Lecture 12: Optimization

COS 598C – Advanced Compilers

Prof. David August
Department of Computer Science
Princeton University

Where are we?

- Analysis
  - Control Flow/Predicate
  - Dataflow
  - SSA
- Optimization

Optimization

- Make the code run faster on the target processor
  - My favorite topic !!
  - Anything goes
    - Look at benchmark kernels, what’s the bottleneck??
    - Invent your own optimizations (easier and harder than you think)
- Classes of optimization
  1. Classical (machine independent)
    - Reducing operation count (redundancy elimination)
    - Simplifying operations
    - Generally good for any kind of machine
  2. Machine specific
    - Peephole optimizations
    - Take advantage of specialized hardware features
  3. ILP enhancing
    - Increasing parallelism
    - Possibly increase instructions
Classical Optimizations

- Operation-level – 1 operation in isolation
  - Constant folding, strength reduction
- Dead code elimination (global, but 1 op at a time)
- Local/Global – Pairs of operations
  - Constant propagation
  - Forward copy propagation
  - Backward copy propagation
  - CSE
  - Constant combining
  - Operation folding
- Loop – Body of a loop
  - Invariant code removal
  - Global variable migration
  - Induction variable strength reduction
  - Induction variable elimination

Caveat

- Traditional compiler class
  - Sophisticated implementations of optimizations, efficient algorithms
  - Spend entire class on 1 optimization
- For this class – Go over concepts of each optimization
  - What it is
  - When can it be applied (set of conditions that must be satisfied)

Static Single Assignment (SSA)

Static Single Assignment Advantages:

- Less space required to represent def-use chains. For each variable, space is proportional to uses + defs.
- Eliminates unnecessary relationships:
  
  for i = 1 to N do A[i] = 0
  for i = 1 to M do B[i] = 1

  - No reason why both loops should be forced to use same register to hold index register.
  - SSA renames second i to new register which may lead to better register allocation.
- SSA form make certain optimizations quick and easy — dominance property.
  - Variables have only one definition - no ambiguity.
  - Dominator information is encoded in the assignments.
Dominance property of SSA form: definitions dominate uses

- If $x$ is $i$th argument of $\phi$-function in node $n$, then definition of $x$ dominates $i$th predecessor of $n$.
- If $x$ is used in non-$\phi$ statement in node $n$, then definition of $x$ dominates $n$.

Dead Code Elimination

Given $d$: $t = x \text{ op } y$

- $t$ is live at end of node $d$ if there exists path from end of $d$ to use of $t$ that does not go through definition of $t$.
- If program not in SSA form, need to perform liveness analysis to determine if $t$ live at end of $d$.
- If program is in SSA form:
  - cannot be another definition of $t$
  - if there exists use of $t$, then path from end of $d$ to use exists, since definitions dominate uses.
  - every use has a unique definition
  - $t$ is live at end of node $d$ if $t$ is used at least once

Algorithm:

```
WHILE (for each temporary $t$ with no uses && statement defining $t$ has no other side-effects) DO
    delete statement definition $t$
```

1: $r1 = 5$
2: $r2 = 10$
3: branch $r3 > r2$
4: $r2' = r2 + 15$
5: $r4 = r3 + X$
6: $r2'' = \Phi (r2', r2)$
7: $M[r4] = r2''$

Dead Code Elimination
Dead Code Elimination

- Remove any operation who’s result is never consumed
- Rules
  - X can be deleted
    - no stores or branches
  - DU chain empty or dest register not live
- This misses some dead code!!
  - Especially in loops
  - Critical operation
    - store or branch operation
  - Any operation that does not directly or indirectly feed a critical operation is dead
  - Trace UD chains backwards from critical operations
  - Any op not visited is dead

\[
\begin{align*}
r_1 &= 3 \\
r_2 &= 10 \\
r_4 &= r_4 + 1 \\
r_7 &= r_1 * r_4 \\
r_2 &= 0 \\
r_3 &= r_3 + 1 \\
r_3 &= r_2 + 1 \\
\text{store (r1, r3)}
\end{align*}
\]

