Register Allocation via Graph Coloring

Beyond Chaitin Briggs

Copyright 2015, Keith D. Cooper & Linda Torczon, all rights reserved.
Students enrolled in Comp 512 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Citation numbers refer to entries in the EaC2e bibliography.
Chaitin-Briggs Allocator (Optimistic Coloring)

- Renumber
- Build
- Coalesce
- Spill costs
- Simplify
- Select
- Build SSA, build live ranges, rename
- Build the interference graph
- Fold unneeded copies
  \[ LR_x \rightarrow LR_y, \text{ and } <LR_x, LR_y> \notin G \Rightarrow \text{combine } LR_x \& LR_y \]
- Estimate cost for spilling each live range
- Remove nodes from the graph
- While stack is non-empty
  - pop \( n \), insert \( n \) into \( G_i \) & try to color it
- Spill uncolored definitions & uses

While \( N \) is non-empty
  - if \( \exists n \) with \( n^c < k \) then push \( n \) onto stack
  - else pick \( n \) to spill push \( n \) onto stack remove \( n \) from \( G_i \)

Briggs’ algorithm
How do these allocators do?

**Results are “pretty good”**
- Simple procedures allocate without spills
- There is some room for improvement
  - Long blocks, regions of high pressure
  - Many implementation issues
- Many people have looked at improving Chaitin-Briggs

**Better allocations**
- Better coloring
- Softer coalescing
- Better spilling
- Spilling partial live ranges

**Better implementations**
- Faster graph construction
- Faster coalescing

**Different approximate graphs**
- Linear Scan allocation
- SSA-based allocation
- Koblenz-Callahan
Roadmap for Today’s Lecture

More Detail

• Building the interference graph
• Coalescing, biased coloring, & limited lookahead

Improvements

• Better coloring
• Better spilling, including live-range splitting, partial live-range spilling, and rematerialization

Different Approximations

• Linear scan allocators
• SSA-based allocators
• Koblenz-Callahan Hierarchical Allocator
Building the Interference Graph

Need two representations

- Bit matrix
  - Fast test for specific interference
  - Need upper (lower) diagonal submatrix
  - Takes fair amount of space & time

- Adjacency lists
  - Fast iteration over neighbors
  - Needed to find colors, count degree
  - Must tightly bound space to make it practical

Both Chaitin & Briggs recommend two passes [73,74,75,49,51,52,56]

- First pass builds bit matrix and sizes adjacency vectors
- Second pass builds adjacency vectors into perfect-sized arrays
Building the Interference Graph

Split the graph into disjoint register classes [101]
• Separate GPRs from FPRs
  ♦ Others may make sense (CCs, predicates)
• Graph is still $n^2$, but $n$ is smaller
• In practice, GPR/FPR split is significant

Clique separators [175]

Build adjacency lists in a single pass [101]
• Block allocate adjacency lists
  (30 edges per block)
• Reduce amount of wasted space & pointer overhead
• Simple time-space tradeoff

Significance:
• 75% of space (Chaitin-Briggs) with one fewer pass [101]
• 70% of time (Chaitin-Briggs) for whole allocation [101]
Building the Interference Graph

**Hash table implementation** [75, 158]

- If graph is sparse, replace bit-matrix with hash table
  - Chaitin tried it and discarded the idea
  - George & Appel claim it beat the bit matrix in space & time

**Our experience** [101]

- Finding a good hash function takes care
  - Universal hash function from Cormen, Leiserson, & Rivest
  - Multiplicative hash function from Knuth
- Takes graphs with many thousands of LRs to overtake split bit-matrix implementation

Significance:

- 199% to 656% space versus Chaitin-Briggs [101]
- 124% to 4500% allocation time versus Chaitin-Briggs [101]
Coalescing

A Little More Detail On Coalescing In Chaitin-Briggs  (See § 13.4.6 in EaC2e)

• Build the interference graph, $I$
  ♦ For a copy, $LR_x \rightarrow LR_y$, add edges from $LR_y$ to each node in LIVENOW except $LR_x$
  • If $LR_x \rightarrow LR_y$, and $<LR_x, LR_y> \notin I$, then allocator can combine $LR_x$ & $LR_y$ and delete the copy operation
    ♦ Briggs showed examples where coalescing eliminated 1/3 of the live ranges
  • Need to update $I$
    ♦ In general, $LR_{xy}$ interferes with the all of $LR_x$ and $LR_y$’s conflicts
    ♦ To get best results, need a precise update, so we iterate build-coalesce

The dominant cost in a Chaitin-Briggs allocator is graph building  [51]

• Circular problem: coalescing reduces number of live ranges, number of live ranges determines cost of building graph, coalescing needs graph, ...

