Global Register Allocation via Graph Coloring

The Chaitin-Briggs Algorithm

Comp 412

Copyright 2018, Keith D. Cooper & Linda Torczon, all rights reserved.

Students enrolled in Comp 412 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.

Chapter 13 in EaC2e
Notes on the Final Exam

• Scheduled Final Exam
  – December 5, 2018, 9AM to Noon, McMurtry Auditorium (DCH 1055)
  – Closed-book, closed-notes, closed-devices exam

• The exam covers material since the last exam, both lecture & book
  – Lectures start with “Computing inside the parser” and run through the last lecture (Friday)
  – Book sections:
    → Chapter 4, § 4.4 and 4.5.2
    → Chapter 6, except § 6.6
    → Chapter 7
    → Chapter 8, § 8.1, 8.2, 8.3, 8.4
    → Chapter 9, § 9.2.1, 9.2.2, 9.2.4 (reaching definitions)
    → Chapter 11, §11.1, 11.2, 11.3, 11.5
    → Chapter 12, except § 12.5
    → Chapter 13, § 13.1, 13.2, 13.4 (ignore 13.4.7), 13.5
We have covered selection & scheduling. Now for global register allocation ...

- Register use is critical to performance
- Global register allocation attempts to make the best decisions for procedure
  - Spill choices
  - Register assignment
- Problem is, of course, \textit{NP}-Complete
- To do a good job requires global analysis, global decision making, and local spill code insertion
Global Register Allocation

The Job

• At each point in the code, determine which values live in registers
• Select a PR for each such value
• A global allocator coordinates assignment & allocation across block boundaries

Critical Properties

• Produce correct code that uses \( k \) or fewer registers
• Minimize spills & restores
• Minimize space to hold spilled values
• Operate efficiently
  – \( O(n) \) to \( O(n^2) \), but not \( O(2^n) \)

PR \( \equiv \) physical register
\( k \) \( \equiv \) number of PRs
\( n \) \( \equiv \) number of operations
Modern Global Allocators

- Most global allocators operate via an analogy to graph coloring.
- Construct a “conflict graph” or “interference graph” that represents constraints on register co-occupancy:
  - One node per live range
  - Edge from \( \text{LR}_i \) to \( \text{LR}_j \) iff they cannot occupy the same register
- Find a \( k \)-coloring \( \Rightarrow \) allocation to \( k \) PRs

What happens when the allocator does not find a \( k \)-coloring?

- Spill some values, to create a nearby problem
- Try again on that nearby problem

A \( k \)-coloring of a graph \( G \) is an assignment of \( k \) colors to the nodes of \( G \) such that no two adjacent nodes have the same color. Typically, in a coloring problem, we favor smaller values of \( k \) over larger ones. In allocation, \( k \) is the number of available PRs.
Graph Coloring (A Background Digression)

The Problem
A graph $G$ is said to be $k$-colorable if and only if the nodes can be labeled with integers from 1 to $k$ such that no edge in $G$ connects two nodes with the same label. The labels are referred to as “colors”.

Examples

Each color can be mapped to a distinct physical register.
Global Register Allocation

Taking a global approach

• Abandon the distinction between local values and global values
• Make systematic use of registers and spill locations
• Adopt a general scheme to approximate good allocation

Graph-coloring paradigm

1. Build an interference graph $G$ for the procedure
   - The interference graph captures which LR\s cannot share a PR
2. (try to) construct a $k$-coloring for $G$
   - Minimal coloring is NP-Complete
   - Spill placement becomes a critical performance issue
     $\rightarrow$ Move spills to infrequently executed places?
3. Map colors into physical registers

(Lavrov ‘61 & Chaitin ‘81)
Structure of a Chaitin-Briggs Allocator

Names from optimizer → Rename Live Ranges → Build Graph → Coalesce Copies → Compute Spill Costs → Simplify Graph → Select Colors

- **Rename**
  - Build an interference graph and use it to eliminate unnecessary copy operations
- **Build Graph**
  - Estimate cost of spilling for each name
- **Coalesce Copies**
  - Try to color the graph in k or fewer colors
- **Compute Spill Costs**
  - If needed, spill uncolored values

Any copies coalesced?

