Compiler Optimization

- The compiler translates programs written in a high-level language to assembly language code
- Assembly language code is translated to object code by an assembler
- Object code modules are linked together and relocated by a linker, producing the executable program
- Code optimization can be done in all three stages
  - what kind of optimization is done depends on the compiler/assembler/linker that is used
  - there are significant differences in optimization capabilities between different compilers
  - important to write clear and simple code that the compiler can analyze

1

The compilation process

- Preprocessing
  - simple textual manipulations of the source code
  - incude files, macro expansions, conditional compilation
- Lexical analysis
  - source code statements are decomposed into tokens (variables, constants, language constucts, ...)
- Parsing
  - syntax checking
  - source code is translated into an intermediate language form
- Optimization
  - one or more optimization phases performed on the intermediate language form of the program
- Code generation
  - intermediate language form is translated to assembly language
  - assembler code is optimized

2

Intermediate language

- Expresses the same computation as the high-level source code
  - represented in a form that is better suited for analysis
  - also includes computations that are not visible in the source code, like address calculations for array expressions
- A high-level language statement is represented by several IL statements
  - IL is closer in complexity to assembly language than to a high-level language

C

```
while (j<n) {
    k = k+j*2;
    m = j*2;
    j++
}
```

IL

```
A: t1 = j
   t2 = n
   t3 = (t1<t2)
   jmp (B) t3
   jmp (C) TRUE
B: t4 = k
   t5 = j
   t6 = t5*2
   t7 = t4+t6
   k = t7
   t8 = j
   t9 = t8*2
   m = t9
   t10 = j
   t11 = t10+1
   j = t11
   jmp (A) TRUE
C:
```

3

Basic blocks

- Basic blocks are regions of code with one entry at the top and one exit at the bottom
  - no branches within a basic block
  - generated from the syntax tree which is built by the parser
- A flow graph describes the transfer of control between basic blocks
- Data dependency analysis
  - builds a directed acyclic graph (DAG) of data dependences
- The compiler both optimizes the code within basic blocks and across multiple basic blocks

```
A: t1 = j
   t2 = n
   t3 = (t1<t2)
   jmp (B) t3

   jmp (C) TRUE

B: t4 = k
   t5 = j
   t6 = t5*2
   t7 = t4+t6
   k = t7
   t8 = j
   t9 = t8*2
   m = t9
   t10 = j
   t11 = t10+1
   j = t11
   jmp (A) TRUE
C:
```

4

Compiler optimization techiques

- Most compiler optimization techniques optimize code speed
  - sometimes on the expense of code size
  - the user can choose what kind of optimizations to apply by compiler options (-O1, -O2, -O3, -Os)
- The basic optimization techniques are typically very simple
  - operate on code within a basic block
  - reduce the number of instructions and memory references
- Loop optimizations operate across basic blocks
  - can move code from one basic block to another
- Peephole optimizations
  - replaces short instruction sequences (1-4 instructions) with more efficient alternatives

Register allocation

- Register allocation decides which values are stored in registers
  - starts on the basic block level
  - global register allocation optimizes use of registers across multiple blocks
- In general, all variables can not be stored in registers
  - *register spilling* – values and memory references have to be stored in memory locations (on the stack) instead of in registers
  - may slow down the code becuse of frequent memory accesses
  - register allocation is not critical in processors with register renaming
- Register storage class in C
  - advises the compiler that a variable will be heavily used
  - the compiler is free to ignore the advice

Simple register allocation method

- Analyze how temporary variables t1, t2, t3, ... are used in a basic block
  - a variable is *dead* when the next reference to it is an assignment or when there are no further references to it
  - a variable is *live* if it will be read in subsequent instructions (used on the right hand side in an expression)
- Simple register allocation method
  - when a variable is seen for the first time it is allocated to a free register or a register containing a dead variable
  - if no such register exists, select the register whos use is furthest ahead, spill that register and allocate it to the new variable
- More advanced register allocation method
  - graph colouring

7

Register allocation via graph colouring

- Build an interference graph of the variables in a basic block
  - nodes represent variables (t1, t2, ...)
  - arc between two nodes if they are both live at the same time
- Two nodes that are alive at the same time can not be allocated to the same register
- The problem is to find a colouring of the interference graph using *N* colours
  - assign each node (variable) a colour (register) so that any two connected nodes have different colours
- Optimal graph colouring is NP-complete
  - have to use heuristic algorithms
  - can not guarantee that we find an *optimal* soulution

8

Compiler optimization techniques

- Different classes of compiler optimization techniques
- Optimizations that improve assembly language code
  - reduces the number of instructions and memory references
  - uses more efficient instructions or assembly language constructs
  - instruction sceduling to improve pipeline utilization
- Optimizations that improve memory access
  - reduces cache misses
  - prefetching of data
- Loop optimizations
  - builds larger basic blocks
  - removes branch instructions
- Function call optimization

