Available online at www.sciencedirect.com

SCIENCE @ DIRECT®

ELSEVIER

Information Processing Letters 98 (2006) 150–155

Information Processing Letters

www.elsevier.com/locate/ipl

Optimal register allocation for SSA-form programs in polynomial time

Sebastian Hack [*], Gerhard Goos

Institut für Programmstrukturen und Datenorganisation, Adenauerring 20a, 76131 Karlsruhe, Germany

Received 11 February 2005; received in revised form 23 November 2005

Available online 17 February 2006

Communicated by F. Meyer auf der Heide

Abstract

This paper gives a constructive proof that the register allocation problem for a uniform register set is solvable in polynomial time for SSA-form programs.

Keywords: Combinatorial problems; Graph algorithms

1. Introduction

Register allocation is the task in a compiler, which maps (temporary) variables to processor registers. The most prominent approach is to map this task to a graph coloring problem. The nodes of the so-called *interference graph* are formed by the temporaries of the program. Whenever the compiler finds out, that two temporaries cannot be held in the same register (due to *interference*), an edge is drawn between the corresponding nodes in the interference graph. Colors correspond to processor registers. Thus, having k registers, a k-coloring of the interference graph forms a correct register assignment.

Chaitin et al. [4] show that for each undirected graph a program can be given which has this graph as its interference graph. So, since graph coloring is NP-complete, register allocation must also be. However, if one only considers programs in SSA-form (e.g., refer to [5]) the situation changes. It turns out, that interference graphs of SSA-form programs belong to the class of *chordal* graphs, which is in turn a subset of the class of *perfect* graphs. It is well known, that chordal graphs can be colored in quadratic time. This also answers the question posed by Andersson [1] whether interference graphs are perfect.

This paper is structured as follows: First, we describe the model of a program used in this paper. Then we give a new definition of liveness for SSA-form programs. After quoting basic facts from graph theory, we will prove that interference graphs of SSA-form programs always have perfect elimination orders and show how they are determined.

2. SSA-form programs

Here, we consider a program (in SSA-form) given by its control flow graph (CFG) whose nodes are made up of labeled single instructions. We will therefore identify

[*] Corresponding author.
E-mail addresses: hack@ipd.info.uni-karlsruhe.de (S. Hack), ggoos@ipd.info.uni-karlsruhe.de (G. Goos).

0020-0190/$ – see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.ipl.2006.01.008

```
if ... then                if ... then
    $\ell_2 : x \leftarrow \cdots$          $\ell_2 : x_1 \leftarrow \cdots$
else                       else
    $\ell_3 : x \leftarrow \cdots$          $\ell_3 : x_2 \leftarrow \cdots$
end                        end
$\ell_4 : y \leftarrow x + 1$          $\ell'_4 : x_3 \leftarrow \phi(x_1, x_2)$
                           $\ell_4 : y_1 \leftarrow x_3 + 1$
```

Fig. 1. Program fragment and its equivalent in SSA-form.

```
$\ell_1 : a \leftarrow \cdots$
if ... then
    $\ell_2 : \cdots$
else
    $\ell_3 : b \leftarrow \cdots$
end
$\ell_4 : y \leftarrow \phi'(a, b)$
```

Fig. 2. Liveness at ϕ'-functions.

the node and its label in the following. Let us call the set of labels $\mathcal{L}$. The CFG has one distinct start node which has no predecessor nodes and is denoted by **start**. The instruction corresponding to a node is of the following form:

$$\ell : (d_1, \ldots, d_m) \leftarrow \tau(u_1, \ldots, u_n).$$

We denote the operation τ at a label ℓ by Op_ℓ. We call $\mathrm{Res}_\ell = \{d_1, \ldots, d_m\}$ the ordered set of result values and $\mathrm{Arg}_\ell = \{u_1, \ldots, u_n\}$ the ordered set of argument values at the label ℓ. All $d_1, \ldots, d_m$ and $u_1, \ldots, u_n$ are elements of the set of (abstract) values $\mathcal{V}$ of the considered program. Given a label ℓ, let us denote $\mathrm{Arg}_\ell(i)$ the ith argument to the operation at ℓ. Each label has an ordered set of k predecessor labels which we will denote by $P^1_\ell, \ldots, P^k_\ell$.