Rewrite VRs with PRs → Spill

Any uncolored live ranges?
Structure of a Chaitin-Briggs Allocator

- Rename Live Ranges
- Build Graph
- Coalesce Copies
- Compute Spill Costs
- Simplify Graph
- Select Colors
- Spill

**Rename (what else would you expect)**

- Build an interference graph and use it to eliminate unnecessary copy operations
- Estimate cost of spilling for each name
- Try to color the graph in k or fewer colors
- If needed, spill uncolored values

Names from optimizer → Live range names

- Rename Live Ranges
  - Live range names
  - Any copies coalesced?
  - Rewrite VRs with PRs

- Build Graph

- Coalesce Copies
  - Any copies coalesced?

- Compute Spill Costs

- Simplify Graph

- Select Colors

- Spill
  - Any uncolored live ranges?

COMP 412, Fall 2018
Live Ranges

In the multi-block case, live ranges are more complex than in the local case.

- Consider x, y, & z in the code to the right
Live Ranges

In the multi-block case, live ranges are more complex than in the local case.

• Consider \( x, y, \) & \( z \) in the code to the right
  – \( x \) has 2 distinct live ranges
Live Ranges

In the multi-block case, live ranges are more complex than in the local case.

• Consider x, y, & z in the code to the right
  – x has 2 distinct live ranges
  – y has 2 distinct live ranges
Live Ranges

In the multi-block case, live ranges are more complex than in the local case.

- Consider x, y, & z in the code to the right
  - x has 2 distinct live ranges
  - y has 2 distinct live ranges
  - z has just 1 live range
    - The 2 defs must be in the same LR so that the use in $B_3$ has 1 name for them
    - z is never live in $B_2$
- Finding live ranges in the global context takes global analysis
Rename Into Live Ranges  (without SSA Form)

The allocator systematically renames values into live ranges

• Rename so that each definition point has a unique name
• Propagate those names forward to uses
  – Classic global data-flow analysis problem: *reaching definitions* ¹
  – Annotates each use with a **DEFS** set and each definition with a **USES** set
• At each use, union all the names in **DEFS** into a set ²
• At each definition, union all the names & sets in **USES** into a set ²
• Creates a **web** of definitions and uses that are interconnected, each represented as a set
• Each web represents one live range
• **Rename** and **rewrite** the code so that each global live range has a unique name

With **SSA Form**, a simpler process produces the same results

¹ See § 9.2 for *reaching definitions*, **DEFS**, and **USES**.
² Use Tarjan’s **disjoint set union-find algorithm** [Tarjan 75]
Rename Into Live Ranges (with SSA Form)

The allocator systematically renames values into live ranges

• The SSA construction creates a new name for each definition point, and propagates them forward to uses (and \( \phi \)-functions)
  – In SSA, we can build live ranges out of SSA names
• Create a singleton set for each SSA name
• At each \( \phi \)-function, union together the sets of all the arguments \(^1\)
• Each of the resulting sets represents a global live range

Without SSA, the process is more complex, but produces the same results

\(^1\) Use Tarjan’s disjoint set union-find algorithm [Tarjan 75]
Structure of a Chaitin-Briggs Allocator

- Rename Live Ranges
- Build Graph
  - Build an interference graph and use it to eliminate unnecessary copy operations
- Coalesce Copies
  - Estimate cost of spilling for each name
  - Try to color the graph in k or fewer colors
- Compute Spill Costs
- Simplify Graph
- Select Colors
  - If needed, spill uncolored values
- Spill
  - Rewrite VRs with PRs

Any copies coalesced?