Constant folding

- Expressions consisting of multiple constants are reduced to one constant value at compile time
- Example:
  - two constants `Pi` and `d`
  - `tmp = Pi/d` is evaluated at compile time
  - the compiler uses the value `tmp` in all subsequent expressions containing `Pi/d`

```
const double Pi = 3.15149;
  ...
d = 180.0;
  ...
t = Pi*v/d;
```

```
  ...
t = v*tmp;
```

- Explicitely declaring constant values as constants helps the compiler to analyze the code
  - also improves code readability and structure

Copy propagation

- Assignment to a variable creates multiple copies of the same value
  - introduces dependencies between statements
  - the assignment must be done before the expression in which the copy is used can be evaluated
- Example:
  - the second statement depends on the first
  - copy propagation eliminates the dependency
  - if `x` is not used in the subsequent computation, the assignment `x = y` can be removed (by dead code elimination)

```
x = y;
z = c+x;
```

```
x = y;
z = c+y;
```

- Reduces register pressure and eliminates redundant register-to-register move instructions

11

Dead code removal

- Remove code that has no effect on the computation
  - often produced as a result of other compiler optimizations
  - may also be introduced by the programmer
- Two types of dead code
  - instructions that are unreachable
  - instructions that produce results that are never used

```
#define DEBUG 0
 ...
if (DEBUG) {
  /* debugging code */
  ...
}
 ...
```

- Can completely change the behaviour of simple synthetic benchmark programs
- Reduces code size, improves instruction cache usage

12

Strenght reduction

- Replace slow operations by equivalent faster ones
  - replace muliplication by a constant *c* with *c* additions
  - replace power function by multiplications
  - replace division by a constant *c* with multiplication by *1/c*
  - replace integer multiplication by 2^n with a shift operation
  - replace integer division by 2^n with a shift operation, for positive values
  - replace integer modulo-2 division by masking out the least significant bit
- Some transformations may affect the precision of floating-point calculations

Expression	Replaced by
`x*2`	`x+x`
x^2	`x*x`
$x^{2.5}$	$x^2 * \sqrt{x}$
`x/n`	`x*(1/n)`
$k*2^n$	`k<<n`
$k/2^n$	`k>>n (k>0)`
`k%2`	`k&1`

Induction variable optimization

- Simplify expressions that change as a linear function of the loop index
  - the loop index is multiplied with a constant
  - replaces a multiplication with a number of additions
- Used in array address calculations for iteration over an array

```
for (i=0; i<N; i++) {
    k = 4*i+m;
    ...
}
```

```
k=m;
for (i=0; i<N; i++) {
    k=k+4;
    ...
}
```

```
    adr = base_address(A) - sizeof_datatype(A)
L1:
    ...
    adr = adr + sizeof_datatype(A)
    ...
    jcc L1
```

Common subexpression elimination

- Replace subexpressions that are evaluated more than once with a temporary variable
  - evaluate the subexpression and store it in a temporary variable
  - use the temporary variable instead of the subexpression
  - the subexpression is computed once and used many times
- Associative order may be important
  - is it correct to replace *(a+b+c)* by *(c+b+a)*
- Used to simplify address calculations in array indexing or pointer de-referencing

```
d = c*(a+b);
e = (a+b)/2;
```

```
tmp=a+b;
d = c*(tmp);
e = (tmp)/2;
```


Loop invariant code motion

- Move calculations that do not change between loop iterations (loop invariant code) out of the loop
  - often used to eliminate load- and store operations from loops
- Hoisting
  - move invariant code before the loop
  - *example*:
    load value of *y* into a register before the loop
- Sinking
  - move invariant code after the loop
  - *example*:
    load value of *s* into a register before the loop
    store value of register into *s* after the loop

```
for (i=0; i<N; i++) {
   X[i] = X[i]*y;
}
```

```
for (i=0; i<N; i++) {
   s = s+X[i];
}
```

Loop unswitching

- Move loop-invariant conditional constructs out of the loop
  - if- or switch-statements which are independent of the loop index are moved outside the loop
  - the loop is instead repeated in the different branches of the if-or case- statement
  - removes branches from within the loop
- Removes branch instructions, increases instruction level parallelism

```
for (i=0; i<N; i++)
{  if (a>0)
      X[i] = a;
   else
      X[i] = 0;
}
```

```
if (a>0)
{ for (i=0; i<N; i++)
     X[i] = a;
}
else
{ for (i=0; i<N; i++)
     X[i] = 0;
}
```


Loop unrolling

- Replicate the body of a loop *k* times and increase the loop counter with *k*
  - *k* is called the unrolling factor
- Reduces loop overhead
- Removes branch instructions
- Produces larger basic blocks
  - increases instruction level parallelism
  - more opportunities for instruction scheduling
- Increases code size

```
/* Copy Y to X */
for (i=0; i<N; i++) {
  X[i] = Y[i];
}
```

```
/* Copy Y to X */
limit = (N/5)*5;
for (i=0; i<limit; i+=5) {
  X[i] = Y[i];
  X[i+1] = Y[i+1];
  X[i+2] = Y[i+2];
  X[i+3] = Y[i+3];
  X[i+4] = Y[i+4];
}
/* Last N%5 elements */
for (i=limit; i<N; i++) {
  X[i] = Y[i];
}
```

