Compiler Construction 2009/2010
Register Allocation for Programs in SSA-Form

Peter Thiemann

February 8, 2010
Outline

1. Motivation
2. Foundations
3. Spilling
4. Coloring
5. Coalescing
6. Register Constraints
7. Conclusion

- register allocation maps temporaries to physical registers such that their live ranges do not interfere
- common technique: graph coloring [Chaitin] of the interference graph
Example: Program and its Interference Graph

Three Registers Needed

\[
\begin{align*}
  a &\leftarrow 1 \\
  b &\leftarrow a + a \\
  c &\leftarrow a + 1 \\
  d &\leftarrow b + 1 \\
  &\text{store } c \\
  e &\leftarrow 1 \\
  d &\leftarrow a + 1 \\
  &\text{store } e \\
  &\text{store } d
\end{align*}
\]
Example Program in SSA Form

- Two registers available: but copy instruction needed
- Three registers available: use all and eliminate copy
SSA and Register Allocation

- $\phi$-functions replaced by moves before register allocation
- moves lead to coalescing
- may lead to spill
any undirected graph occurs as inference graph of a program
finding a minimal $k$-coloring of a general graph is NP-complete
hence, the heuristic feedback algorithm Build $\rightarrow$ Coalesce $\rightarrow$ Color $\rightarrow$ Spill? required
[coalescing changes colorability of graph]
In a chordal graph, every cycle of four or more nodes has a chord, i.e., an edge between two of the nodes that does not belong to the cycle. (Also: triangulated graph)
Background Graph Theory

Definition

- In a **perfect graph**, the chromatic number of each induced subgraph is equal to the size of its largest clique.
- **Chromatic number of** $G$: Minimum $k$ such that $G$ is $k$-colorable.
- **Clique** fully connected subgraph.

- In a perfect graph, graph coloring can be solved in polynomial time.
Interference graphs of SSA programs are **chordal graphs** see also [Pereira&Palsberg 2005] [Brisk 2005] [Bouchez,Darte&Rastello 2005]

⇒ spilling and coalescing can be decoupled

⇒ Every chordal graph is a perfect graph

⇒ number of registers needed = size of largest clique
  the largest set of variables that are live at the same time

⇒ Spilling can be performed once and for all before register allocation
Coloring a chordal graph takes $O(|V|^2)$

Given the dominator tree and the live ranges, then coloring takes $O(\omega(G) \cdot n)$ time
- $n$ number of instructions
- $\omega(G)$ size of largest clique in $G$
  - $\leq$ number of registers after spilling

Usually, $\phi$-functions $\mapsto$ move instructions

Early coalescing is harmful

Instead of coalescing, try to assign the same color
Outline

1. Motivation
2. Foundations
3. Spilling
4. Coloring
5. Coalescing
6. Register Constraints
7. Conclusion
\[ \phi\text{-functions are not functions, but a notational device} \]
\[ \phi\text{-functions do not cause interference} \]
\[ \text{There is no ordering among different } \phi\text{-functions at the beginning of a block; ideally, they should “evaluate” simultaneously} \]
\[ \Rightarrow \text{ different notation} \]
\[ y_1 \leftarrow \phi(x_{11}, \ldots, x_{1n}) \]
\[ \vdots \quad \vdots \]
\[ y_m \leftarrow \phi(x_{m1}, \ldots, x_{mn}) \]
\[ \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix} \leftarrow \Phi \begin{bmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{bmatrix} \]
Let $D_v$ be the node defining $v$.

**Lemma:** If two registers $v$ and $w$ are live at node $n$, then either $D_v$ dominates $D_w$ or $D_w$ dominates $D_v$.

**Lemma:** If $v$ and $w$ interfere and $D_v$ dominates $D_w$, then $v$ is live at $D_w$.

**Lemma:** Let $(u, v)$ and $(v, w)$ be edges in the interference graph, but not $(u, w)$.
If $D_u$ dominates $D_v$, then $D_v$ dominates $D_w$. 