Any uncolored live ranges?
Build the Interference Graph

What is an interference? (or conflict)

Two values interfere if they cannot occupy the same register

- $x$ and $y$ interfere if there exists an operation where both are live and they may have different values
- If $x$ and $y$ interfere, they cannot occupy the same register
  - Unless the compiled code plays bizarre tricks using XOR or the like
  - Or, the compiler can prove that $x$ and $y$ always have the same value in the region where they interfere. (See “copy coalescing”)

The compiler captures the conflict relationship in an interference graph

- Nodes in $G$ represent values, or live ranges
- Edges in $G$ represent individual interferences
  - For $x$ & $y$, $\langle x,y \rangle \in G$ iff $x$ & $y$ interfere
- A $k$-coloring of $G$ can be mapped into an allocation to $k$ registers $^{1}$

$^{1}$ Insight due to Lavrov ([242] in EaC2e)
Build the Interference Graph

To construct the graph

1. Compute **LIVEOUT** sets over live range names\(^1\)
   - Use the standard **LIVE** equations & an iterative solver
   - Renaming already rewrote the code into live range names

2. Iterate over each block, bottom to top
   - Track the current **LIVE** set
   - At each operation, except a register-to-register copy operation,
     - Add an edge from the defined LR to each LR in **LIVE**
     - Remove the defined LR from **LIVE**
     - Add the used LRs to **LIVE**

Data structures are important

- Algorithm needs to test for edge membership in the graph
- Algorithm needs to iterate over edges
  \[\implies\text{Needs both a bit-matrix & adjacency lists}^2\text{ (upper diagonal bit matrix)}\]

---

\(^1\) See § 8.6.1 and 9.2 for the computation of **LIVEOUT**

\(^2\) Implementations that ignored this advice have almost always been exceedingly slow.
Definition

Data-flow analysis (DFA) is a collection of techniques for compile-time reasoning about the run-time flow of values

• Compilers use the results of DFA to prove safety & identify opportunities
  – DFA is not an end unto itself

• DFA almost always involves building a graph
  – Control-flow graph, call graph, or graphs derived from them
  – Sparse evaluation graphs to model the flow of values (SSA viewed as a graph)

• DFA problems are usually formulated as a set of simultaneous equations
  – Sets attached to nodes and edges
  – Sets often have a lattice or semilattice structure

Both dominators and live variables are classical problems in DFA.
Computing Dominators with DFA

**Dominance is one of the simplest possible DFA problems**

- A node \( n \) dominates \( m \) iff \( n \) is on every path from \( n_0 \) to \( m \)
  - Every node dominates itself
  - \( n \)'s *immediate dominator* is its closest dominator, \( \text{IDOM}(n) \)

\[
\text{DOM}(n_0) = \{ n_0 \}
\]

\[
\text{DOM}(n) = \{ n \} \cup \bigcap_{p \in \text{preds}(n)} \text{DOM}(p)
\]

**Computing DOM**

- These simultaneous set equations define the data-flow problem
- Equations have a unique fixed point solution
- An iterative fixed-point algorithm will solve them quickly
  - Section 9.5.2 in EaC2e shows a data structure to speed up solving for \( \text{DOM} \)

\( n_0 \) has no \( \text{IDOM} \), by convention.
A variable is live at some point $p$ if its value can be used

- Domain is the set of variable names in the procedure
- Data-flow equations define $\text{LIVE}$ at the end of a block, $\text{LIVEOUT}$

**Initialization:** $\text{LIVEOUT}(n) = \emptyset$, $\forall n$

**Fixed-point equations:**

$$\text{LIVEOUT}(b) = \bigcup_{s \in \text{succs}(b)} (\text{UEVAR}(s) \cup (\text{LIVEOUT}(s) \cap \text{VARKILL}(s)))$$

In $\text{LIVE}$, information flows in a backward direction on the $\text{CFG}$

where

- $\text{UEVAR}(b)$ is the set of names used in $b$ before definition in $b$
- $\text{VARKILL}(b)$ is the set of names defined in $b$
Build the Interference Graph

**To construct the graph**

1. Compute **LIVEOUT** sets over live range names
   - Use the standard **LIVE** equations & an iterative solver
   - Renaming already rewrote the code into live range names

2. Iterate over each block, bottom to top
   - Track the current **LIVE** set
   - At each operation, except a register-to-register copy operation,
     - Add an edge from the defined **LR** to each **LR** in **LIVE**
     - Remove the defined **LR** from **LIVE**
     - Add the used **LRs** to **LIVE**

**Data structures are important**

- Algorithm needs to test for edge membership in the graph
- Algorithm needs to iterate over edges
  ⇒ Needs both a bit-matrix & adjacency lists \(^2\) (**upper triangular bit matrix**)

---

\(^1\) See § 8.6.1 and 9.2 for the computation of **LIVEOUT**

\(^2\) Implementations that ignored this advice have almost always been exceedingly slow.