• See later slides on speeding up the allocator

Comp 512, Spring 2015
Conservative Coalescing

**Chaitin’s scheme coalesces every copy that it can**

- Coalescing $i$ and $j$ can create $ij^\circ > \max(i^\circ, j^\circ)$
  - May make $ij$ harder to color
  - In some contexts, this is important

- We can limit coalescing to conservative cases [55, 56]
  - Idea: Only create $ij$ if it will get a color
  - Tempting to say that we need $ij^\circ < k$, so $ij$ is trivially colored
  - In fact, we need that $ij$ has fewer than $k$ neighbors of significant degree

- We can also bias the color selection [55,56]
  - If $i$ and $j$ are connected by a copy, try to pick the same color
  - If the other one is not yet colored, pick a color still open for it
  - Generalize to multiple copies \((but\ only\ immediate\ neighbors)\)
  - “biased coloring with limited lookahead”
Building on Conservative Coalescing

**Iterated Coalescing** [158]
- Use conservative coalescing, always
- If no trivially colored node remains, coalesce again
- Coalescing reduces degree in the graph
- Makes sense only if allocator uses conservative coalescing

**Invented for Standard ML of New Jersey**
- Long parameter lists, passed in registers
- Code shape adds many additional edges
- I think that they hit the known bug in “NeedLoad()” [Harvey, PhD thesis]
- Iterated coalescing cured their problem
Editorial Opinion

Conservative coalescing is oversold

• Designed to remove unproductive splits
  ♦ Insert “special” copies & coalesce ones that don’t help
  ♦ Worked pretty well for that purpose

• Looks great on paper
  ♦ Why coalesce if things get worse?
  ♦ Conservative coalescing never makes things worse

In practice, Chaitin gets most coalesced LRs

• Briggs should get even more
• Don’t be afraid to coalesce aggressively
  ♦ With passive splitting & IR spilling, might even be better

Support for Editorial Opinion

Donghua Liu built a coloring allocator where he could adjust the value of $k$

- Roughly, a Chaitin-Briggs allocator in LLVM
- Distinguished between $k$ in conservative coalescing & in coloring
- Distinguished between $k$ in integer registers & floating-point registers
  - Gave him 4 distinct values of $k$ to tune

Holding $k$ for coloring fixed, found the best value of $k$ for coalescing was around 39 to 42 on an Intel Nehalem processor (nominal 32 registers)
Better Coloring

Several Authors Have Tried To Improve The Quality of Coloring

- Optimal coloring  [Wilken]
  - Use backtracking to find minimal chromatic number
  - Took lots of compile-time
  - Found (some) better allocations

- Random-walk coloring  [Dietz]
  - Rather than spill, color remaining nodes in a random walk over the remaining graph
  - Did rather well on random graphs

Neither of these ideas has been widely used (beyond the original authors)

Unfortunately, some codes need more than $k$ registers
  - Better coloring will not help these codes
  - Only helps when better coloring eliminates spills – a narrow range of codes
One Last Coalescing Idea: Faster Coalescing

The bit matrix requires an $n^2$ data structure

• We reduce $n$ by including in the interference graph only IRs involved in copy operations
  ♦ An analog of Briggs’ semi-pruned SSA idea
  ♦ Only include in the analysis things that can matter
  ♦ Only works with “reckless” coalescing (i.e., non conservative)

Experience

• Informally, it runs in about 66% of the time of the coalescing with the full graph
Better Spilling

The Actual Performance Degradation Comes From Spilling

• Strongly suggests that we should look at better ways to spill
• If you want to reduce costs, you have to look where they are incurred
  → *Register allocation version of Dillinger’s observation*

What is wrong with Chaitin’s spill methodology?

