Procedure Optimization (wrap-up) & Register Allocation

CS2210 Lecture 21

Procedure Call Optimization

- Procedure calls can be costly
  - direct call costs: call, return, argument & result passing, stack frame maintenance
  - indirect call costs: (opportunity cost) damage to intraprocedural analysis of caller and callee
- Optimization techniques
  - hardware support
  - inlining
  - tail call optimization
  - interprocedural analysis
  - procedure specialization

Inlining (aka Procedure Integration)

- Replace call with body of callee
  - insert assignments for actual/formal mapping
  - use copy propagation to eliminate copies
  - manage variable scoping correctly!
    - eg. rename local variables or tag names with scopes
- Pros & cons
  - eliminates call overhead, parameter passing and result returning overheads
  - can optimize callee in context of caller and vice versa
  - can slow down compilation & increase code space
Implementation Issues

- Within compilation unit or across?
- Caller and callee in same language?
  - Parameter passing conventions may be different
- Should copies of inlined functions be kept?
  - Should we compile a copy?
- Should recursive procedures be inlined?

What & where should be inlined?

- Considerations
  - callee size
  - Call frequency
  - Benefit from inlining
- Criteria
  - Estimated or actual call frequencies
  - May do "inline trial"
    - Inline and optimize to check for benefit if not big enough do not actually inline
- In practice: Profile-based inlining is much better than static estimates
  - Can get very good speedups (Ayers et al.) found 1.3 average up to 2
  - No conclusive impact on i-cache behavior

Tail Call Elimination

- Tail call = last thing executed before return is a call
  - Return f(n) is tail call
  - Return n * f(n-1) is not
- Can jump to callee rather than call
  - Splice out on stack frame creation and tear down (callee reuses caller’s frame & return address)
  - Effect on debugging
Tail Recursion Elimination

- Tail call is self-recursive
- can turn recursion into iteration
- Extremely important optimization for functional languages (e.g., Scheme) since all iteration is expressed recursively

Implementation
- replace call by parameter assignment
- branch to beginning of procedure body
- eliminate return following the recursive call

Example

```c
void insert_node(int n, struct node* l)
{
  if (n > l->value)
    if (l->next == nil) make_node(l,n);
    else insert_node(n,l->next);
}

void insert_node(int n, struct node*l)
{
  loop:
    if (n>l->value)
      if (l->next == nil) make_node(l,n);
      else {l := l->next; goto loop;}
}
```

Leaf Routine Optimization

- Goal:
  - simplify prologue and epilogue code for procedures that do not call others
  - e.g. don't have to save / restore caller-saved registers
  - works only if there are no calls at all (otherwise procedure not a leaf)
Shrink-Wrapping

- Generalization of leaf procedure optimization
  - try to move prologue and epilogue code close to call to execute it only when necessary

Register Allocation

Reading & Topics

- Chapter 16
- Topics
  - Register allocation methods
Problem

- Assign machine resources (registers & stack locations) to hold run-time data
- Constraint
  - simultaneously live data must be allocated to different locations
- Goal
  - minimize overhead of stack loads & stores and register moves

Solution

- Central insight: can be formulated as graph coloring problem
  - Chaitin-style register allocation (1981) for IBM 390 PL/I compiler
  - represent interference (= simultaneous liveness) as graph
    - color with minimal number of colors
- Alternative
  - bin-packing (used in DEC GEM compiler for Alpha) equally good in practice

Interference Graph

- Data structure to represent simultaneously live objects
- nodes are units of allocation
  - variables
  - better: webs
- edges represent simultaneously live property
  - symmetric, not transitive
Global Register Allocation Algorithm

- Allocate objects that can be register allocated (variables, temporaries that fit, large constants) to symbolic registers \( s_1 \) \( \ldots \) \( s_n \).
- Determine which should be register allocation candidates (simplest case all).
- Construct interference graph
  - allocatable objects and available registers are nodes
  - use arcs to indicate interference and other conflicts (e.g., floating point values and integer registers)
- Construct \( k \)-coloring \( k \) = number available registers
- Allocate object to register of same color

Example

\[
\begin{align*}
x &:= 2 \\
y &:= 4 \\
w &:= x+y \\
z &:= x=1 \\
u &:= x*y \\
x &:= z^2
\end{align*}
\]

assume \( y \) & \( w \) dead on exit

Webs

- webs = maximal intersecting du-chains for a variable
  - separates different uses of same variable, e.g., \( i \) used as loop index in different loops
  - useful when no SSA-form is used (SSA form achieves same effect automatically)
Web Example

```
def x
use y
```

```
def y
use x
```

```
def x
use y
```

