Register Allocation

15-745 Optimizing Compilers
Spring 2007

Back end structure

- IR
- Instruction selector
- Register allocator
- TempMap
- Instruction scheduler
- Assem

After Instruction Selection

```
.text
.globl _main
_main:
  enter
  pushl %edi
  pushl %ebx
  pushl %esi
  movl $2, t7
  movl t7, t11
  imull t7, t11
  movl t11, t10
  imull $37, t10
  movl t10, t8
  movl t7, t17
  addl $1, t17
  movl $33, %eax
  cltd
  idivl t17
  movl %eax, t15
  movl t7, t14
  addl t15, t14
  movl t8, t13
  imull t14, t13
  movl t13, t8
  movl $78, t20
  negl t20
  movl t8, t19
  subl t20, t19
  movl t19, %eax
  popl %edi
  popl %ebx
  popl %esi
  leave
  ret
```
After Instruction Selection

Register allocator's job

Abstract View

Some terminology
A graph-coloring problem

Interference graph: an undirected graph where
- nodes = temps
- there is an edge between two nodes if their corresponding temps interfere

\[
\begin{align*}
v & \leftarrow 1 \\
w & \leftarrow v + 3 \\
x & \leftarrow w + v \\
u & \leftarrow v \\
t & \leftarrow u + x \\
w & \leftarrow t \\
u & \leftarrow u
\end{align*}
\]

History

For early architectures, register allocation was not very important.
Early work by Cocke (in 1971) proposed the idea that register allocation can be viewed as a graph coloring problem.
Chaitin was the first to implement this idea for the IBM 370 PL/1 compiler, in 1981.
In 1982, at IBM, Chaitin’s allocator was used for the PL/8 compiler for the IBM 801 RISC system.

History, cont’d

Motivated by the first MIPS architecture, Chow and Hennessy developed priority-based graph coloring in 1984.

Another popular algorithm for register allocation based on graph coloring is due to Briggs in 1992.

“top down” coloring

“bottom up” coloring
Steps in register allocation

- Build
- Color
- Spill

Building the interference graph

Given liveness information, we can build the interference graph (IG)

How?

Intuitively:
Two variables interfere if they overlap at some point in the program.

Algorithm:
At each point in program,
enter an edge for every pair of variables at that point

An optimized definition & algorithm for edges:
For each defining inst \( i \)
Let \( x \) be definition at inst \( i \)
For each variable \( y \) live at end of inst \( i \)
Insert an edge between \( x \) and \( y \)

Faster?
Better quality?

Building the interference graph

for each defining inst \( i \)
let \( x \) be temp defined at inst \( i \)
for all \( y \) in LIVE-IN of succ\((i)\)
insert an edge between \( x \) and \( y \)
A Better Interference Graph

```c
x = 0;
for(i = 0; i < 10; i++)
{
    x += i;
}
y = global;
y *= x;
for(i = 0; i < 10; i++)
{
    y += i;
}
```

What does the interference graph look like?

What's the minimum number of registers needed?

Live Ranges & Merged Live Ranges

A live range consists of a definition and all the points in a program (e.g. end of an instruction) in which that definition is live.

- How to compute a live range?

```
a = ... a = ...
... = a
```

Two overlapping live ranges for the same variable must be merged

Merging Live Ranges

Merging definitions into equivalence classes:

- Start by putting each definition in a different equivalence class
- For each point in a program
  - if variable is live, and there are multiple reaching definitions for the variable
  - merge the equivalence classes of all such definitions into one equivalence class

```
A = 2 (A_2)

(A,D) (A_2,B,C_1,D_1,D_2)
```

Merged live ranges are also known as “webs”
Example: Merged Live Ranges

A = ... (A₁) \[ \text{IF A goto L1} \]

B = ... (B₁) = A
D = B (D₂)

L₁:
C = ... (C₁)
D = A (D₁)

A = 2 (A₂)

= A
ret D

has two "webs"
makes register allocation easier

Steps in register allocation

Build

Color

"top down" priority coloring

Spill

Priority Coloring

Heuristics for priority allocation:
- highest priority allocated first

Ideas for priority functions?

v <− 1
w <− v + 3
x <− w + v
u <− v
v <− u + x
t <− w
t <− u

Priorities:
v: 4
w: 3
x: 2
u: 3
t: 2

Order:
v, w, u, x, t
Priority Coloring

Allocate in priority order
Another heuristic selects which register to assign to variable
- rotating registers
- lowest numbered register