We will also write $\ell' \rightarrow^i \ell$ if $\ell' = P^i_\ell$. If we do not care about the position, we simply write $\ell' \rightarrow \ell$ to denote that ℓ' is a predecessor to ℓ. A path p is an ordered set $\{\ell_1, \ldots, \ell_n\}$ of at least two nodes, for which $\ell_1 \rightarrow \ell_2, \ell_2 \rightarrow \ell_3, \ldots, \ell_{n-1} \rightarrow \ell_n$ holds. To indicate that a set $p = \{\ell_1, \ldots, \ell_n\}$ is a path, we write $p : \ell_1 \rightarrow \cdots \rightarrow \ell_n$.

Finally, we say a label ℓ *dominates* another label ℓ' if each path from **start** to ℓ' contains ℓ, writing $\ell \preceq \ell'$. Note, since $\preceq$ is reflexive, $\ell \preceq \ell$.

We require the program to be given in SSA-form, which means that each variable is statically only assigned once. Usually, when a program is transferred into SSA-form, for each definition of a non-SSA variable x, a SSA variable x_i is created. The usages of the variables then have to be adjusted to refer to the corresponding SSA variable. This is not always possible. It might be dependent on the control flow, which definition of the variable is applicable at the usage. Consider the left program in Fig. 1. At label ℓ_4, it is dependent on the control flow whether the definition at ℓ_2 or the one at ℓ_3 is relevant for ℓ_4. In SSA-form programs, a special instruction, called ϕ-function is used to disambiguate multiple definitions by control flow. A ϕ-function copies its ith parameter to its result, if it was reached via P^k_ℓ. Note, that a basic block can have multiple ϕ-functions. So, since the program is in SSA-form, for each SSA-variable[1] v there is exactly one label D_v for which $v \in \mathrm{Res}_{D_v}$.

Since we allow only one instruction per label, we replace the set of all ϕ-operations in a basic block

$$y_1 = \phi(x_{11}, \ldots, x_{1n}),$$

$$\ldots$$

$$y_m = \phi(x_{m1}, \ldots, x_{mn})$$

by the more concise ϕ'-operation:

$$\ell : (y_1, \ldots, y_m) = \phi'(x_{11}, \ldots, x_{1n}, \ldots, x_{m1}, \ldots, x_{mn})$$

which sets $y_i = x_{ij}$ if ℓ was reached via P^j_ℓ. It is convenient to define $\mathrm{Arg}'_\ell(j) = \{x_{ij} \mid 1 \leqslant i \leqslant m\}$ subsuming all operands of a ϕ'-operation which refer to P^j_ℓ. Note, that ϕ' is totally equivalent to the traditional ϕ, since SSA semantics states that all ϕ-operations in a basic block are evaluated *simultaneously*. In the following, everything stated for ϕ'-operations implicitly holds for ϕ-operations. Thus we will only use ϕ'.

2.1. Liveness in SSA-form programs

To perform register allocation on SSA-form programs, a precise notion of liveness is needed. The standard definition of liveness

A variable v is live at a label ℓ, if there is a path from ℓ to a usage of v not containing a definition of v.

cannot be straightforwardly transferred to SSA-form programs. The problem arises with ϕ- and accordingly ϕ'-operations. Consider the program in Fig. 2. Surely, a is not live at label ℓ_3 although there is a path from ℓ_3 to a usage of a, namely ℓ_4. The cause for this oddity is that the usual notion of *usage* does not hold for ϕ'-operations. In addition to their arguments, a ϕ'-operation also uses control flow information to produce its result. To make the traditional definition of liveness work, we have to incorporate the predecessor by which a label was reached into the notion of usage:

[1] SSA variables are often called *values*.

Definition 1 *(Usage).*

$$\text{usage} : \mathbb{N} \times \mathcal{L} \times \mathcal{V} \to \mathbb{B},$$
$$(i, \ell, v) \mapsto \begin{cases} v \in \text{Arg}_\ell & \text{if } \text{Op}_\ell \neq \phi', \\ v \in \text{Arg}'_\ell(i) & \text{if } \text{Op}_\ell = \phi'. \end{cases}$$

Now, a usage is not only dependent on a label and a value but also on a number which represents the predecessor by which the label was reached. In our example in Fig. 2, usage$(1, \ell_4, a)$ is true, since a is indeed used if ℓ_4 is entered via ℓ_2. usage$(2, \ell_4, a)$ is false, since a is *not* used if ℓ_4 is reached through ℓ_3. If the operation at a label is not ϕ', this definition resembles the common concept of usage by simply ignoring the predecessor index.