• It chooses values to spill based on the graph rather than the code
  → Value that minimizes (Spill Cost/Degree) may not be live in region of high pressure

• Once it picks a value to spill, it spills that value everywhere
  → *Could limit spilling to regions where demand for registers is greater than supply*
  → *Could break spilled live ranges into pieces and try again* (live-range splitting)
Better Spilling

Some Proposed Improvements

• Clean spilling [38]
  ♦ Spill value once per block, if possible
  ♦ Avoids redundant loads & stores

• Best of three spilling [38]
  ♦ Simplify/Select is cheap relative to Build/Coalesce
  ♦ Try it with several different heuristics

• Rematerialization [55]
  ♦ Recognize values that are cheaper to recreate
  ♦ Rather than spill them, rematerialize them

• Spill partial live ranges

Each of these helps
Better Spill Choice Heuristics

• Clean spilling
  ♦ Minor computations during spill insertion
  ♦ Mostly a matter of paying attention to details

• Best of three spilling
  ♦ Just repeat Simplify/Select with different heuristics
  ♦ Gets at random parts of the algorithm (NP-noise)
  ♦ Might get some of it by renumbering – min of seven

• Rematerialization
  ♦ Tag each value with $c_i$, BOT, or TOP
  ♦ Propagate, performing meet at Ø-functions
  ♦ Modify spill cost computation & spill insertion

Comp 512, Spring 2015
Rematerialization

*Never-killed* values can be rematerialized \((\text{rather than spilled})\)

- Operands are always available
- Computed in a single operation

Cheaper to recompute than to store & reload \((\text{the classic spill})\)

**Allocator must**

- Discover & mark never-killed LRs
- Reflect rematerialization in spill costs
- Use all this knowledge to generate right spills

**Chaitin rematerialized LRs that were entirely never-killed**

- We can do partial LRs
Rematerialization

Big Picture

• Use SSA to break LR into component values
• Tag each component with a value
• Use Wegman & Zadeck SCCP to propagate tags
• Split off never-killed parts from rest of LR
  ♦ Use a “special” copy operation
  ♦ Special copies get coalesced with conservative coalescing
• Use tags to compute spill costs & to insert spill code
• Rely on conservative coalescing and biased coloring to remove unproductive splits (as before)

\[
\begin{align*}
\text{TOP} & \text{ defined by } \text{COPY or } \emptyset \\
\text{inst} & \text{ never-killed op ( ptr)} \\
\text{BOT} & \text{ defined by other op}
\end{align*}
\]
Spilling Partial Live Ranges

• Bottom-up splitting [81,82]
  ♦ Break uncolored live range into basic blocks
  ♦ Recombine them when it does not increase degree

• Aggressive splitting [49]
  ♦ Split aggressively based on the CFG
  ♦ Undo non-productive splits

• Interference region spilling [37]
  ♦ Spill just region that conflicts with colored nodes
  ♦ Run in competition with default spilling

• Passive splitting [106, 98]
  ♦ Use directed interference graph to identify splits
  ♦ Run in competition with default spilling

No data on how this does with Chaitin-Briggs

Improvements ran from + 4x to - 4x spill ops

Improvements of ~36% in spill ops vs. Briggs

Sometimes wins big vs. Bergner, sometimes loses
Interference Region Spilling

**Simple idea:**
- Find region where $i$ & $j$ are both live
- Spill $i$ around this interference region (IR)
- Can reduce total cost of spilling $i$
- Fits entirely in “Insert Spills” phase of a Briggs allocator

**The implementation**
- Take colored subgraph and list of uncolored nodes
- For each uncolored LR, find subranges that can be colored
  - Rest of LR is its IR
- Compare cost of spilling IR versus cost of spilling entire LR
  - Take cheaper alternative
Passive Splitting

Key observation

\[
\begin{align*}
x & \leftarrow \ldots \\
y & \leftarrow \ldots \\
\ldots & \\
\leftarrow x & \\
\leftarrow y &
\end{align*}
\]

\begin{align*}
x & \leftarrow \ldots \\
y & \leftarrow \ldots \\
\ldots & \\
\leftarrow y & \\
\leftarrow x &
\end{align*}

\begin{align*}
x & \leftarrow \ldots \\
y & \leftarrow \ldots \\
\leftarrow x & \\
\leftarrow y & \\
\leftarrow x &
\end{align*}

spilling x does not help with y  
spilling x helps with y  
spilling x does not help with y

The containment graph captures this effect
It is just a directed analog of the interference graph

Comp 512, Spring 2015
Passive Splitting

\[ x \leftarrow y \text{ and not } y \leftarrow x \text{ suggests splitting } x \text{ around } y \]

To split \( x \) around \( y \)
- store \( x \) before each definition of \( y \)
- load \( x \) after each death of \( y \)

What does it cost?
- one memory op at each border of the overlap region
- may (or may not) be cheaper than spilling \( x \) everywhere

This is the base case

In practice, we may need to split around several live ranges
Coloring allocators are often viewed as too expensive for use in JIT environments, where compile time occurs at runtime.

