Graph Coloring Register Allocation

Last Time

- Chaitin et al.
- Briggs et al.

Today

- Finish Briggs et al. basics
- An improvement: rematerialization

Rematerialization

Some expressions are especially simple to recompute:

- Operands are constant
  (though not necessarily known)
- Operands are available globally

Chaitin calls these expressions *never-killed*

Typical examples include:

- Constant
- Constant + frame pointer
- Load of constant parameter
- Load from constant pool
- Access through *display*

Rematerialization

We should recognize that these are cheaper *before* we try to color

Helps resolve spill choices correctly

Original	**Ideal**
$p \leftarrow 123$	
	$p \leftarrow 123$
$y \leftarrow y + [p]$	$y \leftarrow y + [p]$
	$p \leftarrow 123$
$p \leftarrow p + 1$	$p \leftarrow p + 1$

What we get

Chaitin	Chow - Spitting
$p \leftarrow 123$ *spill* p	$p \leftarrow 123$ *spill* p
reload p $y \leftarrow y + [p]$	*reload* p $y \leftarrow y + [p]$
	reload p
reload p $p \leftarrow p + 1$ *spill* p	$p \leftarrow p + 1$

Live Ranges and Values

Chaitin's allocator works with *live ranges*

A live range may include many *values*, connected by common uses

A value corresponds to a single definition, including the merge of two values

You should be thinking:

Hmmm, smells like SSA or something, . . .

Chaitin's allocator can handle rematerializing a live range with a single value

The Plan

To discover and isolate rematerializable values:

- Find values
  (use pruned SSA graph)
- Tag values according to definition
- Propagate tags
  (use *sparse simple constant* algorithm)
- Union connected values if tags are identical

A Lattice

Lattice elements may have one of three types:

$\top$ No information is known – a value defined by a copy instruction or a ϕ-node has an initial tag of $\top$

inst A value defined by an appropriate instruction (*never-killed*) should be rematerialized – the value's tag is just a pointer to the instruction

$\bot$ Value cannot be rematerialized – values defined by "inappropriate" instructions are immediately tagged with $\bot$

$\top$

$inst_1$ $inst_2$ $inst_3$ $inst_n$

$\bot$

The Meet Operation

The meet operation $\sqcap$ is

any	$\sqcap$	$\top$	$=$	any	
any	$\sqcap$	$\bot$	$=$	$\bot$	
$inst_i$	$\sqcap$	$inst_j$	$=$	$inst_i$	if $inst_i = inst_j$
$inst_i$	$\sqcap$	$inst_j$	$=$	$\bot$	if $inst_i \neq inst_j$

$inst_i = inst_j$ compares the instructions on an operand-by-operand basis

Since our instructions have only 2 operands, asymptotic complexity is not affected

Conservative Coalescing Example

We remove splits where the source and destination values have the same tag

SSA	*Splits*	*Minimal*
$p_0 \leftarrow 123$	$p_0 \leftarrow 123$	$p_0 \leftarrow 123$
$y \leftarrow y + [p_0]$	$y \leftarrow y + [p_0]$	$y \leftarrow y + [p_0]$
	$p_1 \leftarrow p_0$	$p_{12} \leftarrow p_0$
$p_1 \leftarrow \phi(p_0, p_2)$ $p_2 \leftarrow p_1 + 1$	$p_2 \leftarrow p_1 + 1$ $p_1 \leftarrow p_2$	$p_{12} \leftarrow p_{12} + 1$

Similarly, we remove copies if the source and destination values have identical *inst* tags

Undoing Splits

Briggs claims that they end up with splits placed *perfectly*

All never-killed values are isolated with the minimum number of splits

Nevertheless, some of the splits (copies) are never required

- Conservative coalescing
- Biased coloring

Briggs' phases

renumber Find all distinct live ranges and number them uniquely

build Construct the interference graph

coalesce For each copy where the source and destination live ranges don't interfere, union the two live ranges and remove the copy

spill costs Estimate the dynamic cost for spilling each live range

simplify Repeatedly remove nodes with degree $< k$ from the graph and push them on a stack. When necessary, choose spill candidates and push them on the stack

select Reassemble the graph with nodes popped from the graph. As each node is added to the graph, choose a color differing from neighbors in the graph. If no color is available, the node is left uncolored

spill code Spill uncolored nodes by inserting a load or store at each use or definition

Conservative Coalescing

Two rounds of coalescing

1. Coalesce copies (*subsumption*)
2. Conservatively coalesce splits

Note that each round of coalescing may be repeated several times

Conservative coalescing removes splits when it doesn't hurt colorability

The conservative approximation is to remove a split if the resulting live range has $< k$ neighbors of degree $\geq k$

Biased Coloring

We also modify *select* slightly

Select (re)assembles the graph 1 node at a time

As each node is added to the graph, we choose a color that differs from any neighbor

With *biased coloring,* we try to choose a color that helps remove a split

An additional refinement

limited lookahead Avoid choosing colors that an uncolored partner can't use

Results

Dynamic measurements on 70 routines (mostly SPEC Fortran)

On a (simulated) machine with 16 integer registers and 16 floating-point registers

Counting loads, stores, copies, ldi's, addi's contributing to spill costs

- New $<$ Old – 28 routines
- New $>$ Old – 2 routines

The new allocator is typically slightly slower (compile time) due to

- Propagation of tags
- Conservative coalescing

Occasionally, the new allocator is faster because of simplified coalescing

Rematerialization

A natural, low-cost extension to Chaitin's ideas on rematerialization

Handles complex live ranges that are wholly or partially rematerializable

Especially significant to splitting allocators (*e.g., Chow*)

- A simple adaptation of Wegman and Zadeck's sparse simple constant propagation algorithm
- Other required engineering changes
- Results showing that significant opportunities for rematerialization occur in practice

Next Time

Read: Traub, Holloway, and Smith, "Quality and Speed in Linear-scan Register Allocation," *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, June 1998, pp. 142-151.