Revisiting Graph Coloring Register Allocation
A Study of the Chaitin-Briggs and Callahan-Koblenz Algorithms

Keith Cooper, Anshuman Dasgupta, Jason Eckhardt

Presenter: Anshuman Dasgupta
Register Allocation

- Process of mapping values in the program to a limited set of physical registers on the target architecture
  - Program values contained in locations called *virtual registers*
  - Must handle arbitrarily large number of virtual registers
  - Registers are the fastest members in the memory hierarchy
  - Proficient allocation extremely important for application performance

- Most programs contain segments where the number of values exceeds the number of physical registers
  - Allocator must insert loads and stores: *spill code*
Register Allocation

- Spills are memory accesses and therefore expensive
- Register allocators attempt to minimize the number of spills
- Optimal register allocation is a NP-complete problem
  - Allocation algorithms use heuristics to approximate optimal solution
Effective approach: Use graph coloring to model the allocation problem
- Build an *interference graph*
- Construct live ranges from examining program values
- Live ranges are nodes in the graph
- Edges between nodes indicate that they cannot share a physical register. The nodes *interfere*.
- Each color represents a physical register
- Neighbor nodes cannot share the same color

The interference graph encodes safety constraints
- The allocator respects these constraints to preserve program semantics
Graph Coloring Register Allocation

- We examine two graph coloring allocation algorithms
  - Popular Chaitin-Briggs algorithm
  - Callahan-Koblenz algorithm

- We shall use two major points of comparison
  - Amount of spill code inserted
  - Efficacy of copy removal

- Copy removal: Tries to merge two live ranges connected by a register-to-register copy
  - Can decrease register pressure
  - Important for good allocation
- 6 major phases
- Aggressive coalescing phase
  - Iterates until no more copies can be coalesced away
- Simple spill insertion strategy if coloring fails
  - Choose spill candidates using heuristics (*spill costs*)
  - Spill all occurrences of candidate live range: loads before every use, stores after every definition
  - Restart process after adding spills
The Chaitin-Briggs Register Allocator

- Often cited shortcomings:
  - No topological program information preserved in interference graph
  - Approximated via spill costs
    - References in deeper loop nests given higher spill cost
  - Spill-everywhere approach

Different strategy suggested by Callahan and Koblenz…
The Callahan-Koblenz Register Allocator

- Callahan-Koblenz developed allocator around the same time as Briggs

- Augments Chaitin-style allocator:
  - Builds hierarchical structure (*tile tree*) to represent program flow
    - A tile is a set of basic blocks
  - Tile boundaries are candidates for live-range splitting
  - Tries to schedule spill code in less frequently executed blocks
  - Algorithm is more intricate than Chaitin-Briggs
Callahan-Koblenz: The Tile Tree

Tile T0: \{start, A, B, C, D\}
Tile T0.1: \{A, B, C\}
Tile T0.1.2: \{B\}
Tile T0.2: \{D\}
The Callahan-Koblenz Register Allocator

postorder traversal of tile tree

preorder traversal of tile tree

Cooper, Dasgupta, Eckhardt

LCPC 2005
The Callahan-Koblenz Register Allocator

- Implemented at Cray, published in 1991, but no comparison with Chaitin-Briggs

- Key questions:
  - How does the Callahan-Koblenz approach affect:
    - The number of dynamic spill instructions executed
    - The removal of register-to-register copies
  - Callahan-Koblenz inserts some extra branches. How does this affect performance?
Spill Code Insertion
Simple strategy for spill code insertion
- Choose candidates based on spill heuristic

Prefer spilling nodes with lower values

Heuristic function for live range $l$, $H(l) = \frac{\text{SpillCost}_l}{\text{Degree}_l}$

\[
\text{SpillCost}_l = \text{LoadCosts}_l + \text{StoreCosts}_l
\]

\[
\text{LoadCost}_l = \sum 10^{\text{loopdepth}(i)}
\]

\[
\text{StoreCost}_l = \sum 10^{\text{loopdepth}(j)}
\]

where $i \in \text{SpillLoads}(l)$, $j \in \text{SpillStores}(l)$
Callahan-Koblenz Spill Costs