---

There is a minor and odd controversy in the literature over using a hash map to represent the interference graph. George & Appel claim a serious improvement from a hash map representation. Others, including Chaitin, & Cooper, Harvey, & Torczon, have shown the opposite result. (up to 4,500x slower in our tests).

Pay attention to this exception.

Standard updates for local **LIVE** computation
Lab 2 renaming algorithm strikes again
Coalesce Unneeded Copies

The precise interference graph enables a powerful coalescing algorithm

• If the code contains $x \rightarrow y$, and $\langle x,y \rangle \not\in G$, then the allocator can combine $x \& y$ into a single live range
  → Remember the words: ... or, the compiler can prove that $x$ and $y$ always have the same value in the region where they interfere.”
  
  – If $x \& y$ do not otherwise interfere, then the copy is unneeded
  – Combine the two live ranges and update their interference
    → The edges for $xy$ are the union of the edges for $x$ and the edges for $y$
    → This update is approximate but not precise

LR $a$ can coalesce with one of LR $b$ or LR $c$

• After coalesce, the conservative update will make the new LR interfere with the remaining LR
  – $ab$ interferes with $c$ & $ac$ interferes with $b$
• The decision affects the possibilities
• Obvious update is conservative but not precise
Coalesce Unneeded Copies

The precise interference graph enables a powerful coalescing algorithm

- If the code contains $x \rightarrow y$, and $\langle x, y \rangle \notin G$, then the allocator can combine $x$ & $y$ into a single live range
  
  → Remember the words: ... or, the compiler can prove that $x$ and $y$ always have the same value in the region where they interfere.”
  
  - If $x$ & $y$ do not otherwise interfere, then the copy is unneeded
  
  - Combine the two live ranges and update their interference
    
    → The edges for $xy$ are the union of the edges for $x$ and the edges for $y$
    
    → This update is approximate but not precise

Assume that the compiler coalesces $a$ & $c$

- Update add $a$’s & $c$’s interferences to $ac$
  
  - $ac$ interferes with $b$
  
  - $b$ is not live at a definition of $ac$
  
  - $ac$ is not live at a definition of $b$

- A “from scratch” graph would not include $\langle ac, b \rangle$

Recall exception for copy operations
Coalesce Unneeded Copies

The precise interference graph enables a powerful coalescing algorithm

- If the code contains \( x \rightarrow y \), and \( <x,y> \notin G \), then the allocator can combine \( x \) & \( y \) into a single live range
  
  \( \rightarrow \) Remember the words: ... or, the compiler can prove that \( x \) and \( y \) always have the same value in the region where they interfere.”

- If \( x \) & \( y \) do not otherwise interfere, then the copy is unneeded
- Combine the two live ranges and update their interference
  
  \( \rightarrow \) The edges for \( xy \) are the union of the edges for \( x \) and the edges for \( y \)
  
  \( \rightarrow \) This update is approximate but not precise

And, so, the allocator iterates Build & Coalesce

- Strong technique for copy elimination
- Can significantly shrink the number of live ranges
  
  - In thesis, Briggs showed reductions by up to 1/3
- Initial pass of Build dominates cost of allocation

---

1 See also Cooper, Harvey, & Torczon, “How to Build an Interference Graph,” *Software—Practice and Experience*, 28(4), April 1998.
How do opportunities for coalescing arise?

Parameters at a procedure call

- Consider $x$ passed in two different positions at two different calls
  - Naïve code binds $x$ to two different PRs, which is not satisfiable
  - Proper code shape copies $x$ into the PR at each call site
- Leads to a string of copies before and after each call

Optimizing transformations insert copies for several reasons

- **LVN** replaces redundant op with a copy from the earlier definition
- Compensation code in the scheduler
- Translation out of **SSA** form replaces $\phi$-functions with copy operations
- Many transforms insert a new computation & use a copy to move it into place, leaving the old code for dead code elimination to remove

Copy coalescing aggressively eliminates unnecessary copy operations

---

1 See § 9.3 for information Static Single Assignment Form
2 See, for example, § 10.7.2 on Operator Strength Reduction
Any copies coalesced?

