Interference Graphs of Programs in SSA-form

Interner Bericht 2005-15

Sebastian Hack

Institut für Programmstrukturen und Datenorganisation
Fakultät für Informatik
Universität Karlsruhe
ISSN 1432-7864

Abstract

Register allocation is the task of mapping the variables in a program to processor registers. This problem is often reduced to coloring the so-called interference graph which the compiler computes from the input program. Theoretically, for each undirected graph there is a program having that graph as its interference graph. In this paper, we show that interference graphs of programs in SSA-form are *chordal*.

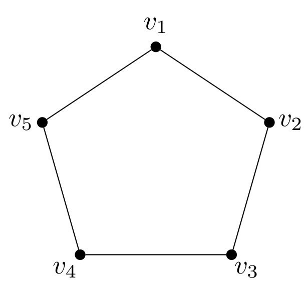
1 Introduction

The register allocation problem was stated as a graph coloring problem by Chaitin [CAC+81]. For each variable in the program, there is a vertex in the interference graph. Whenever the compiler finds out that two variables cannot be held in the same register (they are simultaneously live), an edge is drawn between the two corresponding vertices in the interference graph.

It is easy to see, that each undirected graph is the interference graph of some program. Consider an undirected graph $G = (V, E)$. Following program has G as its interference graph: for each vertex in V, there is a variable in this program. For each edge between two vertices v and w (we will shortly write $vw \in E$), the following fragment causes an interference between the variables for v and w:

$$\begin{aligned} v &:= 0 \\ w &:= 1 \\ v &:= v + w \end{aligned}$$

Repeating these three lines for each edge in G yields a program which has G as interference graph. For example, consider the graph G in figure 1(a) and its generating program in figure 1(b).



(a) Interference Graph G

$$\begin{aligned} v_1 &:= 0 \\ v_2 &:= 1 \\ v_1 &:= v_1 + v_2 \\ &\dots \\ v_5 &:= 0 \\ v_1 &:= 1 \\ v_5 &:= v_5 + v_1 \end{aligned}$$

(b) The program

Figure 1: A program and its interference graph G

Andersson [And03] posed the question, whether interference graphs are always *perfect* (see definition 6) and validated a weaker condition, the so-called 1-perfectness for a set of interference graphs produced by several compilers, experimentally. However, pathological counterexamples can be constructed easily, as one can see in figure 1(b). This graph is the so called C^5 (see section 3) and is the smallest example of a non-perfect graph.

This changes if we do not consider variables but *values* for allocation. The essential property of the SSA-form is, that for each value, there is exactly one variable or equivalently, each variable is (statically) only assigned once. So, looking at the example above, the edge from v_5 to v_1 can only exist, if v_5 is assigned twice which violates the SSA property. This suggests that *not* every undirected graph might occur as an interference graph to programs in SSA-form. In fact, the interference graphs of SSA-form programs are just the set of *chordal* graphs. Chordal graphs are known to be *perfect* (see [Gol80] for example).

The rest of this report is organized as follows: In the next section, we precisely describe our model of a program, the SSA-form of a program and the notion of liveness in our setting. In section 3, we quote some basic definitions from graph theory. Finally, in section 4, we prove that the interference graphs of programs in SSA-form are chordal.

2 Prerequisites

2.1 Programs

We assume a program to be given by its control flow graph (CFG). A CFG is a directed graph, whose nodes are called labels. Each label in a CFG corresponds to one instruction[1] of the form

$$\ell : (y_1, \dots, y_m) := \tau(x_1, \dots, x_n)$$

where $\mathrm{Op}_\ell = \tau$ is the instruction performed at this label defining the variables $y_1, \dots, y_m$ using $\mathrm{Arg}_\ell = \{x_1, \dots, x_n\}$. Furthermore, each label ℓ has a set of control flow predecessor labels

$$P_\ell = \{P_\ell^1, \dots, P_\ell^n\}$$

If ℓ' is the i-th predecessor of ℓ we also write $\ell' \to^i \ell$, if the predecessor index is irrelevant, we simply write $\ell' \to \ell$. Each CFG has a distinct label **start** with $P_{\mathbf{start}} = \emptyset$. A sequence of labels $p = \{\ell_1, \dots, \ell_n\}$ is called a *path,* if $\ell_1 \to \ell_2, \dots, \ell_{n-1} \to \ell_n$. We then also write $p : \ell_1 \to \dots \to \ell_n$. As we require our program to be in SSA-form, each variable is only assigned once. We will denote the label at which a variable v is defined by D_v.

