Optimization

Code Optimization

☐ Goals
  ☐ Improve program speed
  ☐ Shrink program size (e.g. better cache hits)

☐ Criteria
  ☐ Safety ... did it change the program’s results
  ☐ Profitable ... actually works

☐ Kinds
  ☐ local (inside basic blocks)
  ☐ global (across basic blocks)
Basic Blocks

- code with no flow in or out.
  - pint 20
  - stor 2,35
  - L1: pint 10
    - load 2,35
    - less
    - fjmp L2:
    - load 2,35
    - stor 1,16
    - load 2,35
    - pint 1
    - sub
    - jmp L1
    - L2:
- three basic blocks
Safe vs Unsafe

- Safe -- results are the same
- Unsafe -- results may be different
  - Associative reordering of operands
  - Movement of expressions and code sequences
  - Loop unrolling

- Sometimes users may want "unsafe" optimizations

- Profitability --
  - May improve "average case"
  - May not improve "edge" cases
    - Loop invariant movement ...
  - Optimizer bugs ...

- Debugging and optimization
Categories of Optimizations

- Source Language
  - semantic routines
  - language-specific
  - target machine independent

- Code-Generation
  - source language-independent
  - exploit target machine

- Intermediate representation optimization
  - language independent
  - target machine independent
Source Language Optimizations

- constant bounds in loops and arrays
- inhibiting code generation for unreachable code
  - identifying "constant variables"
- Unrolling loops
  
  for i < 1 to 10 do A[i] <-- 2*i ; end for ;
  
  a[1] <-- 2; a[2] <-- 4; ....

- runtime checks ... Only if needed ...

- Design can also impact code quality
  - named constants
  - += type operators
  - case vs if
  - for loop / protected loop variables
  - restricted jumps and gotos (easier flow analysis)
Source Language Optimizations (page 2)

Poor language features
- name parameters
- functions that have side effects
- alias creation
- exceptions (unexpected & invisible jumps)
  - resume normal flow?
Code-generation Optimization

- careful allocation and use of registers, avoid spills
- thorough use of instruction sets and addressing modes
  - VAX: `sobgtr Rn, Label`
  - `load A, r1; mul 3, r1; stor r1, A; => mul 3, A`
- exploitation of special hardware
  - vector pipelines, ....

Peep hole optimization

- constant folding
- Strength reduction: `*2` replaced by `<<1`
- Null sequences: `a <-- 3; a <-- b*c;`
- Combine several operations into one (`sobgtr`)
- Algebraic laws: `0-x => -x; +0, *1`
- Special instructions: `b <- b-1; decr b`
Other code generation optimizations

- \( a[3] \leftarrow 10; \ (A[1:10]) \)
- \( \text{pint} \ 10; \ \text{pint} \ 3; \ \text{pint} \ 1; \ \text{sub} \ 1,A \)
- \( \text{pint} \ 10; \ \text{pint} \ 3; \ \text{stx}, l,A-1 \)

Common subexpression elimination

- \( b \leftarrow a+5; \ c \leftarrow (a+5) \times 6; \ d \leftarrow d+1; \)
- \( l[a] < l[b] \land l[c] < l[b] \)

Problems:
- aliasing
  - pointers
  - reference parameters
Intermediate Representation Optimizations

Local and Global

- Local -- "peephole optimization" (earlier)
- Global -- Program is a graph of basic blocks
- Some Optimizations may require both

- e.g. Common subexpression identification
  
  \[
  A \leftarrow B + C; \\
  D \leftarrow B + C; \\
  \text{if } A > 0 \text{ then } E \leftarrow B + C; \text{ end if;}
  \]

  2 CSE in local, one more in global

- Local ... easy, global ... harder
- Global may yield good ones in array processing
Loops turn out to be a great source of optimizations

- Expression invariant
  □ moved to loop entry and evaluated once
  □ may be "unsafe" or unproductive

\[ \text{while } J > I \text{ do } a[J] \leftarrow 10/I; J \leftarrow J+2; \text{ end while}; \]