- **Rename Live Ranges**
- **Build Graph**
- **Coalesce Copies**
- **Compute Spill Costs**
- **Simplify Graph**
- **Select Colors**
- **Spill**

**Rename**

Build an interference graph and use it to eliminate unnecessary copy operations

**Estimate cost of spilling for each name**

Try to color the graph in k or fewer colors

If needed, spill uncolored values

Rewrite VRs with PRs
Computing Spill Costs

The allocator needs a spill-cost estimate for each LR to guide spill choice

• Accurate estimates lead to better spill decisions \( \text{(we hope)} \)
• Compute estimated spill cost for a specific LR as

  the sum over all defs in an LR of \( \text{frequency} \times \text{cost(\text{store})} \) plus
  the sum over all uses in an LR of \( \text{frequency} \times \text{cost(\text{load})} \)

  \( \text{frequency} \) is estimated as \( 10^d \) where \( d \) is the loop nesting depth of the reference to the LR

Multiple sharp corner cases

• Some live ranges have infinite spill costs
  – No value dies during the live range
  – Caveat: > \( k-1 \) infinite spill cost LRs together is a problem \( \text{(code shape?)} \)
• Multiple restores in a low-demand block & other local issues \[38\]
• Rematerialization is more complex in a global allocator \[55\]

In the patent filing (and not the papers), Chaitin suggests that an implementation of the allocator might defer the spill cost computation until the first time that it attempts to choose a spill candidate. This variant would reduce the number of LRs for which costs are computed and, thus, the cost.
Structure of a Chaitin-Briggs Allocator

- **Rename Live Ranges**
- **Build Graph**
- **Coalesce Copies**
- **Compute Spill Costs**
- **Simplify Graph**
- **Select Colors**
- **Spill**

**Rename**

Build an interference graph and use it to eliminate unnecessary copy operations

**Compute Spill Costs**

Estimate cost of spilling for each name

**Select Colors**

Try to color the graph in $k$ or fewer colors

**Spill**

If needed, spill uncolored values
Coloring the Graph

The heart of a Chaitin-Briggs allocator is the coloring algorithm

The coloring algorithm operates by carefully ordering the nodes & then attempting to color the nodes in that order

Key Observation

With $k$ colors, any node with fewer than $< k$ neighbors in the interference graph can be colored, no matter what colors its neighbors receive

– Cannot assign colors to its $< k$ neighbors in a way that uses all $k$ colors
– These *trivially colored nodes* can be colored in *any* order

Big Picture

• Compute an order that colors the most constrained nodes first
• Following the order, assign colors
• If some nodes fail to color, spill their live ranges and try again
  – Spilled LR becomes a collection of small LRs around defs & uses

Guarantees an eventual coloring
Simplify the Graph  

**(Briggs’ algorithm)**

Simplify constructs an ordering on the nodes of $G$

- Encodes the order into a stack that *Select Colors* will use
- Most constrained nodes on top of the stack

```
initialize the stack & spill list to empty
while $G$ is not empty
  if $\exists$ $n$ with degree($n$) < $k$
    then push $n$ onto the stack
  else
    pick a node $n$ to spill
    push $n$ onto the stack
    remove $n$ & its edges from $G$
```

- At the end of Simplify, $G$ is empty & every node is on the stack
  \[ \Rightarrow \] Data structures are ready for *Select Colors*