Order: V, W, U, X, T

Graph coloring

Once we have the interference graph, we can attempt register allocation by searching for a K-coloring
This is an NP-complete problem (K≥3)*
But a linear-time simplification algorithm by Kempe (in 1879) tends to work well in practice

Kempe’s algorithm

Basic observation:
- given a graph that contains a node with degree less than K, the graph is K-colorable iff the graph without that node is K-colorable
- this is called the “degree<K” rule
So, step #1 of Kempe’s algorithm:
- iteratively remove nodes with degree<K

Steps in register allocation

Build
Color
Spill
“bottom up” Briggs-Chaitin
Simplify
Coalesce
Potential Spill
Select
Kempe’s algorithm, cont’d

If all nodes are removed by step #1, then the graph is K-colorable.
However, the degree<K rule does not always work, for example:

This graph is 3-colorable, but the degree<3 rule doesn’t work.

In step #1, each removed node should be pushed onto a stack.
– when all are removed, we pop each node and put it back into the graph, assigning a suitable color as we go.
In case we get stuck (i.e., there are no nodes with degree<K), we apply step #2:
– choose a node with degree≥K and optimistically remove it, and then continue.

Example

Stack

Example

Stack
**Another Example**

Now what?

**Be optimistic:**
- Put a node with degree $\geq k$ on stack
- Lose guarantee that anything we put on stack is colorable
- If we're lucky this node will still be colorable when popped from stack

**Be realistic:**
- If unlucky, this node will have to be spilled (allocated to memory)
- Mark as potential spill to avoid recomputation later

---

**Select**

Pop a node from the stack

Assign it a color that does not conflict with neighbors in interference graph

This will always be possible, unless the node is a potential spill

If it is not possible, mark as actual spill

---

**Steps in register allocation**

1. **Build**
2. **Color**
3. **Spill**

---

**Spilling to Memory**

**RISC Architectures**
- Only load and store can access memory
  - every use requires load
  - every def requires store
  - create new temporary for each location

**CISC Architectures**
- can operate on data in memory directly
  - makes writing compiler easier(?), but isn't necessarily faster
  - pseudo-registers inside memory operands still have to be handled
Spilling a use

For an instruction like
\[-t \leftarrow (u,v)\]
If \(u\) is marked as an actual spill, transform to
\[-u' := u\] \textit{(i.e., a load instruction)}
\[-t' \leftarrow (u',v)\]
where \(u'\) is a new temp
\(u\) and \(u'\) are special:
\(-u\) is spilled and thus \textit{unallocatable} \\
\(-u'\) is marked as \textit{unspillable}

Spilling a def

For an instruction like
\[-t \leftarrow (u,v)\]
If \(t\) is marked as an actual spill, transform to
\[-t' \leftarrow (u,v)\] \textit{(i.e., a store instruction)}
\[-t := t'\]
where \(t'\) is a new temp
\(t\) and \(t'\) are special:
\(-t\) is spilled and thus \textit{unallocatable} \\
\(-t'\) is marked as \textit{unspillable}

Spilled (unallocable) temps

Question: Where do the spilled temps get stored?
Answer: On the stack, in \textit{stack slots}

\begin{center}
\begin{tabular}{|c|c|}
\hline
\textit{ebp} & \textit{esp} \\
\hline
old & return addr \\
\hline
\ldots & \\
\hline
\textit{slot} \(n\) & \\
\hline
\ldots & \\
\hline
\textit{slot} \(1\) & \\
\hline
\textit{slot} \(0\) & \\
\hline
\end{tabular}
\end{center}

Each spilled temp should be allocated into a stack slot
The compiler can maintain a counter for the “next” slot number

To “mark” an actual spill, give it a slot number

Stack slots

In order to create the stack slots at run time, the prelude code needs to modify \%\textit{esp}

\begin{center}
\texttt{main:} \begin{align*}
\text{pushl} & \%\text{ebp} \\
\text{movl} & \%\text{esp}, \%\text{ebp} \\
\text{movl} & \%\text{edi}, t2 \\
\text{movl} & \%\text{ebx}, t3 \\
\text{movl} & \%\text{esi}, t4 \\
\text{subl} & \$((n\times4)), \%\text{esp}
\end{align*}
\end{center}

Note that the \texttt{subl} can be generated only after register allocation is finished
Spill code generation

The effect of spill code generation is to turn long live ranges into a bunch of tiny live ranges.
This introduces new temps.
Hence, register allocation must start over from scratch whenever spill code is generated.

