Register Allocation by Priority-based Coloring

Presented by Stephen Hines
COP 5622: Advanced Topics in Compilation
September 2003
Presentation Overview

- Basic Principles
  - Live Ranges
  - Graph Coloring Algorithms
- Register Allocation by Priority-based Coloring (PBC)
  - Local Register Allocation
  - Global Register Allocation
    - Priority-based Node Coloring Algorithm
- Measured Results
- Conclusions
Register Allocation is a code-improving transformation that attempts to map local variables to registers.

This provides faster access to the values of these variables, since memory references and/or memory operations can be eliminated.

However, since registers are a limited resource, it is necessary to “decide” which variables should be allocated, and which will remain in memory.

Usually this transformation is considered as a graph coloring problem.
Live Ranges

- Live ranges denote the effective life span of a variable or register being used.
  - Definitions of the variable in the live range do not reach any other uses outside the live range.
  - Any uses of the variable in the live range are not defined from outside of the live range.
- Each register is assigned live ranges or pieces of live ranges that do not conflict (interfere).
- For variables, loads may be necessary at entry points of live ranges, and stores may be necessary at exit points.
When using caller-save procedure conventions, it will be necessary to store live scratch registers before calls and load these registers following calls.

This can have a great impact on later code improving transformations (such as determining which live ranges to associate with registers).

Bad choices can actually degrade performance or mask opportunities for further improvements.
Graph Coloring Algorithms

- Determining whether a graph is r-colorable is NP-complete
- Nodes are colored such that no two connected nodes have the same color
- The algorithm must continually guess colors for nodes and backtrack if necessary, choosing a different color scheme
  - Solution can be verified in polynomial time
  - But proving that no solution exists requires exponential time (trying each potential combination of colors for nodes)
Graph Coloring Algorithms (2)

- When applying this algorithm for **Register Allocation**:
  - Graph is referred to as the **Interference Graph**
    - Nodes – Register Candidates (live range of a register or variable)
    - Edges – Connections to other register candidates that are active (live) at the same time (hence interfering)
    - Colors – Registers
  - Must handle worst case behavior (e.g. Exponential time bound)
  - Usually does not consider costs of the allocation (register-memory operations) or other effects (necessary loads/stores around calls)
    - This is known as the **Overallocation Problem**
Register Allocation by Priority-based Coloring

- Proposed by Frederick Chow and John Hennessy (1984)
  - Implemented as part of the global optimizer UOPT
- Designed to improve upon traditional register allocation by graph coloring
- Machine independent algorithm, driven by machine dependent parameters
- Consists of 2 phases
  - Local Register Allocation
  - Global Register Allocation
Register Allocation by Priority-based Coloring (2)

- Traditional
  - Suffers from Overallocation Problem
  - If $r$ colors are not enough, then nodes must be marked for spilling, however these decisions greatly impact performance
  - Behaves poorly as number of registers needed for register allocation approaches the maximum number of registers for machine (possibly exponential run time due to backtracking)

- Priority-based Coloring
  - All variables have home locations, from which they can be loaded/stored, hence no need for spills
  - Algorithm stops when all registers are used up (no backtracking)
  - Run time is $O(r(l-r))$, where $l$ is # of live ranges and $r$ is # of registers
These three quantities are machine dependent, and will be used to make better choices for PBC Register Allocation later.

- **MOVCost**: The cost of a move from a register to memory or memory to a register.
- **LODSave**: The execution time savings for a register access as opposed to a corresponding load operation.
- **STRSave**: The execution time savings for setting a register as opposed to a corresponding store operation.
Local Register Allocation

- This is allocation of local registers in straight-line code
- Occurs before Global Register Allocation, to help reduce necessary work
- A variable's savings can be approximated as:

\[
NETSAVE = LODSAVE \times u + STRSAVE \times d - MOV\text{COST} \times n
\]

\[
u = \text{uses of variable}
\]

\[
d = \text{definitions of variable}
\]

\[
n = 0,1,2
\]

- The value of \(n\) varies with whether or not an initial load and/or final store is needed for the variable's live range
Because we are dealing with individual basic blocks, it is possible that NETSAVE is not as accurate as we would like.

Stores at the end of a block and Loads at the beginning can be removed after doing global allocation if the same register is assigned.

So instead we will compute both a MAXSAVE and a MINSAVE, corresponding to the maximum (no moves necessary), and the minimum respectively:

\[
\begin{align*}
\text{MAXSAVE} &= \text{LODSAVE} \times u + \text{STRSAVE} \times d \\
\text{MINSAVE} &= \text{LODSAVE} \times u + \text{STRSAVE} \times d - \text{MOVLCOST} \times n
\end{align*}
\]
Local Register Allocation (3)

- Criteria for local register allocation:

\[
\text{MINSAVE} < \text{MOVCOST} \times (p + s)
\]

\[
p = \text{number of predecessors}
\]

\[
s = \text{number of successors}
\]

- If this condition is satisfied, then the variable should be allocated a register for the duration of the basic block

- This is one way in which the Overallocation Problem is avoided
Global Register Allocation

- Performed after Local Register Allocation
- Costs and savings are weighted by loop nest depths
  - Inner loop variables should have higher priority
- Iterations of step 2. in the algorithm will color a single live range (in prioritized order)
- Does not suffer from the Overallocation Problem, since the benefit must outweigh the cost in order to perform the allocation
- Insufficient registers are handled by splitting the live ranges of variables
Priority-based Node Coloring

1. For each live range \( lr \), if \( \text{num_neighbors}(lr) < \text{num_colors_remaining} \), place it in the unconstrained pool.