**Spill metric:** Chaitin suggested

\[ \text{spill cost} / \text{current degree} \]
Picking a Spill Candidate

When ∀ n ∈ G, \( n° ≥ k \), Simplify must pick a spill candidate

Chaitin’s Heuristic

- Minimize \( \text{spill cost} \div \text{current degree} \) among remaining LRs
- If \( x \) has a negative spill cost, spill it pre-emptively
  - Cheaper to spill it than to keep it in a register
- If \( x \) has an infinite spill cost, it cannot be spilled
  - No LR dies between \( x \)’s definition and its use
  - No more than \( k \) definitions since last value died (code shape safety valve)

\[
\text{cand} \leftarrow \text{LR}_0 \\
\text{for } 1 < i \leq n \\
\quad \text{if } ((\text{cost}\text{(cand)} / \text{degree}\text{(cand)}) > (\text{cost}\text{(LR}_i) / \text{degree}\text{(LR}_i))) \\
\quad \text{then cand} \leftarrow \text{LR}_i \\
\text{spill cand}
\]
Picking a Spill Candidate

When ∀ n ∈ G, n° ≥ k, Simplify must pick a spill candidate

Chaitin’s Heuristic

• Minimize \(\text{spill cost} \div \text{current degree}\) among remaining LRs
• If \(x\) has a negative spill cost, spill it pre-emptively
  – Cheaper to spill it than to keep it in a register
• If \(x\) has an infinite spill cost, it cannot be spilled
  – No LR dies between \(x\)’s definition and its use
  – No more than \(k\) definitions since last value died \((\text{code shape safety valve})\)

\[
\text{cand} \leftarrow \text{LR}_0 \\
\text{for } 1 < i \leq n \\
\quad \text{if } ((\text{cost}(\text{cand}) \div \text{degree}(\text{cand})) \\
\quad \quad \quad > (\text{cost}(\text{LR}_i) \div \text{degree}(\text{LR}_i))) \\
\quad \quad \quad \text{then } \text{cand} \leftarrow \text{LR}_i \\
\text{spill cand}
\]

\[
\text{cand} \leftarrow \text{LR}_0 \\
\text{for } 1 < i \leq n \\
\quad \text{if } (\text{cost}(\text{cand}) \times \text{degree(\text{LR}_i})) \\
\quad \quad \quad > (\text{cost}(\text{LR}_i) \times \text{degree}(\text{cand})) \\
\quad \quad \quad \text{then } \text{cand} \leftarrow \text{LR}_i \\
\text{spill cand}
\]

All those divisions are expensive, so cross multiply to save cycles

Nehalem data from scheduling lecture was 5 cycles versus 22 in double precision; 3 versus 41 in integer.
Select Colors

Select attempts to color $G$ in the order encoded in the stack by Simplify

• Most constrained nodes on top of the stack

while stack is not empty
  pop $n$ from the stack
  re-insert $n$ & its edges into $G$
  compute available colors from $n$’s neighbors
  if a color $c$ exists for $n$
    then assign $c$ to $n$
  else leave $n$ uncolored

• Assigns colors to the nodes of $G$, in pre-determined order
• Computes available colors by looking at the colors of $n$’s neighbors
  – Replaces degree with colorability, allowing spill candidates to color
• Easily proved fact: Only spill candidates can remain uncolored
Build an interference graph and use it to eliminate unnecessary copy operations

Estimate cost of spilling for each name

Try to color the graph in k or fewer colors

If needed, spill uncolored values
Spill

Spill inserts spill code for any uncolored register

• Chaitin-Briggs allocators spill entire live ranges
  – Insert a load before each use in the spilled LR
  – Insert a store after each def in the spilled LR
  – Converts a single LR into multiple tiny LRs with infinite spill cost

• This philosophy ensures that the “big loop” halts
  – If necessary, the code will devolve down to all tiny LRs & need few colors
  – As long as target machine has enough registers to hold operands for its most demanding operation, it will color
  – In practice, it will color long before that point — say 2 to 4 iterations

“Spill everywhere” is a weakness of allocators like Chaitin-Briggs

• Leads to spilling in areas of low pressure
• Provoked work on live-range splitting (another truly hard problem)
Structure of a Chaitin-Briggs Allocator

- **Rename Live Ranges**
  - Rename

- **Build Graph**
  - Build an interference graph and use it to eliminate unnecessary copy operations