What to Spill?

When choosing potential spill node want:
- A node that makes graph easier to color
  • Fewer spills later
- A node that isn’t “expensive” to spill
  • An expensive node would slow down the program if spilled
- We can apply heuristics both when choosing potential spill nodes and when choosing actual spill nodes
  • not required to spill node that we popped off stack and can’t color

A Spill Heuristic

Pick node (live range) $n$ that minimizes:

$$
\sum_{\text{def} \in n} 10^{\text{depth}(\text{def})} + \sum_{\text{use} \in n} 10^{\text{depth}(\text{use})}
\frac{\text{degree}(n)}{\text{degree}(n)}
$$

This heuristic prefers nodes that:
- Are used infrequently
- Aren’t used inside of loops
- Have a large degree

Could use any one of several other heuristics as well...
Rematerialization

An alternative to spilling

– Recompute value of variable instead of store/load to memory
– Example:

\[
\begin{align*}
v &\leftarrow 1 \\
w &\leftarrow v + 3 \\
x &\leftarrow w + v \\
u &\leftarrow v \\
t &\leftarrow u + x \\
\end{align*}
\]

\[
\begin{align*}
v &\leftarrow 1 \\
w &\leftarrow v + 3 \\
x &\leftarrow w + v \\
u &\leftarrow v \\
t &\leftarrow u + x \\
\end{align*}
\]

Build Take Two

\[
\begin{align*}
v &\leftarrow 1 \\
w_1 &\leftarrow v + 3 \\
w_2 &\leftarrow w_1 \\
x &\leftarrow w_2 + v \\
u &\leftarrow v \\
t &\leftarrow u + x \\
w_3 &\leftarrow w_2 \\
\end{align*}
\]

Recalculate interference graph

Simplify->Select

\[
\begin{align*}
v &\leftarrow 1 \\
w_1 &\leftarrow v + 3 \\
w_2 &\leftarrow w_1 \\
x &\leftarrow w_2 + v \\
u &\leftarrow v \\
t &\leftarrow u + x \\
w_3 &\leftarrow w_2 \\
\end{align*}
\]

Another Example

\[
\begin{align*}
t_1 &\leftarrow () \\
t_2 &\leftarrow () \\
t_3 &\leftarrow (t_1, t_2) \\
t_4 &\leftarrow (t_1, t_3) \\
t_5 &\leftarrow (t_1, t_2) \\
t_6 &\leftarrow (t_4, t_5) \\
\end{align*}
\]

Assume 2 machine registers, \{r1,r2\}. Assume \t4 may not be in r1. Then we have the interference graph.
Simplification steps... 

Choosing potential spills... 

Completing simplification
Selecting colors...

Actual spills...

Select complete!

Spill code generation...

Notice: Live ranges for t3 and t4 have been chopped up into lots of small live ranges
...and start over!

```
r2 <- ()
r1 <- ()
r1 <- (r2,r1)
slot0 := r1
r1 := slot0
r1 <- (r2,r1)
slot1 := r1
r1 <- (r2,r1)
r2 := slot1
r1 <- (r2,r1)
```

**Move Coalescing**

Eliminate moves by assigning the src and dest to the same register

```
movl t1,t2
addl t3,t2
```

```
movl %eax,%eax
addl %edx,%eax
```

When can we coalesce t1 and t2?

How can we modify our interference graph to do this?

---

**Steps in register allocation**

Build → Color → Spill

```
"bottom up"
```

Briggs-Chaitin

Coalesce → Potential Spill → Select

---

**Example**

```
v <- 1
w <- v + 3
x <- w + v
u <- v
t <- u + x
    <- w
    <- t
    <- u
```

First compute live ranges...

...then construct interference graph
**Example**

\[
\begin{align*}
v & \leftarrow 1 \\
w & \leftarrow v + 3 \\
x & \leftarrow w + v \\
u & \leftarrow v \\
t & \leftarrow u + x \\
& \leftarrow w \\
& \leftarrow t \\
& \leftarrow u \\
\end{align*}
\]

\(u\) and \(v\) are special:

A move whose source is not live-out of the move is a candidate for coalescing

\[v \leftarrow 1\]
\[w \leftarrow v + 3\]
\[x \leftarrow w + v\]
\[u \leftarrow v\]
\[t \leftarrow u + x\]
\[\leftarrow w\]
\[\leftarrow t\]
\[\leftarrow u\]

**Is Coalescing Always Good?**

\[\begin{align*}
\text{move edge} \\
\text{u} \\
\text{x} \\
\text{v} \\
\text{a} \\
\text{b} \\
\text{u} \\
\text{v} \\
\end{align*}\]

And the winner is?