Constant Folding

- Simplify 1 operation based on values of src operands
  - Constant propagation creates opportunities for this
- All constant operands
  - Evaluate the op, replace with a move
    - \( r_1 = 3 * 4 \rightarrow r_1 = 12 \)
    - Don't evaluate excepting ops !, what about floating-point?
  - Evaluate conditional branch, replace with BRU ornoop
    - if (1 < 2) goto BB2 \rightarrow BRU BB2
    - if (1 > 2) goto BB2 \rightarrow convert to a noop
- Algebraic identities
  - \( r_1 = r_2 + 0, r_2 - 0, r_2 | 0, r_2 \wedge 0, r_2 << 0, r_2 >> 0 \)
  - \( r_1 = 1 = 0 \)
  - \( r_1 = r_2 * 1, r_2 / 1 \)
  - \( r_1 = r_2 \)

Strength Reduction

- Replace expensive ops with cheaper ones
  - Constant propagation creates opportunities for this
- Power of 2 constants
  - Multiply by power of 2, replace with left shift
    - \( r_1 = r_2 * 8 \rightarrow r_1 = r_2 << 3 \)
  - Divide by power of 2, replace with right shift
    - \( r_1 = r_2 / 4 \rightarrow r_1 = r_2 >> 2 \)
  - Remainder by power of 2, replace with logical and
    - \( r_1 = r_2 \text{ REM } 16 \rightarrow r_1 = r_2 \& 15 \)
- More exotic
  - Replace multiply by constant by sequence of shift and adds/subs
    - \( r_1 = r_2 * 6 \)
      - \( r_{100} = r_2 << 2; r_{101} = r_2 << 1; r_1 = r_{100} + r_{101} \)
      - \( r_1 = r_2 * 7 \)
      - \( r_{100} = r_2 << 3; r_1 = r_{100} - r_2 \)
Optimize this applying

1. constant folding
2. strength reduction
3. dead code elimination

Constant Propagation

- Forward propagation of moves of the form
  - \( r \times L \) (where \( L \) is a literal)
  - Maximally propagate
  - Assume no instruction encoding restrictions
- When is it legal?
  - SRC: Literal is a hard coded constant, so never a problem
  - DEST: Must be available
    - Guaranteed to reach
    - May reach not good enough

Simple Constant Propagation

Given \( d: t = c, c \) is constant Given \( u: x = t \) or \( b \)

- if program not in SSA form:
  - need to perform reaching definition analysis
  - use of \( t \) in \( u \) may be replaced by \( c \) if \( d \) reaches \( u \) and no other definition of \( t \) reaches \( u \)
- if program is in SSA form:
  - \( d \) reaches \( u \), since definitions dominate uses, and no other definition of \( t \) exists on path from \( d \) to \( u \)
  - \( d \) is only definition of \( t \) that reaches \( u \), since it is the only definition of \( t \).
  - any use of \( t \) can be replaced by \( c \)
  - any \( \phi \)-function of form \( v = \phi(c_1, c_2, \ldots, c_n) \), where \( c_i = c \), can be replaced by \( v = c \)
Local Constant Propagation

- Consider 2 ops, X and Y in a BB, X is before Y
  - 1. X is a move
  - 2. src1(X) is a literal
  - 3. Y consumes dest(X)
  - 4. There is no definition of dest(X) between X and Y
  - 5. No danger betw X and Y
    - When dest(X) is a Macro reg, BRL destroys the value

r1 = 5
r2 = '_x'
r3 = 7
r4 = r4 + r1
r1 = r1 + r2
r1 = r1 + 1
r3 = 12
r8 = r1 - r2
r9 = r3 + r5
r3 = r2 + 1
r10 = r3 - r1

Global Constant Propagation

- Consider 2 ops, X and Y in different BBs
  - 1. X is a move
  - 2. src1(X) is a literal
  - 3. Y consumes dest(X)
  - 4. X is in a_in(BB(Y))
  - 5. Dest(x) is not modified between the top of BB(Y) and Y
  - 6. No danger betw X and Y
    - When dest(X) is a Macro reg, BRL destroys the value