The traditional definition of liveness quoted above, uses paths which end in usages of some variable to define liveness. In this traditional setting, usages and paths are unrelated. With Definition 1, paths and usages are no longer unrelated. So it is straightforward to merge them in one term.

Definition 2 *(Usepath).* A path $p : \ell_1 \to \cdots \to \ell_n$ is a *usepath* from ℓ_1 to ℓ_n concerning a value v, if v is used at ℓ_n regarding this path. More formally:

$$\text{usepath} : \mathcal{L}^n \times \mathcal{V} \to \mathbb{B},$$
$$(p : \ell_1 \to \cdots \to \ell_n, v)$$
$$\mapsto \begin{cases} \text{usage}(i, \ell_n, v) & \text{if } p = \ell_1 \to^i \ell_n, \\ \text{usepath}(\ell_2 \to \cdots \to \ell_n, v) & \text{otherwise.} \end{cases}$$

Referring to the example in Fig. 2, ℓ_1, ℓ_2, ℓ_4 is a usepath of a and ℓ_1, ℓ_3, ℓ_4 is a usepath of b.

Using this definition of usage together with the traditional definition of liveness stated above, one obtains a realistic model of liveness in SSA programs:

Definition 3 *(Liveness).* A value v is live at a label ℓ_1 iff there exists a label ℓ_n with usepath$(\ell_1 \to \ell_2 \to \cdots \to \ell_n, v)$ and $D_v \notin \{\ell_2, \ldots, \ell_n\}$.

We use the definition of usepaths to re-formulate the notion of a strict program coined by Budimlić et al. [3].

Definition 4 *(Strict program).* A program is called *strict*, iff for each value v each path from **start** to some label ℓ with usepath(**start** $\to \cdots \to \ell, v)$ contains the definition of v.

From now on, we will only consider strict programs.[2] The next lemma is essential for the rest of this paper and has also been given by Budimlić relying on a slightly different liveness definition.

Lemma 5. *Each label ℓ at which a value v is live is dominated by D_v.*

Proof. Suppose, ℓ is not dominated by D_v. Then, there is path from **start** to ℓ not containing D_v. From the fact, that v is live at ℓ follows, that there is a usepath of v from ℓ to some ℓ'. So there is a usepath from **start** to ℓ' not containing D_v which contradicts the definition of a strict program. □

2.2. Common facts about SSA-form programs

Since the definition of liveness given above seems rather unusual, we shortly derive some well-known facts about SSA-form programs from our definition. These facts are not vital for the rest of the paper and are only given to clarify certain properties of SSA-form programs.

Corollary 6. *Each value v, used in a non-ϕ'-operation at a label ℓ is dominated by its definition.*

Proof. Then, for each predecessor of P^j_ℓ of ℓ, usage(j, ℓ, v) holds. With Definition 3, v is live at each P^j_ℓ. With Lemma 5, D_v dominates each predecessor of ℓ. Thus D_v also dominates ℓ.

Corollary 7. *If a value $v \in \text{Arg}'_\ell(i)$ for a ϕ'-operation at a label ℓ and some i, then the definition of v dominates P^i_ℓ.*

Proof. Surely, usage(i, ℓ, v) holds. So $p : P^i_\ell \to \ell$ is a usepath concerning v. So, after Definition 3, v is live at P^i_ℓ. Thus, with Lemma 5, $D_v \preceq P^i_\ell$.

Corollary 8. *Let ℓ be a label with* $\text{Op}_\ell \neq \phi'$. *Each pair of values $v, w \in \text{Arg}_\ell$ interfere.*

Proof. Due to Definition 3, v and w are live at each predecessor of that label. So v and w interfere.

Often, one can read statements like:

[2] Surely, each non-strict program can be turned into a strict one by inserting instructions which initially define the variables by an arbitrary value.

- ϕ'-operations do not cause interferences.
- Concerning liveness, ϕ'-operations can be treated as if they had no arguments.
- ϕ'-operations extend the lifetimes of their arguments to the end of the respective predecessor label.

All these statements try to describe Corollary 8 the other way around. With the definition of usepaths, they are covered implicitly. In our model the basic assumption is, that the property of usage is always tied to a value *and* a path, which makes Corollary 8 the "special" case.