Interference Graphs of SSA Programs
Problem: the interference graph does not reflect the number of uses of a register

⇒ ∃ work to break the live ranges in smaller pieces

[Bouchez 2005] shows that “splitting live ranges to lower the register pressure to a fixed $k$ while inserting a minimum number of reload instructions is NP-complete”
Lemma

For each clique $C \subset G$ with $V_C = \{v_1, \ldots, v_n\}$, there is a permutation $\sigma : V_C \rightarrow V_C$ such that $D_{\sigma(v_i)}$ dominates $D_{\sigma(v_{i+1})}$ for $1 \leq i < n$.

Theorem

Let $G$ be the interference graph of an SSA program and $C$ be an induced subgraph of $G$. $C$ is a clique in $G$ iff there exists a label in the program where all $V_C$ are live.
Let $\ell$ be a node where $l > k$ variables are live
Belady’s algorithm spills those $l - k$ variables whose uses are farthest away (in minimum number of instructions executed) from $\ell$.

$$\text{nextuse}(\ell, v) = \begin{cases} 
\infty & \text{if } v \text{ not live at } \ell \\
0 & \text{if } v \text{ not used at } \ell \\
1 + \min_{\ell' \in \text{succ}[\ell]} \text{nextuse}(\ell', v) & \text{otherwise}
\end{cases}$$

Apply Belady’s algorithm to each basic block
Belady’s Algorithm for Basic Blocks

- Let $P$ be the set of variables passed into block $B$: the variables live-in at $B$ and the results of the $\phi$-functions.
- Let $\sigma : P \rightarrow P$ be a permutation which sorts $P$ ascendingly according to $\text{nextuse}$.
- ⇒ Pass the set of variables $I = \{p_{\sigma(1)}, \ldots, p_{\sigma(\min(k,l))}\}$ in registers.
- Traverse the nodes in a basic block from entry to exit.
- Let $Q$ be the set of all variables currently in registers ($|Q| \leq k$, initially $Q \leftarrow I$).
At an instruction

\[ \ell : (y_1, \ldots, y_m) \leftarrow \tau \left( x_1, \ldots, x_n \right) \]

\( D_\ell \)

\( U_\ell \)

set \( R \leftarrow U_\ell \setminus Q \)

if \( R \neq \emptyset \), then

- reloads have to be inserted and \( \max(|R| + |Q| - k, 0) \) variables are removed from \( Q \)
- remove those with highest nextuse

If \( v \in I \) is displaced before used, then \( v \) need not be passed to \( B \) in a register

Let \( in_B \) be the set \( v \in I \) which are used in \( B \) before they are displaced.
Belady’s Algorithm for Basic Blocks
continued

- \( \tau \) displaces \( \max(|D_\ell| + |Q| - k, 0) \) variables from \( Q \)
- To decide which variables to displace we use

\[
\text{nextuse}'(\ell, v) = 1 + \min_{\ell' \in \text{succ}[\ell]} \text{nextuse}(\ell', v)
\]

- Let \( out_B \) be the set \( Q \) after processing the last node in a block
Belady’s Algorithm Extended

- To connect the blocks, ensure that each variable in $in_B$ is in a register on entry to $B$.
- At the end of each predecessor $P'$ of $B$ insert reloads for all $in_B \setminus out_{P'}$ (recall edge splitting)
Coloring Chordal Graphs

- perfect elimination orders (PEO)
- order in which variables are removed from graph
- basis: simplicial nodes (all neighbors belong to the same clique)
- **Lemma**: Each chordal graph has a simplicial node
- Removing a node from a chordal graph preserves chordality
- PEOs are related to the dominance tree
Theorem
An SSA variable $v$ can be added to a PEO of $G$ if all variables whose definitions are dominated by the definition of $v$ have been added to the PEO.