- **Coalesce Copies**
  - Estimate cost of spilling for each name

- **Compute Spill Costs**
  - Try to color the graph in $k$ or fewer colors

- **Simplify Graph**

- **Select Colors**
  - If needed, spill uncolored values

- **Spill**
  - Rewrite VRs with PRs

Names from optimizer → Live range names ➔ Rename Live Ranges ➔ Live range names ➔ Build Graph ➔ Coalesce Copies ➔ Compute Spill Costs ➔ Simplify Graph ➔ Select Colors ➔ Spill ➔ Any copies coalesced? ➔ Any uncolored live ranges?
Example of Chaitin-Briggs Coloring

Assume $k = 3$

1 is the only node with degree < 3
Example of Chaitin-Briggs Coloring

Assume $k = 3$

Now, 2 & 3 have degree $< 3$
Example of Chaitin-Briggs Coloring

Assume $k = 3$

Now all nodes have degree $< 3$
Example of Chaitin-Briggs Coloring

Assume $k = 3$
Example of Chaitin-Briggs Coloring

Assume $k = 3$

Stack

Colors:

1: ☺️
2: 😄
3: 💙
Example of Chaitin-Briggs Coloring

Assume $k = 3$

Stack

Colors:
1: 
2: 
3: 

COMP 412, Fall 2018
Example of Chaitin-Briggs Coloring

Assume $k = 3$
Example of Chaitin-Briggs Coloring

Assume $k = 3$

Colors:
1:  
2:  
3:  

Stack

2
1
Example of Chaitin-Briggs Coloring

Assume $k = 3$
Example of Chaitin-Briggs Coloring

Assume $k = 3$
Chaitin’s algorithm, in the IBM PL.8 compiler, spilled when it could not find a trivially-colored live range:

- It would mark the LR to spill, remove it from \( G \), and continue with Simplify.
- At the end of Simplify, if any \( LR \) was marked to spill, it would insert spill code & start over with Rename.

It misses some easily colored graphs.

The “diamond” graph is 2-colorable. The first thing that Chaitin’s algorithm does is spill a node.
Simplify In Chaitin’s Allocator

Simplify constructs an ordering on the nodes of $G$

- Encodes the order into a stack that Select Colors will use
- Most constrained nodes on top of the stack

```plaintext
initialize the stack & spill list to empty
while $G$ is not empty
  if $\exists n$ with $n^\circ < k$
    then push $n$ onto the stack
  else
    pick a node $n$ to spill
    add it to the spill list
    remove $n$ & its edges from $G$
```

Uses degree as a proxy for colorability
Takes *trivially colored* nodes in any order

Only takes constrained node when forced
Chooses the node using its “spill metric”

- If any node is marked to spill, it spills, then repeats the whole process
  ⇒ Can overspill when it selects a node that does not help the problem

People make 2 major mistakes implementing Chaitin or Chaitin-Briggs. The first is to use one data structure for the graph, rather than two. The second is to iterate the big loop on each spill choice rather than simplifying the full graph and then invoking the Spill phase.
Brigg’s modification is simple.

- When no *trivially-colored* LR is found, pick the same node that Chaitin’s algorithm would and push it.
- In Select, try to color it. If that fails, spill it.
  
  → *Select finds colors for nodes that were marked to spill but do not reduce demand in a region of the code where demand > k.*

- This “optimistic coloring” can *k*-color more graphs.

Brigg’s modification gets the “diamond” graph in 2 colors.

Degree is a *loose upper bound* on colorability.

Brigg, Cooper, Kennedy, & Torczon, PLDI 89 (See also, TOPLAS 1994)
Simplify In Brigg’s Allocator

Simplify constructs an ordering on the nodes of $G$

- Encodes the order into a stack that Select Colors will use
- Most constrained nodes on top of the stack

```
initialize the stack to empty
while G is not empty
    if $\exists$ n with $n^\circ < k$
        then push n onto the stack
    else
        pick a node n to spill
        push it on the stack
        remove n & its edges from G
```

- Uses degree as a proxy for colorability
- Takes trivially colored nodes in any order
- Uses the same spill metric to select a node
- Still simplifies the entire graph

