Lecture 12: Introduction to Optimization

I. HANDOUT handout4.docx (optimization example)

II. Optimization

A. What is optimization
   1. Modifying the generated code to
      a. Make it more efficient (time or space)
      b. Without changing its semantics or behavior
   2. What are the basic principles
      a. The compiler should do as good or better than hand-coded assembler
      b. Take an example: handout4
         i. See what the compiler generates
         ii. How could you improve on this
         iii. What code would you write?
   3. Optimization is an art, not a science
      a. Optimization often involves guessing
         i. It can (and does) result in pessimization
      b. There are many things that can be done
         i. However, each effects the others
         ii. What order things get done in is important
         iii. No order is best
         iv. Some things might have to be repeated
   4. Optimization is a field unto itself
      a. Data structures are important
      b. Complex algorithms have evolved (and become useful given the large size of the problems)
      c. Problems that are inherently n**2 or worse need to be done in linear time
      d. Problems that are NP-complete need to be done in linear time

B. Optimization structure
   1. Where it fits into the compiler (diagram)
a. Front end -> Icode Gen -> ICODE -> Object Gen
b. Optimization consists of a number of modules linked to ICODE
   i. Add nodes for each of the following as we go

2. Local Optimization
a. Optimizations done solely within a block
   i. Constant propagation
   ii. Value numbering
   iii. Algebraic identities
   iv. Dead code elimination
b. Generally can also be done over multiple blocks
   i. But this requires more information

3. Dominator optimization
a. Basic concepts
   i. Optimizations that do not move computations around
   ii. Done within each basic block
   iii. But use the notion of domination to combine information from different blocks
b. What gets done (use examples from the handout)
   i. Eliminate redundant computations
   ii. Copy propagation (avoiding copy operations)
   iii. Constant propagation
   iv. Constant folding
   v. Algebraic identities, peephole optimizations
   vi. Dead code elimination
   c. These can be done first within a basic block (local optimizations)
      i. As code is generated
      ii. As a post-pass over the code
      iii. But this duplicates efforts later on
b. SSA form
   i. This is easier to do if you track where each value is computed
   ii. SSA form ensures there is a single assignment to each temporary
e. What we’ll cover
   i. Computing and using dominators
   ii. Conversion into and out of SSA form
   iii. A subset of possible dominator optimizations

4. Interprocedural optimization
a. Basic concepts
i. Allow optimization over and within procedure calls
ii. Determine what variables are or might be modified by a procedure
iii. Determine what variables are or might be used by a procedure
iv. Handle aliasing

b. If you don’t do this
i. Every call invalidates all global variables
ii. Accessible variables
iii. Aliased access to other variables
iv. Arrays
c. Difficulties
i. Might not have all the procedures at once
ii. Pointers (aliasing)
iii. Pointers to functions, virtual methods, etc.
d. We’ll look at ways of doing this
i. Context sensitivity

5. Dependence Optimization
a. Basic concepts
i. Reduce memory references
ii. Allow memory references to be overlapped
iv. Loop unrolling and jamming
b. We’ll cover this a bit

6. Global Optimization
a. Basic concepts
i. Clean up the icode
ii. Move code around for better performance
iii. Optimizations that involve multiple blocks
b. What gets done
i. Reshaping: find and move loop-invariant computations
ii. Strength reduction: convert multiplies to adds
iii. Redundant expression elimination (again)
iv. Moving loads and stores

7. Limiting Resources
a. Basic concepts
i. Peephole optimization
  ➢ Finding more efficient instructions
Combining instructions
  ii Removing copies
    ➢ Don’t insist that temporaries hold a consistent value
  iii Removing empty basic blocks
b. How it is done
  i Lowering: make icode instructions correspond to machine code
  ii Renaming temporaries throughout
  iii Coallescing temporaries that are copies
  iv Reducing register pressure by moving instructions around and by
    storing and reloading values
c. What we’ll look at
  i Renaming, coalescing temporaries
  ii Some peephole optimization

8. Register allocation
a. Basic concepts
  i Assign each temporary to a register
  ii Make sure register usage doesn’t overlap
  iii NP-complete
b. Global register allocation
  i Assign temporaries to registers over the whole routine
  ii Matter of priorities, checking overlap, etc.
  iii Generally optional
c. Local register allocation
  i Assign temporaries to registers in a basic block
  ii This has to be done (machine code requires it)

9. Instruction Scheduling
a. Basic concepts
  i Today’s machines are pipelined
  ii Want to arrange code to keep the pipeline fill
  iii Memory access patterns and the cache
b. How it is done
  i Simulate a model of the processor going through each block
  ii Move instructions around as appropriate
  iii NP-complete

10. Parallelization
  a. Determine how to run a loop in multiple threads
    i Determine dependencies in the computation
b. Escape analysis: what is local and what is global
   i. Generating memory barriers if needed

c. Optimizing for cache hits

11. Shape analysis
   a. Want to do alias analysis for pointer-based data structures
      i. Build a model of what each pointer can contain
         ➢ Finite approximation to infinite model
   b. Use this model to determine possible aliasing, etc.

12. Data flow analysis
   a. Tracking the flow of values through the program
      i. Where can this NEW’d object appear
      ii. Where can this parameter be used
      iii. Where can this sensitive value go to?