Proof
For a contradiction, assume $v$ is not simplicial. Hence, $v$ has two neighbors $a$ and $b$ which are not connected. As all variables whose definitions are dominated by $D_v$ are already part of the PEO and removed, it must be that $D_a$ dominates $D_v$. By a previous lemma, $D_v$ dominates $D_b$, contradicting the assumption.
COLORPROGRAM (Program $P$)
COLORRECURSIVE (start block of $P$)

COLORRECURSIVE (Basic block $B$)

assigned ← colors of the live-in($B$)
for each instruction $(b_1, \ldots, b_m) \leftarrow \tau(a_1, \ldots, a_n)$ from entry to exit do
  for $a \in \{a_1, \ldots, a_n\}$ do
    if last use of $a$ then
      assigned ← assigned \ color($a$)
  for $b \in \{b_1, \ldots, b_n\}$ do
    color($b$) ← one of allcolors \ assigned
  for each $C$ where $B = idom(C)$ do
    COLORRECURSIVE($C$)
Coalescing Phase

- **Goal:** minimize number of copy/move instructions
- **Causes of copy/move instructions**
  - $\phi$-functions
  - register constraints of target architecture (pre-colored nodes)
Implementation of $\phi$-functions

- Seems to require two registers
- However, implementing $\Phi$ by the moves $i_3 \leftarrow i_2; j_3 \leftarrow j_2$ creates an interference between $i_3$ and $j_2$
Interference from Implementation of $\Phi$

\[
\begin{align*}
i_3 & \quad \quad \quad j_3 \\
i_2 & \quad \quad \quad j_2 \\
i_1 & \quad \quad \quad j_1
\end{align*}
\]
Consider \((b_1, \ldots, b_n) \leftarrow \sigma(a_1, \ldots, a_n)\)

A multi-assignment that permutes the contents of the registers according to \(\sigma\)

For the example program, a permutation is needed that swaps two registers:
Example Program After Register Assignment

- \( R_1 \leftarrow 1 \)
- \( R_2 \leftarrow 1 \)

\[
\begin{pmatrix}
R_1 \\
R_2
\end{pmatrix}
\leftarrow \Phi
\begin{bmatrix}
R_1 & R_2 \\
R_2 & R_1
\end{bmatrix}
\]

if \( R_2 < 100 \)

- return \( R_2 \)
- \( R_1 \leftarrow R_2 + R_1 \)
- \( R_2 \leftarrow R_2 + 1 \)
Example Where Copying is Needed

\[
\begin{align*}
    i_1 &\leftarrow 1 \\
    \begin{pmatrix}
        i_3 \\
        j_3
    \end{pmatrix} &\leftarrow \Phi \begin{bmatrix}
        i_1 & i_2 \\
        i_1 & j_2
    \end{bmatrix} \\
    \text{if } i_3 &< 100
\end{align*}
\]

\[\begin{align*}
    \text{return } j_3 & \\
    j_2 &\leftarrow j_3 + i_3 \\
    i_2 &\leftarrow j_3 + 1
\end{align*}\]

\(\Phi\) duplicates \(i_1\) into \(i_3\) and \(j_3\)
Example Where Copying is Needed

\[
\begin{align*}
  i_1 &\leftarrow 1 \\
  j_1 &\leftarrow 1 \\
  \begin{pmatrix} i_3 \\ j_3 \end{pmatrix} &\leftarrow \Phi \begin{bmatrix} i_1 & i_2 \\ j_1 & j_2 \end{bmatrix} \\
  \text{if } i_3 &< 100 \\
  \text{return } j_3 \\
  j_2 &\leftarrow i_1 + i_3 \\
  i_2 &\leftarrow j_3 + 1
\end{align*}
\]

- \( i_1 \) interferes with \( \Phi \)
Duplication in the Removal of $\Phi$

- Duplication (i.e., extra registers) are only needed if
  - a $\Phi$ argument is used multiple times in one column
  - a $\Phi$ argument is live-in at the block of $\Phi$
- Interference with a value defined by $\Phi$ does not require duplication.
Register swaps  Swap instructions of the processor;
xor trick: \[ a \leftarrow a \oplus b; \ b \leftarrow a \oplus b; \ a \leftarrow a \oplus b \]

Moves  assuming a free backup register, each cycle \( C \) can be implemented with \(|C| + 1\) move instructions for example, \$at\ in MIPS
The cost of implementation for a permutation $\sigma$ is related to the number of fixpoints of $\sigma$