Linear scan allocators use an approximate interference graph and a version of the bottom-up local algorithm [Poletto & Sarkar]

- Interference graph is an interval graph
  - Optimal coloring (without spilling) in linear time
  - Spilling handled well by bottom-up local allocator

- Algorithm does allocation in a “linear” scan of the graph
- Linear scan produces faster, albeit less precise, allocations

Linear scan allocators hit a different point on the curve of cost versus performance.
Linear Scan Allocation

Building the Interval Graph

• Consider the procedure as a linear list of operations
• A live range for some name is an interval (x,y)
  ♦ x and y are the indices of two operations in the list, with x < y
  ♦ Every operation where name is live falls between x & y, inclusive
    → Precision of live computation can vary with cost
  ♦ Interval graph overestimates interference

The Algorithm

• Use Best’s algorithm — bottom-up local
• Distance to next use is well defined
• Algorithm is fast & produces reasonable allocations

Variations have been proposed that build on this scheme
Global Coloring from SSA Form

**Observation:** The interference graph of a program in SSA form is a chordal graph.

**Observation:** Chordal graphs can be colored in $O(N)$ time.

**These two facts suggest allocation using an interference graph built from SSA Form**

- Chaitin-Briggs works from live ranges that are a coalesced version of SSA names
- SSA allocators use raw SSA names as live ranges
- Allocate live ranges, then insert copies for $\phi$-functions

SSA-based allocation has created a lot of excitement in the last couple of years.

Global Coloring from SSA Form

Coloring from SSA Names has its advantages

• If graph is $k$-colorable, it finds the coloring
  ♦ (Opinion) An SSA-based allocator will find more $k$-colorable graphs than a live-range based allocator because SSA names are shorter and, thus, have fewer interferences.

• Allocator should be faster than a live-range allocator
  ♦ Cost of live analysis folded into SSA construction, where it is amortized over other passes
  ♦ Biggest expense in Chaitin-Briggs is the Build-Coalesce phase, which SSA allocator avoids, as it destroys the chordal graph
Global Coloring from SSA Form

**Coloring from SSA Names has its disadvantages**

- Coloring is rarely the problem
  - Most non-trivial codes spill; on trivial codes, both SSA allocator and classic Chaitin-Briggs are overkill. (Try linear scan?)

- SSA form provides no obvious help on spilling
  - Shorter live ranges will produce local spilling (good & bad)
  - May increase spills inside loops

- After allocation, code is still in SSA form
  - Need out-of-SSA translation
  - Introduce copies after allocation, which may create need to spill

- Need a post-allocation coalescing phase
  - Algorithms exist that do not use an interference graph
  - They are not as powerful as the Chaitin-Briggs coalescing phase

**TAANSTAAFL:** The problem is still NP-Complete. Changing the definition of live range does not make it solvable to optimality in polynomial time.
What About A Hybrid Approach?

How can the compiler attain both speed and precision?

**Observation:** lots of procedures are small & do not spill
**Observation:** some procedures are hard to allocate

**Possible solution:**
- Try different algorithms
- First, try linear scan
  - It is cheap and it may work
- If linear scan fails, try heavyweight allocator of choice
  - Might be Chaitin-Briggs, SSA, or some other algorithm
  - Use expensive allocator only when cheap one spills

This approach would not help with the speed of a complex compilation, but it might compensate on simple compilations
An Even Stronger Global Allocator

Hierarchical Register Allocation (Koblenz & Callahan)

- Analyze control-flow graph to find hierarchy of tiles
- Perform allocation on individual tiles, innermost to outermost
- Use summary of tile to allocate surrounding tile
- Insert compensation code at tile boundaries \((\text{LR}_x \rightarrow \text{LR}_y)\)

**Strengths**

- Decisions are largely local
- Use specialized methods on individual tiles
- Allocator runs in parallel

**Weaknesses**

- Decisions are made on local information
- May insert too many copies

- Anecdotes suggest it is fairly effective
- Target machine is multi-threaded multiprocessor (Tera MTA)

Eckhardt’s MS (Rice, 2005) shows that K&C produces better allocations than C&B, but is much slower
Partial Bibliography

- Briggs, Cooper, & Torczon, “Improvements to Graph Coloring Register Allocation,” ACM TOPLAS 16(3), May, 1994.
- Cooper, Harvey, & Torczon, “How to Build an Interference Graph,” Software–Practice and Experience, 28(4), April, 1998.