3. Graph theory

Here we quote definitions from basic graph theory and the theory of perfect graphs important to this paper. Let $G = (V, E)$ be an undirected graph. If there is an edge from $v \in V_G$ to $w \in V$, we write $vw \in E_G$. We leave out the G in E_G, V_G if it is clear from the context, which graph is considered. We call a graph G *complete*, iff for each $v, w \in V$, there is an edge $vw \in E$. We call G' an induced subgraph of G, if $V_{G'} \subseteq V_G$ and for all nodes $v, w \in V_{G'}$, $vw \in E_G \to vw \in E_{G'}$ holds.

Definition 9 *(Simplicial vertex).* A vertex $v \in V_G$ is called *simplicial*, if v and its neighbors induce a complete subgraph in G.

Definition 10 *(Perfect Elimination Order, PEO).* We call a linearly ordered sequence of vertices $v_1, \dots, v_n$ a *perfect elimination order*, if each v_i is simplicial in $G - \{v_1, \dots, v_{n-1}\}$ where $G - \{a_1, \dots, a_m\}$ is the graph obtained by deleting all vertices $\{a_1, \dots, a_m\}$ and their incident edges from the graph.

The class of graphs for which perfect elimination orders exist are also called *chordal* or *triangulated* graphs. Gavril [6] gives an algorithm for coloring chordal graphs in $O(|V|^2)$. The algorithm constructs a PEO for a given chordal graph by searching and removing a simplicial node from the graph each step. Afterwards, the nodes are inserted into the graph in reverse order. Each node is assigned a color which is not occupied by a neighbor of the node to insert. It is further proven, that this algorithm leads to a minimal coloring of the graph.

4. Interference graphs of SSA-programs

We say two values v and v' interfere, iff there is a label ℓ where v and v' are live (regarding Definition 3). Now, we can define the interference graph $IG = (V, E)$ of an SSA-form program. The set of vertices is made up by the values occurring in the program, $V_{IG} = \mathcal{V}$. Since nodes in the interference graph and values are identical, we identify both terms in the following. We draw an edge between to values v and v' iff they interfere and write $vv' \in E_{IG}$. The following lemmas lead to a theorem that connects the dominance relation of a program to perfect elimination orders in the interference graph of that program. Lemmas 11 and 12 have also been shown by Budimlić et al. [3] and are given for the sake of completeness, here.

Lemma 11 shows that each edge in the interference graph is directed according to the dominance relationship of the values their nodes represent.

Lemma 11. *If two values v and w are live at some label ℓ, either D_v dominates D_w or vice versa.*

Proof. By Lemma 5, D_v and D_w dominate ℓ. Thus, either D_v dominates D_w or D_w dominates D_v. □

The next lemma shows what is trivial in basic blocks also holds for complete programs in SSA-form: if one value starts living before another (it dominates the other) and both interfere, the value is still alive at the definition of the other.

Lemma 12. *If v and w interfere and $D_v \preceq D_w$, then v is live at D_w.*

Proof. Assume, v is not live at D_w. Then there is no usepath from D_w to some ℓ' concerning v. Since all labels where w is live are dominated by D_w, there is no label where v and w are simultaneously live. So v and w do not interfere which contradicts the proposition. □

Lemma 13 shows how the dominance order relation is reflected by the interference graph. It says that all values dominating a value v and interfering with v form a clique in the interference graph. This is used later on to connect the perfect elimination order to the dominance relation.

Lemma 13. *$ab, bc \in E$ and $ac \notin E$. If $D_a \preceq D_b$, then $D_b \preceq D_c$.*

Proof. Due to Lemma 11, either $D_b \preceq D_c$ or $D_c \preceq D_b$. Assume $D_c \preceq D_b$. Then (with Lemma 12), c is live at D_b. Since a and b also interfere and $D_a \preceq D_b$, a is also live at D_b. So, a and c are live at D_b which cannot be by precondition. □

Lemma 14. *A value v can extend a perfect elimination order, if each value whose definition is dominated by D_v is already contained in the PEO.*

Proof. To extend a PEO, v must be simplicial. Assume v is not simplicial. Then there exist two neighbors a, b for which $va, vb \in E$ *but* $ab \notin E$ (by Definition 9). Due to the proposition, all values whose definitions are dominated by D_v have already been removed from *IG*. Thus, D_a dominates D_v. By Lemma 13, D_v dominates D_b which contradicts the proposition. Thus, v is simplicial. □

Theorem 15. *The interference graph of a SSA-form program P is chordal.*

Proof. Consider the tree T of immediate dominators (cf. [9]) concerning the control flow graph of P. We start with an empty PEO and apply Lemma 14 recursively on T starting at the leaves. This constructs a PEO for the interference graph of P. Since each graph which has a PEO is chordal, cf. [7], the interference graph of P is chordal.