Variable $x$ is a fixpoint if

$$(\ldots, x', \ldots) = \sigma(\ldots, x, \ldots)$$

and $x$ and $x'$ are assigned the same register

$\Rightarrow$ no code needs to be generated for a fixpoint
Optimizing $\Phi$-functions

Problem Statement

Given a $k$-coloring $f : V \rightarrow \{1, \ldots, k\}$ define the cost of $p$ by

$$c_f(\ell) = \sum_{i=1}^{m} \sum_{j=1}^{n} \text{cost}_f(y_i, x_{ij})$$

where $\text{cost}_f(a, b) = \begin{cases} w_{ab} & \text{if } f(a) \neq f(b) \\ 0 & \text{otherwise} \end{cases}$ with $w_{ab} \geq 0$ the cost of copying $b$ to $a$.

The overall cost of a program $P$ under coloring $f$ is

$$c(P, f) = \sum_{\ell \text{ is } \Phi\text{-node}} c_f(\ell)$$
Optimizing $\Phi$-functions

Problem Statement

SSA-Maximize-Fixed-Points

Given an SSA program $P$ and its interference graph $G$. Find a coloring $f$ of $G$ for which $c(P, f)$ is minimal.

Theorem

SSA-Maximize-Fixed-Points is NP-complete.
Heuristics for Optimizing $\Phi$-functions

- Start with a $k$-coloring
- Modify color assignments to lower the cost
  Non-local changes in the coloring may be required!
- A valid $k$-coloring is always maintained
- For each row $i$ of the $\Phi$-function

\[
\begin{pmatrix}
  p_1 \\
  \vdots \\
  p_m
\end{pmatrix}
\leftarrow \Phi
\begin{bmatrix}
  a_{11} & \ldots & a_{1n} \\
  \vdots & \ddots & \vdots \\
  a_{m1} & \ldots & a_{mn}
\end{bmatrix}
\]

define an optimization unit (OU) consisting of $p_i$ and all $a_{ij}$
that do not interfere with $p_i$ (at least one)
Perm-Optimizer

\textbf{COALESCE}(G)
\begin{align*}
\text{pinned} & \leftarrow \emptyset \\
\textbf{for} \text{ each OU } (p, a_1, \ldots, a_k) \textbf{ do} \\
& \quad \textbf{for} \text{ each color } c \text{ assignable to } p \textbf{ do} \{\text{Init}\} \\
& \quad \quad C_c \leftarrow G[p, a_1, \ldots, a_k] \{\text{conflict graph}\} \\
& \quad \quad S_c \leftarrow \text{max weighted stable subset of } C_c \{\text{weight of } a_i \text{ is } w_{pa_i}\} \\
& \quad \text{Insert } (c, C_c, S_c) \text{ in min-queue } Q \{\text{ordered by } w(S_c)\} \\
\textbf{repeat} \{\text{Test}\} \\
& \quad \text{candidates} \leftarrow \emptyset \\
& \quad g \leftarrow f \{\text{copy the current coloring}\} \\
& \quad \text{pop } (c, C, S) \text{ from } Q \\
& \quad C' \leftarrow \text{TEST}(c, C, S) \\
& \quad \textbf{if } C' \neq \text{nil} \textbf{ then} \\
& \quad \quad S' \leftarrow \text{maximum weighted stable subset of } C' \\
& \quad \quad \text{Insert } (c, C', S') \text{ into } Q \\
\textbf{until} \ C' = \text{nil} \\
& \textbf{if } |\text{candidates}| > 1 \textbf{ then} \\
& \quad \text{pinned} \leftarrow \text{pinned} \cup \text{candidates} \\
& \quad f \leftarrow g \{\text{update coloring}\}
\end{align*}
\( \text{TEST}(c, C, S) \)
\{ \( S = \{p, a_1, \ldots, a_l\} \) processed in this order \}
for \( u \in S \) do
\( (s, v) \leftarrow \text{TRYCOLOR}(u, \text{nil}, c) \)
if \( s = \text{ok} \) then
\( \text{candidates} = \text{candidates} \cup \{u\} \)
else if \( s = \text{candidate and } v \neq p \) then
return \( (V_C, E_C \cup \{(v, u)\}) \)
else
return \( (V_C, E_C \cup \{(u, u)\}) \)
return nil
TRYCOLOR($v \in V_G, u \in V_G, c$)