Class Problem

Optimize this applying
1. constant propagation
2. constant folding
3. strength reduction
4. dead code elimination

```
\[ \begin{align*}
  r1 &= 0 \\
  r2 &= 10 \\
  r4 &= 1 \\
  r7 &= r1 + 4 \\
  r6 &= 8 \\
  r2 &= 0 \\
  r3 &= r2 / r6 \\
  r3 &= r4 + r6 \\
  r3 &= r3 + r2 \\
  r2 &= r2 + r1 \\
  r6 &= r7 + r6 \\
  r1 &= r1 + 1 \\
  \text{store} (r1, r3)
\end{align*} \]```
Forward Copy Propagation

- Forward propagation of the RHS of moves
  - \( r_1 = r_2 \)
  - ...
  - \( r_4 = r_1 + 1 \rightarrow r_4 = r_2 + 1 \)
- Benefits
  - Reduce chain of dependences
  - Eliminate the move
- Rules (ops X and Y)
  - X is a move
  - src1(X) is a register
  - Y consumes dest(X)
  - X.dest is an available def at Y
  - X.src1 is an available expr at Y

Backward Copy Propagation

- Backward propagation of the LHS of moves
  - \( r_1 = r_2 + r_3 \rightarrow r_4 = r_2 + r_3 \)
  - ...
  - \( r_5 = r_1 + r_6 \rightarrow r_5 = r_4 + r_6 \)
  - ...
  - \( r_4 = r_1 \rightarrow \text{noop} \)
- Rules (ops X and Y in same BB)
  - dest(X) is a register
  - dest(X) not live out of BB(X)
  - Y is a move
  - dest(Y) is a register
  - Y consumes dest(X)
  - dest(Y) not consumed in (X...Y)
  - dest(Y) not defined in (X...Y)
  - There are no uses of dest(X) after the first redefinition of dest(Y)

CSE – Common Subexpression Elimination

- Eliminate recomputation of an expression by reusing the previous result
  - \( r_1 = r_2 * r_3 \)
  - \( \rightarrow r_{100} = r_1 \)
  - ...
  - \( r_4 = r_2 * r_3 \rightarrow r_4 = r_{100} \)
- Benefits
  - Reduce work
  - Moves can get copy propagated
- Rules (ops X and Y)
  - X and Y have the same opcode
  - src(X) = src(Y), for all srcs
  - expr(X) is available at Y
  - if X is a load, then there is no store that may write to address(X) along any path between X and Y
  - Prop is a load, call it redundant load elimination rather than CSE

If \( r_5 = r_2 + r_3 \) and \( r_5 = r_2 + r_9 \), then call it redundant load elimination rather than CSE.
Class Problem

Optimize this applying
1. constant propagation
2. constant folding
3. strength reduction
4. dead code elimination
5. forward copy propagation
6. backward copy propagation
7. CSE

```
r1 = 9
r4 = 4
r5 = 0
r6 = 16
r2 = r3 * r4
r8 = r2 + r5
r9 = r3
r7 = load(r2)
r5 = r9 * r4
r3 = load(r2)
r10 = r3 / r6
store(r8, r7)
r11 = r2
r12 = load(r11)
store(r12, r3)
```

Constant Combining

- Combine 2 dependent ops into 1 by combining the literals
  - r1 = r2 + 4
  - ...
  - r5 = r1 - 9 → r5 = r2 - 5
- First op often becomes dead
- Rules (ops X and Y in same BB)
  - X is of the form rx + K
  - dest(X) != src1(X)
  - Y is of the form ry + K
    (comparison also ok)
  - Y consumes dest(X)
  - src1(X) not modified in (X...Y)

```
r1 = r2 + 4
r3 = r1 < 0
r2 = r3 + 6
r7 = r1 - 3
r8 = r7 + 5
```

Operation Folding

- Combine 2 dependent ops into 1 complex op
  - Classic example is MPYADD
  - r1 = r2 * r3
  - ...
  - r5 = r1 + r4 → r5 = r2 * r3 + r4
- First op often becomes dead
- Borders on machine dependent opti (often it is !!)
- Rules (ops X and Y in same BB)
  - X is an arithmetic operation
  - dest(X) != any src(X)
  - Y is an arithmetic operation
  - Y consumes dest(X)
  - X and Y can be merged
  - src(X) not modified in (X...Y)

```
r1 = r2 & 4
r3 = r1 ^ -1
r2 = r3 < 6
r4 = r2 <= 0
r5 = r6 <= 1
r7 = r5 + r8
```