As we can see by Theorem 15, a post-order visitation of the dominator tree of a program yields a PEO. Since the vertices are colored reversely along a PEO, a pre-order visitation of the dominator tree defines a sequence in which the values can be colored optimally using the algorithm described in [6]. Since the liveness analysis annotates the set of live values to each label, we always have the set of neighbors present upon coloring a value. Thus, we do not have to construct the interference graph itself.

5. Leaving the SSA-form

As no real-world processor has a ϕ-instruction the compiler has to destroy the SSA-form of the program at some point in time. Conventionally, ϕ-functions are replaced by copies in its predecessor blocks to implement the control flow dependent copy as described in Section 2. In doing so, one modifies the interference graph of the program since new interferences are introduced, as shown in Fig. 3. x_3 is now interfering with y_1 and y_2 which has not been the case in the interference graph of the SSA-form program. These interferences are introduced due to the fact, that the atomic, simultaneous evaluation by ϕ'-functions (as mentioned in Section 2) is broken down to a sequential set of operations. In the worst case, these new interferences render the interference graph un-chordal which also might invalidate our

```
if ... then                                  if ... then
    $\ell_2 : x_1 \leftarrow \cdots$                      $\ell_2 : x_1 \leftarrow \cdots$
    $\ell_3 : y_1 \leftarrow \cdots$                      $\ell_3 : y_1 \leftarrow \cdots$
else                                             $\ell_4 : x_3 \leftarrow x_1$
    $\ell_4 : x_2 \leftarrow \cdots$                      $\ell_5 : y_3 \leftarrow y_1$
    $\ell_5 : y_2 \leftarrow \cdots$                  else
end                                              $\ell_4 : x_2 \leftarrow \cdots$
$\ell_6 : (x_3, y_3) \leftarrow \phi'(x_1, y_1, x_2, y_2)$        $\ell_5 : y_2 \leftarrow \cdots$
                                                 $\ell_4 : x_3 \leftarrow x_2$
                                                 $\ell_5 : y_3 \leftarrow y_2$
                                             end
```

Fig. 3. A program fragment in SSA-form and after destroying SSA using copy instructions.

register allocation. We can however destroy the SSA-form of the program and convert a register allocation with k registers of a SSA-form program into a register allocation with exactly the same number of registers for the resulting non-SSA-form program.

Consider a ϕ'-function

$$\ell : (y_1, \ldots, y_m) \leftarrow \phi'(x_{11}, \ldots, x_{1n}, \ldots, x_{m1}, \ldots, x_{mn})$$

at some label ℓ. Arriving at ℓ and coming from P_ℓ^j, the x_{ij} are copied *at once* into the y_i according to SSA semantics. Consider a valid register allocation The simultaneous assignment given by the ϕ'-function corresponds to a l-to-m mapping of registers, where $l \leqslant m \leqslant k$, and k represents the number of register available.[3] Furthermore, all y_i are assigned to different registers, since all y_i interfere. So the question of removing a ϕ'-function reduces to implementing l-to-m mappings between registers on the control flow edges to the ϕ's label using ordinary processor instructions *and m* registers.

Theorem 16. *Any simultaneous assignment from l registers to m registers, where $l \leqslant m$, can be implemented with m registers using only copy and swap instructions.*

Proof. Consider following simultaneous assignment:

$$(y_1, \ldots, y_m) \leftarrow (\underbrace{x_1, \ldots, x_1, \ldots, x_l, \ldots, x_l}_{m}).$$

In general, there may be multiple y_i to which the same x_j is assigned. For each x_j we arbitrarily pick one of the y_i to which it is assigned and denote it by $[x_j]$. Note, that this induces an equivalence relation $\sim$ on the $y_1, \ldots, y_n$: $y_i \sim y_j$ if there is some x_k which is assigned to y_i and y_j. Thus, y_i and y_j are members of the equiv-

[3] Note, that the same value x can be assigned to different y_i by a ϕ'-function. E.g., $(y_1, y_2) = \phi'(a_1, b_1, a_1, b_2)$.

alence class $[x_k]$. We denote the set of the x_j by X and the set of the equivalence classes $[x_j]$ by $[X]$.