$c_v \leftarrow g(v)$

if $c = c_v$ then
    return (ok, nil)
else if $v \in \text{pinned}$ then
    return (pinned, $v$)
else if $v \in \text{candidates}$ then
    return (candidate, $v$)
else if $c$ is not allowed for $v$ then
    return (forbidden, $v$)

for each $n$ with $(v, n) \in E_G, n \neq u, g(n) = c$ do

    { try to swap colors with neighbor }
    $(s, v') \leftarrow \text{TRYCOLOR}(n, v, c_v)$

    if $s \neq \text{ok}$ then
        return $(s, v')$

$g(v) \leftarrow c$

return (ok, nil)
Outline

1. Motivation
2. Foundations
3. Spilling
4. Coloring
5. Coalescing
6. Register Constraints
7. Conclusion
Most processor architectures have instructions where the operands are restricted to specific registers.

Graph coloring approach:
1. split live range at constraining definition
2. add one pre-colored node for each register
3. connect definition with all pre-colored nodes, except the one with the required color

For chordal graphs, coloring is in P iff each color is used only once in pre-coloring.

Unrealistic constraint for register allocation
⇒ Delegate to the Perm-Optimizer
Register Constraints by Perm-Optimization

- Insert $(a'_i) = \Phi[a_i]$ (for all live registers) in front of each instruction with register constraints
  - all live variables can change register at that point
  - interference graph breaks in two unconnected components
  - each color occurs only once as pre-coloring in each component
- first do coloring, then Perm-Optimization
Example Register Constraints

Code and Colored Interference Graph

\[ a_{R_1} \leftarrow \ldots \]
\[ b \leftarrow \ldots \]
\[ c \leftarrow b + 1 \]
\[ d \leftarrow a + 1 \]
\[ e_{R_1} \leftarrow b + c \]
\[ f \leftarrow c + d \]
\[ \vdots \]
Example Register Constraints with $\Phi$ Inserted

Code and Colored Interference Graph

\[
\begin{align*}
a_{R_1} & \leftarrow \ldots \\
b & \leftarrow \ldots \\
c & \leftarrow b + 1 \\
d & \leftarrow a + 1 \\
\begin{pmatrix} b' \\ c' \\ d' \end{pmatrix} & \leftarrow \Phi \begin{bmatrix} b \\ c \\ d \end{bmatrix} \\
e_{R_1} & \leftarrow b' + c' \\
f & \leftarrow c' + d' \\
\ldots
\end{align*}
\]
Outline

1 Motivation
2 Foundations
3 Spilling
4 Coloring
5 Coalescing
6 Register Constraints
7 Conclusion
Interference graphs for SSA programs are chordal
⇒ main phases of register allocation (spilling, coloring, coalescing) can be decoupled
Procedure for spilling based on the correspondence live sets ↔ cliques in interference graph (without constructing the graph)
(Optimal spilling via ILP solving)
Optimal coloring in \textit{linear time} (w/o constructing the graph)
Optimal coalescing is NP-complete
  \begin{itemize}
    \item Heuristic
      \begin{itemize}
        \item (Optimal coalescing via ILP solving)
      \end{itemize}
  \end{itemize}
Register constraints expressible
Alternatives

- [Pereira&Palsberg, APLAS 2005] observe that 95% of the methods in the Java 1.5 library give rise to chordal interference graphs and give an algorithm for register allocation under this assumption.
- [Pereira&Palsberg, PLDI 2008] give a general, industrial strength framework for register allocation based on puzzle solving. It first transforms its input to elementary programs, a strengthening of SSA programs.
- [Pereira&Palsberg, CC 2009] propose a different, spill-free way to perform SSA elimination after register coloring.
- [Pereira&Palsberg, CC 2010] present Punctual Coalescing, a scalable, linear time, locally optimal algorithm for coalescing.
- [Hack&Good, PLDI 2008] register coalescing by graph recoloring.