• If any node is marked to spill, it spills, then repeats the whole process

⇒ Can overspill when it selects a node that does not help the problem

People make 2 major mistakes implementing Chaitin or Chaitin-Briggs. The first is to use one data structure for the graph, rather than two. The second is to iterate the big loop on each spill choice rather than simplifying the full graph and then invoking the Spill phase.
Variations & Improvements

Better Spill Choice

• Bernstein et. all suggest repeating Simplify-Select-Spill with several different spill metrics & keeping the best allocation

• Tried three specific new metrics & found best-of-three always did as well as Chaitin; often beat Chaitin
  – Spill cost ÷ degree², spill cost ÷ area, and spill cost ÷ area², where area is the sum over the operations in the live range of |LIVE| at that operation
    → Conceptually, a Reimann sum of LR’s impact on demand for registers...
  – No single metric always won
  – Best of three always did as well or better than spill cost ÷ degree

• Briggs tried randomly renumbering the nodes and running the allocator 5 to 10 times (unpublished)
  – Achieved essentially the same results
  – Why? Because min is deterministic and renaming forces a different order. (min always breaks a tie in the same direction — the first value found)
Variations & Improvements

Spilling partial live ranges

• Bergner introduced *interference region spilling*
• Limits spilling to regions of high demand for registers

Splitting live ranges

• Simple idea — break up one or more live ranges into smaller parts
• Allocator can then use different PRs for distinct subranges
• Allocator can spill subranges independently *(use 1 spill location for all)*

Rematerialization

• Identify expressions whose arguments are always available
• Recognize when re-computation is cheaper than a load
  – Any `load`, plus some simple expressions
• Restore becomes recompute
Variations & Improvements

**Conservative Coalescing** ([55,56] in EaC2e)
- Only coalesce $x$ & $y$ if $xy$ has degree $< k$
- Invented for a specific purpose — use in a splitting allocator
- Never makes graph harder to color
- Refuses to coalesce some copies that are completely unneeded

**Iterative coalescing** ([158] in EaC2e)
- Use conservative coalescing as base case because it is “safe”
- Simplify graph until no *trivially-colored* nodes remain
- Coalesce & try again
- If coalescing does not create *trivially-colored* nodes, then spill

**Biased Coloring** ([55,56] in EaC2e)
- In Select, prefer colors that match copy-connected neighbors
Variations & Improvements

Spill Placement is an open issue and an opportunity

• Chaitin-Briggs spills a live range at each def and each use
  – Spill metric only considers placement as a numerical component of costs
  – Can insert spills into an inner loop or other frequently executed place
  – A global allocator might make global decisions about placement

• Bernstein et al. worked on spill placement in a single block
  – When possible, spill or restore a value only once per block
  – Avoid redundant spill code — “clean spilling”

• Can envision several approaches (never done, to my knowledge)
  – Work placement into the spill choice metric
  – Follow allocation with a code motion pass

The local allocator (lab 2) takes a wildly different approach

• Only spills in regions of high demand for registers
• All spill locations execute an equal number of times
Summary: Chaitin-Briggs Allocator

- Find live ranges, rename, compute \textbf{LIVE}
- Build the interference graph, \( G \)
- Fold unneeded copies 
  \( x \rightarrow y \) & \((x,y) \notin G \Rightarrow \text{coalesce } x \& y \)
- Estimate spill cost for each live range
- Remove nodes from \( G \)
- While stack is non-empty 
  pop \( n \), insert \( n \) into \( G \), & try to color \( n \)
- Spill any uncolored definitions & uses
Linear Scan Allocation

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.

Sun’s HotSpot server compiler uses a complete Chaitin-Briggs allocator [279].
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
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
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]
Building the Interference Graph

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

- In practice, building first graph dominates cost of allocation
- We can reduce that cost including in the interference graph only IRs involved in copy operations
  - Only include in the analysis things that can matter \((e.g., \text{Briggs semi-pruned SSA})\)
  - Only works with Chaitin-style coalescing \((i.e., \text{non conservative})\)

Experience

- Informally, it runs in about 66% of the time of the coalescing with the full graph
- That is enough improvement to matter

The allocator then needs the full graph for Simplify and Select
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 ($LR_x \rightarrow LR_y$)

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

Still, a promising idea

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

COMP 412, Fall 2018

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