- Induction variables
  □ for \( j \leftarrow a \to b \ldots \) 2\( j \) or \( x^j \)
  □ strength reduction, + instead of *
More Loop Optimization

□ Loop invariants --

    for i <-- 1 to 100 do
    for j <-- 1 to 100 do
    for k <-- 1 to 100 do
        a[i,j,k] <-- i*j*k;
        end for;
    end for;
    end for;
end for;

□ Any invariants?
    □ i*j in inner loop
    □ address of a[i,j] in inner loop
    □ address of a[i], move out two levels
    □ C programmers can do this code movement
Optimization in perspective

- Many programs may not need optimization
- Good optimization improvement?
  - 10%? 25%? 50%?
- Change in algorithm?
  - O(n^2) to O(n log n)
- Use of profiling
  - User finding where to put effort
  - 90% in 10% rule
- Optimization rarely produces optimal code, why?
  - undecidable problems!
    - reachability, dead code elimination
      - simple cases are done
  - too expensive
    - generating optimal code from dags is exponential in the number of shared sub dags
Subprogram calls

- Block structured / OO languages => procedure calls!
- Building ARs ... expensive! (e.g static chains ...)
- Typical implementation ... "closed"
- Consider "inline" implementations
  - Consider parameters
    - macro expansion not always good
  - May open optimization of parameter use
    - folding, deletion of unreachable code
    - constant "variables" ...

Issues
- When to use
- Who decides (user or compiler)
  - C++ inline, body in class definition
- How to do it correctly
  - e.g. value parameters with assignment
Inline calls

- Call graphs
  - recursive routines identified
  - call frequency
  - one arc -> do it inline

- Add to graphs size of functions
  - small ones should be inlined
    - e.g. accessor and mutator functions in C++ classes

- Profiler may help identify candidates

- Frequently called ones

- ones called from loops
Implementation

- inline should do same as closed
- simple mutator, accessor \(\rightarrow\) very easy
  - almost like macros
- more complex \(\ldots\) \(a[j]\) as a value
  - \(j\) may change
  - \(a[j]\) should be same value
  - use of temporaries
- reference parameters, address in temporaries
- (Note, use of temps may cost a lot less than calls)

- local variables?
  - add to callers frame
Optimization of closed routines

- subroutine call -- one instruction
- block structured
  - parameters
  - registers to save and restore
  - displays to be updated
  - etc ...

- Non-recursive subroutines could be allocated a static AR!
- Call graph also can help
  - routines not in same call sequence can share
  - topological sorts can help
- (Can imply no code generation before full parsing.)
- AR’s in "static" storage
- Dynamic elements on stack (e.g. dynamic arrays)
- Problems? F(a,b,c,G(x),e,z);
Use of registers as temps
Proc uses a register, do we save the old value?
Use call graph to mark number of regs needed per procedure
Save only needed registers or allocate so no saves needed
Have callee save registers is another option.
Local variables in registers
Parameters in registers
Return values in registers (common)
Other Optimization techniques

Interprocedural dataflow analysis
- estimate the effects of a call
- predict which variables need to be "killed"
- 2 sets to maintain
  - DEF(P) -- variables defined by P
  - USE(P) -- variables used by P
  - localDEF(p), localUSE(p)
- May also help generate warnings
- adding in ones for called procedures
- iterative computation
- Formal vs Actual parameters
Global Data Flow Analysis

Common optimizations resulting from global data flow analysis

- Very Busy Expressions (loop invariants, used on all paths)
- Global common subexpression elimination
- Live variable analysis
  - Live variable -- any variable who’s current value will be used before the value is overwritten by a store.
  - Dead variables at end of block don’t need to be stored!
- Graph based algorithms for computing live/dead variables
- Uninitialized variable analysis
  - Use of a dead variable may imply use of uninitialized variable.

No details of algorithms ... see the book