Procedure inlining

- Also called in-line expansion
- Replace a function call by the body of the function
  - eliminates the overhead of the function call
  - improves possibilities for compiler analysis and optimization
- Increases code size
  - upper limit on the size and complexity of functions that can be inlined

```
double max(double a, double b) {
  return ((a>b) ? a : b);
}
...
for (i=0; i<N; i++) {
  Z[i] = max(X[i], Y[i]);
}
```

```
...
for (i=0; i<N; i++) {
  Z[i] = (X[i]>Y[i]) ? X[i] : Y[i];
}
```


Compiler optimization in gcc

- Lexical analysis and parsing
  - reads in the source program as a strem of characters
  - statements are read as a syntax tree
  - data type analysis, data types attatched to tree nodes
  - constant folding, arithmetic simplifications
- Intermediate language generation (RTL)
  - syntax tree representation is converted to RTL
  - optimizations for conditional expressions and boolean operators
  - tail recursion detection
  - decisions about loop arrangements
- At the end of RTL generation, decisions about function inlining is done
  - based on the size of the function, type and number of parameters

Compiler optimization in gcc (cont.)

- Branch optimization
  - simplifies branches to the next instruction and branches to other branch instructions
  - removes unreferenced labels
  - removes unreachable code
    - unreachable code that contains branches is not detected in this stage, they are removed in the basic block analysis
- Register scan
  - finds first and last use of each pseudo-register
- Jump threading analysis
  - detects conditional branches to identical or inverse tests and simplifies these (only if *-fthread-jumps* option is given)
- Common subexpression elimination
  - constant propagation
  - reruns branch optimization if needed

21

Compiler optimization in gcc (cont.)

- Loop optimization
  - loop invariant code motion and strength reduction
  - loop unrolling
  - if *-rerun-cse-after-loop* option is given, the common subexpression elimination phase is performed again
- Stupid register allocation (if compiling without optimization)
  - simple register allocation, includes some data flow analysis
- Data flow analysis
  - divides the program into basic blocks
  - removes unreachable loops and computations whos results are never used
  - live range analysis of pseudo-registers
  - builds a data flow graph where the first instruction that uses a value points at the instruction that computes the value
  - combines memory references that adds or subtracts to/from a value to produce autoincrement/autodecrement addressing

22

Compiler optimization in gcc (cont.)

- Instruction combination
  - combines groups of 2-3 instructions that are related by data flow into a single instruction
  - combines RTL expressions, algebraic simplifications
  - selects expressive instructions from the instruction set
- Instruction scheduling
  - uses information about instruction latency and throughput to reduce stalls
  - especially memory loads and floating-point calculations
  - re-orders instructions within a basic block to reduce pipeline stalls
- Register class preferencing
  - analyses which register class is best suited for each pseudo register

23

Compiler optimization in gcc (cont.)

- Local register allocation
  - allocates registers defined in the ISA to pseudo registers
  - only within basic blocks
- Global register allocation
  - allocates remaining registers to pseudo registers (pseudo registers with a life span covering more than one basic block)
- Reloading
  - renumbers pseudo registers with hardware register numbers
  - allocates stack slots to pseudo registers that did not get hard registers
  - finds instructions that have become invalid because the value it operates on is not in a register, or is in a register of wrong type
  - reloads these values temporarily into registers, inserts instructions to copy values between memory and registers

24

Compiler optimization in gcc (cont.)

- Realoading reruns the instruction sceduling phase
  - also frame pointer elimination (if *-fomit-frame-pointer* option is given)
  - inserts instructions around subroutine calls to save and restore clobbered registers
- Instruction scheduling is rerun
  - tries to avoid pipeline stalls for memory loads generated for spilled registers
- Jump optimization is rerun
  - removes cross jumping
  - removes no-op move instructions
- Delayed branch scheduling
  - inserts instructions into branch slots (on architectures with delayed branches)

25

Compiler optimization in gcc (cont.)

- Conversion from hard registers to register stack
  - floating-point registers on x87 FPU
- Final code generation
  - outputs assembler code
  - performs machine-specific peephole optimization
  - generates function entry and exit code sequences
- Debugging information output
  - outputs information for use by the debugger (if debugging switch is on)

26

Examining assembly code in gcc

- To examine the assembly language code that the compiler produces, compile with

```
gcc -c -g -O2 -Wa,-alhd,-L program.c
```

Compiler directives
- **-c** compile, but do not link
- **-g** produce debugging information
- **-O2** optimization level
- **-Wa** pass options to assembler

Assembler directives
- **-alhd** produce listing with assembly language, high-level language but no debugging information
- **-L** retain local labels

- Can also use *objdump* to examine object code

```
gcc -g program.c -o program
objdump -d -S -l program
```