Special care has to be taken for ϕ-instructions. SSA semantics states, that all ϕ-instructions in a basic block are simultaneously evaluated upon entering that block. Thus, we have to put all ϕ-instructions at the start of a block into one label. Alternatively, we define an instruction ϕ', which subsumes such a set of ϕ-instructions. We replace

$$\begin{aligned} \ell : y_1 &= \phi(x_{11}, \dots, x_{1n}) \\ &\dots \\ y_m &= \phi(x_{m1}, \dots, x_{mn}) \end{aligned}$$

[1] Note, that this setting implies that the instructions are already scheduled. We explicitly do not consider the kind of register allocation problems stated by Sethi [Set75], in which a schedule has to be found to minimize the register pressure.

by the more concise:

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

which sets $y_i = x_{ij}$ if ℓ was reached via P_ℓ^j. For convenience, we define

$$\mathrm{Arg}'_\ell[j] = \{x_{ij} \mid 1 \leq i \leq m\}$$

to refer to the arguments of a ϕ'-instruction corresponding to P_ℓ^j.

The notion of dominance is crucial to programs in SSA-form. We say, a label ℓ_1 *dominates* a label ℓ_2, if each path from **start** to ℓ_2 contains ℓ_1 and write $\ell_1 \preceq \ell_2$.

2.2 Liveness

The interference graph G of a program is built based on the information retrieved by the liveness analysis, which computes for each label the set of *live* variables according to following definition:

Definition 1 (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.*

Assume the set of variables live at a label ℓ is $\{v_1, \ldots, v_n\}$, then there is an edge $v_i v_j \in E_G$ for each v_i, v_j with $1 \leq i < j \leq n$.

Remark 1 (Interference). Two variables v and w interfere iff there exists a label at which both are live.

For ordinary instructions, it is clear that all their arguments interfere due to definition 1. However, this is not true for ϕ'-instructions because the traditional definition of usage does not hold for ϕ'-instructions. If v is an argument to a ϕ'-instruction at a label ℓ, it depends on the predecessor by which ℓ is reached, if v is used at ℓ or not. So, to make the traditional definition of liveness work, we have to incorporate the predecessors of a label into the notion of usage:

Definition 2 (Usage)**.**

$$\begin{aligned} \mathrm{usage} : \mathbb{N} \times \mathit{Labels} \times \mathit{Variables} &\rightarrow \mathbb{B} \\ (i, \ell, v) &\mapsto \begin{cases} v \in \mathrm{Arg}_\ell & \textit{if } \mathrm{Op}_\ell \neq \phi' \\ v \in \mathrm{Arg}'_\ell[i] & \textit{if } \mathrm{Op}_\ell = \phi' \end{cases} \end{aligned}$$

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. If the instruction 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. Since with definition 2, a usage is also dependent on control flow information, it is straightforward to merge usage and paths into one term:

Definition 3 (Usepath)**.** *A path $p : \ell_1 \to \dots \to \ell_n$ is a* usepath *from ℓ_1 to ℓ_n concerning a value v, iff v is used at ℓ_n regarding this path. More formally:*

$$\begin{aligned} \text{usepath} : \mathit{Paths} \times \mathit{Variables} &\to \mathbb{B} \\ (p : \ell_1 \to \ldots \to \ell_n, v) &\mapsto \begin{cases} \text{usage}(i, \ell_n, v) & \textit{if } p = \ell_1 \to^i \ell_n \\ \text{usepath}(\ell_2 \to \dots \to \ell_n, v) & \textit{otherwise} \end{cases} \end{aligned}$$

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

We use the definition of usepaths to re-formulate the notion of a strict program coined by [BCH+02].

Definition 5 (Strict Program)**.** *A program is called* strict*, iff for each value v, each path from* **start** *to some label ℓ with* usepath$(\mathbf{start} \to \dots \to \ell, v)$ *contains the definition of v.*

3 Chordal Graphs

In this section, we quote definitions from basic graph theory and the theory of perfect graphs important to this report. Let $G = (V, E)$ be an undirected graph. We call a graph G *complete*, iff for each $v, w \in V_G$, there is an edge $vw \in E_G$ and denote it by K^n, $n = |V_G|$. We call H an induced subgraph of G, if $V_H \subseteq V_G$ and for all nodes $v, w \in V_H$, $vw \in E_G \iff vw \in E_H$ holds. H is called a clique if H is complete and $H \subseteq G$ for some G. $\omega(G)$ is the size of the largest clique in G. A graph $G = (V, E)$ with $V = \{v_1, \dots, v_n\}$ and $E = \{v_1v_2, \dots, v_{n-1}v_n, v_nv_1\}$ is called a cycle and is denoted by C^n.

A *coloring* is a partition of V_G into subsets $C_1, \dots, C_k$ whereas $v, w \in C_m$ implies that $vw \notin E_G$. The *chromatic number* $\chi(G)$ is the smallest k for which $C_1, \dots, C_k$ is a coloring of G.

Definition 6. *A graph G is called* perfect, *iff $\omega(H) = \chi(H)$ for each $H \subseteq G$.*

Definition 7. *A graph G is called* chordal *iff it does not contain any induced C^n for $n \geq 4$.*

Remark 2. Let $G = (V, E)$ be an undirected graph. In general, determining $\chi(G)$ is NP-complete. If G is chordal, determining $\chi(G)$ can be done in $O(|V|^2)$ as proved in [Gav72].