2 colorable

3 colorable

**When should we coalesce?**

Always
- If we run into trouble start un-coalescing
  - no nodes with degree \(< k\), see if breaking up coalesced nodes fixes
- yuck

Only if we can prove it won’t cause problems
- Briggs: Conservative Coalescing
- George: Iterated Coalescing

\[y \leftarrow u \leftarrow x \leftarrow v \leftarrow a \leftarrow b\]

Briggs: Conservative Coalescing

- Can coalesce \(u\) and \(v\) if:
  \[-(\# \text{ of neighbors of } uv \text{ with degree } \geq k) < k\]
- Why?
  - Simplify pass removes all nodes with degree \(< k\)
  - \# of remaining nodes \(< k\)
  - Thus, \(uv\) can be simplified

\[y \leftarrow u \leftarrow x \leftarrow v \leftarrow a \leftarrow b\]

What does Briggs say about

- \(k = 3?\)
- \(k = 2?\)
George: Iterated Coalescing

Can coalesce $u$ and $v$ if foreach neighbor $t$ of $u$
  • $t$ interferes with $v$, or,
  • degree of $t < k$

Why?
  - let $S$ be set of neighbors of $u$ with degree $< k$
  - If no coalescing, simplify removes all nodes in $S$, call that graph $G^1$
  - If we coalesce we can still remove all nodes in $S$, call that graph $G^2$
  - $G^2$ is a subgraph of $G^1$

Why Two Methods?

• Why not?
  • With Briggs, one needs to look at all neighbors of $a$ & $b$
  • With George, only need to look at neighbors of $a$.

So:
  - Use George if one of $a$ & $b$ has very large degree
  - Use Briggs otherwise

Optimistic Coalescing

Aggressively coalesce
If coalesced node spills, uncoalesce

Will this always work?
Steps in register allocation

- **top-down**
  - Prioritize
  - Select

- **bottom-up**
  - Simplify
  - Coalesce
  - Potential Spill
  - Select

Which one is better?

Alternative Allocators

Graph allocator, as described, has issues
- What are they?

Alternative: Single pass graph coloring
- Build, Simplify, Coalesce as before
- In select, if can’t color with register, color with stack location
  - Keep going
  - Requires second, reload phase
    - “fixes” spilled variables
    - Might require that we reserve a register
    - Can get messy

Claim: Does a pretty good job
- Why?
  - Key is order nodes are colored (top-down)

Advantages? Disadvantages?

Alternative Allocators

Local/Global Allocation
- Allocate “local” pseudo-registers
  - Lifetime contained within basic block
  - Register sufficiency no longer NP-Complete!
- Allocate global pseudo-registers
  - Single pass global coloring
  - Coloring heuristic may reverse local allocation
- Reload pass to fix spills (allocator does not generate spill code)
- Can also do global then local
- Advantages? Disadvantages?

In Chaitin’s words

“…since I was a mathematician, the register allocation kept getting simpler and faster as I understood better what was required. I preferred to base algorithms on a simple, clean idea that was intellectually understandable rather than write complicated ad hoc computer code…”

So I regard the success of this approach, which has been the basis for much future work, as a triumph of the power of a simple mathematical idea over ad hoc hacking. Yes, the real world is messy and complicated, but one should try to base algorithms on clean, comprehensible mathematical ideas and only complicate them when absolutely necessary. In fact, certain instructions were omitted from the 801 architecture because they would have unduly complicated register allocation…”

— G. Chaitin, 2004
Avoiding Spills

Bottom-up vs Top-down: Speed

Bottom-up vs Top-down: Size

Complexity of Register Allocation
Complexity of Register Allocation

Graph color is NP-complete
   – what does this tell us about register allocation?

Given arbitrary graph can construct program with matching interference graph
   – simply determining if spilling is necessary is therefore NP-complete... or is it?

Can exploit structure of reducible program

Complexity of optimizing spill code?
   – NP-complete even without control flow

Complexity of optimal coalescing?
   – NP-complete

---


Complexity of local register allocation?
   – linear algorithm for register sufficiency

SSA Form?
   – interference graph is turns out to be both perfect and chordal
      • can color in linear time
   – BUT all bets are off after SSA elimination

---


Complexity of optimizing spill code?
   – NP-complete even without control flow

Complexity of optimal coalescing?
   – NP-complete