Constructing and Representing the Interference Graph

- Construction alternatives:
  - as side effect of live variables analysis (when variables are units of allocation)
  - compute du-chains & webs (or ssa form), do live variables analysis, compute interference graph
- Representation
  - adjacency matrix: $A[\min(i,j), \max(i,j)] = true$ iff (symbolic) register $i$ adjacent to $j$ (interferes)

Adjacency List

- Used in actual allocation
- $A[i]$ lists nodes adjacent to node $i$
- Components
  - color
  - disp: displacement (in stack for spilling)
  - spcost: spill cost (initialized 0 for symbolic, infinity for real registers)
  - nints: number of interferences
  - adjnds: list of real and symbolic registers currently interfere with $i$
  - rmvadj: list of real and symbolic registers that interfered with $i$ but have been removed
Register Coalescing

- Goal: avoid unnecessary register to register copies by coalescing register
  - ensure that values are in proper argument registers before procedure calls
  - remove unnecessary copies introduced by code generation from SSA form
  - enforce source / target register constraints of certain instructions (important for CISC)

- Approach:
  - search for copies $s_i := s_j$ where $s_i$ and $s_j$ do not interfere (may be real or symbolic register copies)

Computing Spill Costs

- Have to spill values to memory when not enough registers can be found (can't find $k$-coloring)

- Why webs to spill?
  - least frequently accessed variables
  - most conflicting

- Sometimes can rematerialize instead:
  - = recompute value from other register values instead of store / load into memory (Briggs: in practice mixed results)

Spill Cost Computation

- $defwt \times \sum 10^{depth(def)} + usewt \times \sum 10^{depth(use)} - copywt \times \sum 10^{depth(copy)}$

- $defwt / usewt / copywt$ costs relative weights assigned to instructions
- $def$, $use$, $copy$ are individual definitions /uses/ copies
- frequency estimated by loop nesting depth
Coloring the Graph

Assume 3 registers available

Graph Pruning

- Improvement #1 (Chaitin, 1982)
  - Nodes with <k edges can be colored after all other nodes and still be guaranteed registers
  - So remove <k-degree nodes first
    - this may reduce the degree of remaining graph and make it colorable

Algorithm

while interference graph not empty
  while there is a node with <k neighbors
    remove it from graph, push on stack
  if all remaining nodes have >= k neighbors, then
    blocked:
    - pick a node to spill (lowest cost)
    - remove from graph, add to spill set
  if any nodes in spill set:
    insert spill codes for all spilled nodes, reconstruct interference graph and start over
while stack not empty
  pop node from stack, allocate to register
Coloring the Graph with pruning

weight order:
c
da2
d2
b
al
e
1. Assume 3 registers available
2. Assume 2 registers available

An Annoying Case

If only 2 register available: blocked must spill!

Improvement #2: blocked != spill (Briggs et al. 1989)

- Idea: just because node has k neighbors doesn’t mean it will be spilled (neighbors can have overlapping colors)
- Algorithm: like Chaitin, except
  - when removing blocked node, just push onto stack (“optimistic spilling”)
  - when done removing nodes, pop nodes off stack and see if they can be allocated
  - really spill if cannot be allocated at this stage
**Improvement #3: Priority-based Coloring (Chow & Hennessy 1984)**

- **Live-range splitting**
  - when variable cannot be register-allocated, split into multiple subranges that can be allocated separately
  - move instructions inserted at split points
  - some live ranges in registers, some in memory
  - selective spilling
- Based on **variable live ranges**
  - can result in more conservative interference graph than webs
  - live range = set of basic blocks a variables is live in

---

**Live Range Example**

\[
\begin{align*}
  x &:= a + b \\
  y &:= x + c \\
  \text{if } y = 0 \text{ goto L1} \\
  z &:= y + d \\
  w &:= z \\
  \text{L1:} \\
  \text{variables } x, y, z, w
\end{align*}
\]

---

**Improvement #4: Rematerialization**

- Idea: instead of reloading value from memory, **recompute** instead if recomputation cheaper than reloading
- Simple strategy choose rematerialization over spilling if
  - can recompute a value in a single instruction, and
  - all operands will always be available
- Examples
  - constants, address of a global, address of variable in stack frame
Evaluation

- Rematerialization
  - showed a reduction of -26% to 33% in spills
- Optimistic spilling
  - showed a reduction of -2% to 48%
- Priority-based coloring
  - may often not be worthwhile
  - appears to be more expensive in practice than optimistic spilling