Consider a register allocation $\rho : \mathcal{V} \to \mathcal{R}$ of the SSA-form program obtained by the algorithm described in the last section. Let π by a function mapping $\rho(x_j)$ to $\rho([x_j])$. Note, that since all y_i interfere, all $[x_j]$ also interfere and by the fact that all x_j interfere, π is injective. π may also be partial, since l might be smaller than $k = |\mathcal{R}|$.

Each register in $\rho([X])$ which is not in $\rho(X)$ can be assigned immediately since its value is not needed anymore. So we apply the following recursive scheme: Let $y = \pi(x)$. If $y \in [X]$ and $y \notin X$ we issue a copy from $\rho(x)$ to $\rho(y)$ and recursively consider the mapping $\pi|_{X \setminus \{x\}}$.

At the end of this recursive procedure, either all elements of $[X]$ are processed and thus all of X since π is injective or the remaining subset of $[X]$ equals the one of X. Thus this rest represents a permutation of registers which can be, as known from basic linear algebra, implemented by a sequence of swap instructions. If the processor does not possess swap instructions, one can use three xors, adds or subs (cf. [10]).

Finally, each $y_i \in [x_j]$ can be processed by copying $\rho([x_j])$ to $\rho(y_i)$. $\square$

6. Conclusions

We have shown that the interference graphs of strict SSA-form programs are chordal which leads to a coloring algorithm running in quadratic time. Furthermore, the coloring algorithm does not need to have the interference graph materialized but uses a coloring sequence induced by the dominance relation of the program. We also showed, how a register allocation of a SSA-form program using m registers can be turned into a register allocation of a corresponding non-SSA program using also no more than m registers, by implementing the ϕ'-functions properly.

7. Related work

At the time this paper was submitted, chordal graphs played no role in register allocation. Meanwhile, they have drawn the attention of other researchers in the area. The paper which initiated our research on the topic is by Andersson [1] who investigated interference graphs in real-world compilers and found that all of them were 1-perfect (1-perfectness means that the chromatic number of the graph equals the size of its largest clique). The result of the quest for a proof of this observation is this paper. Independently of us, Brisk proved the perfectness of strict SSA-form programs [2]. In his proof he also shows their chordality without referring to it. Pereira and Palsberg extended Andersson's studies and found that, with SSA-optimizations enabled, 95% of the (non-SSA!) interference graphs of the Java standard library were chordal. They use this fact to derive new spilling and coalescing heuristics for graph coloring register allocators. Finally, the authors of this paper published a more technical proof (without using perfect elimination orders) of this paper's result in a technical report [8].

Acknowledgements

We thank our colleagues Michael Beck, Marco Gaertler, Götz Lindenmaier and especially Rubino Geiß for many fruitful discussions. We also thank the anonymous referees for their suggestions helping to improve this paper.

References

[1] C. Andersson, Register allocation by optimal graph coloring, in: G. Hedin (Ed.), CC 2003, Lecture Notes in Comput. Sci., vol. 2622, Springer-Verlag, Heidelberg, 2003, pp. 33–45.

[2] P. Brisk, F. Dabiri, J. Macbeth, M. Sarrafzadeh, Polynomial time graph coloring register allocation, in: 14th Internat. Workshop on Logic and Synthesis, ACM Press, New York, 2005.

[3] Z. Budimlić, K.D. Cooper, T.J. Harvey, K. Kennedy, T.S. Oberg, S.W. Reeves, Fast copy coalescing and live-range identification, in: Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, ACM Press, New York, 2002, pp. 25–32.

[4] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, P.W. Markstein, Register allocation via coloring, J. Comput. Languages 6 (1981) 45–57.

[5] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, F.K. Zadeck, An efficient method of computing static single assignment form, in: Symp. on Principles of Programming Languages, ACM Press, New York, 1989, pp. 25–35.

[6] F. Gavril, Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and independent set of a chordal graph, SIAM J. Comput. 1 (2) (1972) 180–187.

[7] M.C. Golumbic, Algorithmic Graph Theory and Perfect Graphs, Academic Press, New York, 1980.

[8] S. Hack, Interference graphs of programs in SSA-form, Tech. Rep. 2005-25, Universität Karlsruhe, June 2005.

[9] T. Lengauer, R.E. Tarjan, A fast algorithm for finding dominators in a flowgraph, Trans. Programm. Languages Systems 1 (1) (1979) 121–141.

[10] H.S. Warren, Hacker's Delight, Addison-Wesley, Reading, MA, 2003.