Higher values indicate better fit for a register

\[
Weight_t = \sum_{s \in \text{subtiles}(t)} (Reg_s(v) - Mem_s(v)) + LocalWeight_t(v)
\]

\[
LocalWeight_t(v) = \sum_{b \in \text{blocks}(t)} P(b) \cdot Ref_b(v)
\]

\[
Transfer_t(v) = \sum_{e \in E(t)} P(e) \cdot Live_e(v)
\]

\[
Reg_t(v) = \begin{cases} 
0, & \text{if } \neg \text{InReg}_t(v) \\
\min(Transfer_t(v), Weight_t(v)), & \text{if } \text{InReg}_t(v)
\end{cases}
\]

\[
Mem_t(v) = \begin{cases} 
Transfer_t(v), & \text{if } \neg \text{InReg}_t(v) \\
0, & \text{if } \text{InReg}_t(v)
\end{cases}
\]
Experimental Methodology

- Implemented both allocators on LLVM
  - LLVM from Univ. of Illinois is a SSA-based, language independent, intermediate representation and compiler framework

- We ran our experiments on:
  - Pentium 4, 3.2 GHz., 1 GB RAM, Redhat Linux 9.0
  - 7 allocatable general purpose integer registers
  - 8 floating point registers

- Evaluated on SPEC CPU 2000 integer benchmarks and epic from the Mediabench suite
Dynamic Spill Code Comparison

Mean spill-code reduction: 20.5 %

Callahan-Koblenz can insert copies on tile boundaries

Improvement with tile boundary copies: 19.1%

On epic, does much worse…

Cooper, Dasgupta, Eckhardt

LCPC 2005
Why Callahan-Koblenz Performs Better

Before allocation

Cooper, Dasgupta, Eckhardt

Chaitin-Briggs

Callahan-Koblenz

Before allocation

Cooper, Dasgupta, Eckhardt

Chaitin-Briggs

Callahan-Koblenz

Heavy use of $x$

Heavy use of $x$

Heavy use of $x$
Note disproportionate number of dynamic memory spills on tile boundaries for epic

- Occurs due to differing locations for global values at each level in triply nested loops
- Can tweak spill heuristic to correct this anomaly
Removal of register-to-register copies
Inter-register Copy Removal

- Helps allocation by decreasing register pressure

\[ r_1 = x \text{ op } y \]
\[ r_2 = r_1 \]
use \( r_2 \)

\[ r_1 = x \text{ op } y \]
\[ r_2 = r_1 \]
use \( r_1 \)

After copy removal
Different Strategies Used For Copy Removal

- Chatin-Briggs uses coalescing and biased coloring
  - Coalesce if \( r_1 \) and \( r_2 \) are connected by a copy and do not interfere
  - Copies between a physical and virtual register (instruction peculiarities, procedure calling conventions) are marked
  - Coloring phase attempts to assign the same color to the virtual register

- Callahan-Koblenz uses preferencing
  - On encountering a copy between \( r_1 \) and \( r_2 \), add one to the other’s preference list
  - Try to satisfy preference during coloring

- Chaitin-Briggs’ strategy is far more aggressive
Coalescing + biased coloring outperforms preferencing

- 3.6% fewer static copies in code
- 4.5% fewer copies executed

We expected coalescing to win but were surprised at the competitive performance of preferencing
Tile tree construction may warrant an insertion of basic blocks

- Most inserted blocks fall through to successor. No extra branches needed
- Some do not.

- We measured the overhead of these branches
  - 5.8% more static branches
  - But only marginal increase in branches executed: 1.4%
- Branches at tile boundaries are infrequently executed
Cooper, Dasgupta, Eckhardt

**Execution Times of Allocated Code**

- Callahan-Koblenz achieves a 6.1% improvement over Chaitin-Briggs on average.
- We chose not to use this metric as our major criteria for comparison:
  - Very architecture dependent
  - Might not reflect qualitative differences in allocation.
Conclusions

- Considering program structure yields substantial reduction in dynamic spill code
  - Tile boundary based spilling outperforms spill-everywhere
- We were concerned about the performance of Callahan-Koblenz’s copy coalescing mechanism
  - Chaitin-Briggs is very aggressive in removing copies
- Preferencing does reasonably well
  - Performs within 4.5% of Chaitin-Briggs
- Control-flow overhead incurred by Callahan-Koblenz is small
Future Work

- Use insights gained from examining the allocators to devise better allocation strategies
  - Hybrid approach: Use aggressive coalescing with Callahan-Koblenz

- Several improvements have been suggested in the literature to address the spill-everywhere approach
  - Compare these strategies with Callahan-Koblenz

- Both allocators are compile-time intensive
  - Can we design faster allocators while preserving allocation efficacy?
  - Will be invaluable in a JIT environment