4 Interference Graphs of SSA-form Programs

In this section, we prove that interference graphs of programs in SSA-form are chordal. An informal reason why interference graphs of SSA-form programs must be chordal is given by the work of GAVRIL [Gav74], who shows that chordal graphs are the intersection graphs of trees[2]. Since the dominance relation defines a tree (see [LT79]) and the (non-ϕ') usages of values are dominated by their definition, the lifetime of a variable in an SSA-form program can be thought of as such a tree.

In the following, we consider a *strict* program (see definition 5) and its interference graph G. Let us begin by proving some lemmas[3].

Lemma 1. *Each label ℓ at which a value v is live is dominated by D_v.*

Proof. Assume, ℓ is not dominated by D_v. Then there exists a path from **start** to ℓ not containing D_v. From the fact that v is live at ℓ it follows that there is a usepath of v from ℓ to some ℓ' not containing D_v (see definition 4). This implies, that there is a usepath of v from **start** to ℓ' not containing D_v which is impossible in a strict program. □

Lemma 2. *If two values v and w are live at some label ℓ, either D_v dominates D_w or vice versa.*

Proof. By Lemma 1, D_v and D_w dominate ℓ. Thus, either D_v dominates D_w or D_w dominates D_v. □

Lemma 3. *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 of v from D_w to some ℓ'. So v and w cannot interfere. □

Lemma 4. *Let $ab, bc \in E_G$ and $ac \notin E_G$. If $D_a \preceq D_b$, then $D_b \preceq D_c$.*

Proof. Due to Lemma 2, either $D_b \preceq D_c$ or $D_c \preceq D_b$. Assume $D_c \preceq D_b$. Then (with Lemma 3), 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. □

Finally, we can prove that the interference graph of a program in SSA-form contains no cycle larger than 3:

Theorem 1 (Chordality)**.** *The interference graph of a program in SSA-form is chordal.*

Proof. We will prove the theorem by showing that G has no induced subgraph $H \cong C^n$ for any $n \geq 4$. We consider a chain in G

$$x_1 x_2 \dots x_n \in E_G \quad \text{with } n \geq 4 \text{ and } \forall i \geq 1,\, j > i+1 : x_i x_j \notin E_G$$

Without loss of generality we assume $D_{x_1} \preceq D_{x_2}$. Then, by induction with Lemma 4, $D_{x_i} \preceq D_{x_{i+1}}$ for all $1 < i < n$. Thus, $D_{x_i} \preceq D_{x_j}$ for each $j > i$.

[2] in this context, a tree is a kind of interval with one start point and multiple end points which are directed downwards.

[3] Lemmas 1 and 3 have also been given by BUDIMLIĆ in [BCH+02] under a different setting of liveness

Assume, there is an edge $x_1 x_n \in E_G$. Then, there is a label ℓ where x_1 and x_n are live. By Lemma 1, ℓ is dominated by D_{x_n} and due to the latter paragraph, ℓ is also dominated by each $D_{x_i}, 1 \leq i < n$. Let us consider a label $D_{x_i}, 1 < i < n$. Since D_{x_i} dominates ℓ, there is a path from D_{x_i} to ℓ. Since D_{x_i} does *not* dominate D_{x_1}, there is a path from D_{x_i} to ℓ which does *not* contain D_{x_1}. Thus, x_1 is live at D_{x_i}. As a consequence, $x_1 x_n \in E_G$ implies $x_1 x_i \in E_G$ for all $1 < i \leq n$. So, G cannot contain an induced $C^n, n \geq 4$ and thus is *chordal.* □

References

[And03] Christian Andersson. Register allocation by optimal graph coloring. In G. Hedin, editor, *CC 2003*, volume 2622 of *LNCS*, pages 33–45, Heidelberg, 2003. Springer-Verlag.

[BCH+02] Zoran Budimlić, Keith D. Cooper, Timothy J. Harvey, Ken Kennedy, Timothy S. Oberg, and Steven W. Reeves. Fast copy coalescing and live-range identification. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 25–32. ACM Press, 2002.

[CAC+81] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Journal of Computer Languages*, 6:45–57, 1981.

[Gav72] Fănică Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and independent set of a chordal graph. *SIAM Journal on Computing*, 1(2):180–187, June 1972.

[Gav74] Fănică Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *J. Combin. Theory Ser. B*, (16):47–56, 1974.

[Gol80] Martin Charles Golumbic. *Algorithmic Graph Theory And Perfect Graphs*. Academic Press, 1980.

[LT79] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *Transactions on Programming Languages And Systems*, 1(1):121–141, July 1979.

[Set75] Ravi Sethi. Complete register allocation problems. *SIAM Journal On Computing*, 4(3):226–248, 1975.