number 218603-9. Jens Palsberg is supported by the National Science Foundation award number 0401691.

References

1. Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition, 2002.
2. 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".
3. Hans Bodlaender, Jens Gustedt, and Jan Arne Telle. Linear-time register allocation for a fixed number of registers. In *SIAM Symposium on Discrete Algorithms*, pages 574–583, 1998.
4. Florent Bouchez. Allocation de registres et vidage en mémoire. Master's thesis, ENS Lyon, 2005.
5. 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.
6. Philip Brisk, Foad Dabiri, Jamie Macbeth, and Majid Sarrafzadeh. Polynomial-time graph coloring register allocation. In *14th International Workshop on Logic and Synthesis*. ACM Press, 2005.
7. 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 *International Conference on Programming Languages Design and Implementation*, pages 25–32. ACM Press, 2002.

8. 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.
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. François de Ferriére, Christophe Guillon, and Fabrice Rastello. Optimizing the translation out-of-SSA with renaming constraints. *ST Journal of Research Processor Architecture and Compilation for Embedded Systems*, 1(2):81–96, 2004.
11. Martin Farach and Vincenzo Liberatore. On local register allocation. In *9th ACM-SIAM symposium on Discrete Algorithms*, pages 564 – 573. ACM Press, 1998.
12. M. R. Garey, D. S. Johnson, G. L. Miller, and C. H. Papadimitriou. The complexity of coloring circular arcs and chords. *SIAM J. Algebraic Discrete Methods*, 1(2): 216–227, 1980.
13. M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete problems. *Theoretical Computer Science*, 1(3):193–267, 1976.
14. Fanica Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SICOMP*, 1(2):180–187, 1972.
15. Fanica Gavril. The intersection graphs of subtrees of a tree are exactly the chordal graphs. *Journal of Combinatoric*, B(16):46–56, 1974.
16. Sebastian Hack. Interference graphs of programs in SSA-form. Technical Report ISSN 1432-7864, Universitat Karlsruhe, 2005.
17. Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in SSA-form. In *15th International Conference on Compiler Construction*. Springer-Verlag, 2006.
18. Allen Leung and Lal George. Static single assignment form for machine code. In *Conference on Programming Language Design and Implementation*, pages 204–214. ACM Press, 1999.
19. Daniel Marx. A short proof of the NP-completeness of circular arc coloring, 2003.
20. Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In *Proceedings of APLAS'05, Asian Symposium on Programming Languages and Systems*, pages 315–329, 2005.
21. B. K. Rosen, F. K. Zadeck, and M. N. Wegman. Global value numbers and redundant computations. In *ACM SIGPLAN-SIGACT symposium on Principles of Programming languages*, pages 12–27. ACM Press, 1988.
22. Ravi Sethi. Complete register allocation problems. In *5th annual ACM symposium on Theory of computing*, pages 182–195. ACM Press, 1973.