2. Repeat (a-c) until all live ranges are colored or no registers left for live ranges:

   a. Perform (i) or (ii) on each live range \( lr \) until \( \text{TOTALSAVE} \) for each one has been computed:

      i. If \( \text{num_neighbors}(lr) < \text{num_colors_remaining} \):

         1) Calculate \( n_i \) (0-2) for each block \( i \) of \( lr \) depending on needed loads/stores

         2) Calculate \( \text{NETSAVE}_i \) for each block \( i \)

\[
\text{NETSAVE}_i = \text{LODSAVE} \times u_i + \text{STRSAVE} \times d_i - \text{MOV\text{\textsuperscript{COST}}} \times n_i
\]
Priority-based Node Coloring (2)

3) Calculate TOTALSAVE for lr, where \( w_i \) is the loop nesting depth

\[
TOTALSAVE = \sum_{i \in lr} (NETSAVE_i \times w_i)
\]

ii. If num_neighbors(lr) >= num_colors_remaining:

1) Split off a new live range \( lr_1 \) from lr, creating a new node for the graph.
   a) New node should start with a definition block from an entry point to \( lr \)
   b) Add blocks contiguous to \( lr_1 \), updating all neighbor information, until
      \( \text{num_neighbors}(lr_1) = \text{num_colors_remaining} - 1 \)

2) Place the new live range appropriately in the unconstrained pool, or in the
   constrained pool (for which we are calculating TOTALSAVE)

3) Move members in the unconstrained pool that are no longer unconstrained to the
   constrained pool
Priority-based Node Coloring (3)

- b. For each live range $lr$, compute $\text{ADJSAVE}$:
  \[
  \text{ADJSAVE} = \frac{\text{TOTALSAVE}}{\text{number of nodes}(lr)}
  \]

- c. Choose highest value of $\text{ADJSAVE}$ from all uncolored, constrained live ranges and assign a color to it

- 3. Assign colors to all live ranges in the unconstrained pool
Chow and Hennessy tested their algorithm with both the DEC 10 and the Motorola 68000.

Static allocation statistics display percentages of memory references that overcome the cost of allocation.

Since these machines have memory-memory operations, it makes sense to leave certain infrequently used variables in memory.

<table>
<thead>
<tr>
<th>Static Register Allocation Statistics</th>
<th>DEC 10</th>
<th>M68K</th>
</tr>
</thead>
<tbody>
<tr>
<td>% var refs in regs</td>
<td>0.77</td>
<td>0.86</td>
</tr>
<tr>
<td>% var assigns in regs</td>
<td>0.76</td>
<td>0.84</td>
</tr>
</tbody>
</table>
Running times for different optimizations show that register allocation is both effective and enabling.

<table>
<thead>
<tr>
<th>Average Running Times (percentages)</th>
<th>DEC 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Optimizations</td>
<td>1.00</td>
</tr>
<tr>
<td>Only Local Optimizations</td>
<td>0.95</td>
</tr>
<tr>
<td>Local Opt. + Register Allocation</td>
<td>0.79</td>
</tr>
<tr>
<td>All Except Register Allocation</td>
<td>0.87</td>
</tr>
<tr>
<td>Full Global Optimizations</td>
<td>0.61</td>
</tr>
</tbody>
</table>

The number of registers available for allocation also affects running times, but obviously there are diminishing returns.

<table>
<thead>
<tr>
<th>Average Running Times (percentages)</th>
<th>DEC 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Registers</td>
<td>1.00</td>
</tr>
<tr>
<td>2 Registers</td>
<td>0.88</td>
</tr>
<tr>
<td>4 Registers</td>
<td>0.84</td>
</tr>
<tr>
<td>6 Registers</td>
<td>0.75</td>
</tr>
<tr>
<td>All 9 Registers</td>
<td>0.73</td>
</tr>
</tbody>
</table>
Conclusions

- **Priority-based Coloring** provides an efficient $O(r(l-r))$ complexity solution for performing Register Allocation
  
  - Not the optimal solution for allocation, but relatively fast to compute
  
- Machine independent implementation using machine dependent parameters
  
- Alleviates many problems with traditional graph coloring
  
  - **Overallocation Problem**
  
  - Bad worst case performance (exponential from backtracking)
  
  - Additional spills when not enough registers are available