EECS 583 – Class 8 Classic Optimization

University of Michigan

September 27, 2021

Announcements & Reading Material

- ✤ HW2 Get busy on it ASAP!
- Talk about course projects next class Start thinking about forming/joining a group (2-4 students)
- Today's class
 - *Compilers: Principles, Techniques, and Tools*,
 A. Aho, R. Sethi, and J. Ullman, Addison-Wesley, 1988,
 9.9, 10.2, 10.3, 10.7 Edition 1; 8.5, 8.7, 9.1, 9.4, 9.5 Edition 2
- Material for Wednesday
 - » "Compiler Code Transformations for Superscalar-Based High-Performance Systems," S. Mahlke, W. Chen, J. Gyllenhaal, W. Hwu, P. Chang, and T. Kiyohara, *Proceedings of Supercomputing '92*, Nov. 1992, pp. 808-817
 - » And if you want more on ILP optimizations: D. J. Kuck, The Structure of Computers and Computations. New York, NY: John Wiley and Sons, 1978. (optional!)

- Make the code run faster on the target processor
 - » My (Scott's) favorite topic !!
 - » Other objectives: Power, code size
- Classes of optimization
 - » <u>1. Classical</u> (machine independent, done at IR level)
 - Reducing operation count (redundancy elimination)
 - Simplifying operations
 - Generally good for any kind of machine
 - » 2. Machine specific (done in llc)
 - Peephole optimizations
 - Take advantage of specialized hardware features
 - » 3. Parallelism enhancing (IR level often, but sometimes llc)
 - Increasing parallelism (ILP or TLP)
 - Possibly increase instructions

A Tour Through the Classical Optimizations

- For this class Go over concepts of a small subset of the optimizations
 - » What it is, why its useful
 - » When can it be applied (set of conditions that must be satisfied)
 - » How it works
 - » Give you the flavor but don't want to beat you over the head
- Challenges
 - » Register pressure?
 - » Parallelism verses operation count

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
- Better Algorithm
 - » 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



Local Constant Propagation

•	Forward propagation of moves
	of the form

- rx = L (where L is a literal)
- » Maximally propagate
- 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)
 - A. There is no definition of dest(X) between X and Y (local available defs)

BB1 1. r1 = 52. $r2 = `_x'$ 3. r3 = 74. r4 = r4 + r15. r1 = r1 + r26. r1 = r1 + 17. r3 = 128. r8 = r1 - r29. r9 = r3 + r510. r3 = r2 + 111. r10 = r3 - r1

Note, ignore operation format issues, so all operations can have literals in either operand position

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)) (global available defs)
 - » 5. Dest(x) is not modified between the top of BB(Y) and Y (local available defs)



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
 - $r1 = 3 * 4 \rightarrow r1 = 12$
 - $r1 = 3 / 0 \rightarrow ???$ Don't evaluate excepting ops !, what about floating-point?
 - » Evaluate conditional branch, replace with BRU or noop
 - if (1 < 2) goto BB2 \rightarrow BRU BB2
 - if (1 > 2) goto BB2 \rightarrow convert to a noop
- Algebraic identities
 - » $r1 = r2 + 0, r2 0, r2 | 0, r2 \land 0, r2 << 0, r2 >> 0$
 - r1 = r2
 - » r1 = 0 * r2, 0 / r2, 0 & r2
 - r1 = 0

»
$$r1 = r2 * 1, r2 / 1$$

• r1 = r2

Class Problem



Forward Copy Propagation

- Forward propagation of the RHS of moves
 - » r1 = r2
 - » ...
 - » r4 = r1 + 1 → r4 = r2 + 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



CSE – Common Subexpression Elimination

- Eliminate recomputation of an expression by reusing the previous result
 - » r1 = r2 * r3

» ...

»
$$r4 = r2 * r3$$
 → $r4 = r100$

- Benefits
 - » Reduce work
 - » Moves can get copy propagated

= r1

- Rules (ops X and Y)
 - » X and Y have the same opcode
 - » $\operatorname{src}(X) = \operatorname{src}(Y)$, 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



if op is a load, call it redundant load elimination rather than CSE, Redundant store elimination also possible, but applied a bit differently – think about it!

Class Problem



Class Problem Solution



Loop Invariant Code Motion (LICM)

- Move operations whose source operands do not change within the loop to the loop preheader
 - » Execute them only 1x per invocation of the loop
- Rules
 - » X can be moved
 - » src(X) not modified in loop body
 - » X is the only op to modify dest(X)
 - » for all uses of dest(X), X is in the available defs set
 - » for all exit BB, if dest(X) is live on the exit edge, X is in the available defs set on the edge
 - » if X not executed on every iteration, then X must provably not cause exceptions
 - » if X is a load or store, then there are no writes to address(X) in loop



LICM Example



Global Variable Migration

- Assign a global variable temporarily to a register for the duration of the loop
 - » Load in preheader
 - » Store at exit points
- Rules
 - » X is a load or store
 - » address(X) not modified in the loop
 - » if X not executed on every iteration, then X must provably not cause an exception
 - All memory ops in loop whose address can equal address(X) must always have the same address as X



Global Variable Migration Example



Induction Variable Strength Reduction

- Create basic induction variables from derived induction variables
- Induction variable
 - » BIV (i++)
 - 0,1,2,3,4,...
 - » DIV (j = i * 4)
 - 0, 4, 8, 12, 16, ...
 - DIV can be converted into a BIV that is incremented by 4
- Issues
 - » Initial and increment vals
 - » Where to place increments



Induction Variable Strength Reduction (2)

*	Rules		BB1		
	»	X is a *, <<, + or – operation			
	»	src1(X) is a basic ind var			
	»	src2(X) is invariant			
	»	No other ops modify dest(X)			
	»	dest(X) = src(X) for all srcs			
	»	dest(X) is a register	BB2 1. r5 = r4	= r4 - 3	
*	Tra	nsformation	2. $r4 = r4 + 1$		
	»	Insert the following into the preheader	BB3		BB4
		• $new_reg = RHS(X)$			
	»	If opcode(X) is not add/sub, insert to the		3. r7 =	r4 * r9
		bottom of the preheader			
		• new_inc = inc(src1(X)) opcode(X) src2(X)			
	>>	else	PP5 A re	-r/2	
		• $new_inc = inc(src1(X))$		- 14 << 2	
	*	Insert the following at each update of src1(X)			
		• new_reg += new_inc	BB6		

» Change $X \rightarrow dest(X) = new_reg$

Class Problem



