## **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

## The compilation process

1

2

## Preprocessing

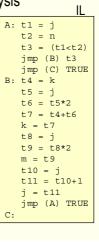
- simple textual manipulations of the source code
- incude files, macro expansions, conditional compilation
- Lexical analysis
  - source code statements are decomposed into tokens (variables, constants, language constucts, ...)
- Parsing
  - syntax checking
  - source code is translated into an intermediate 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

## 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



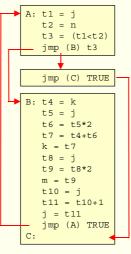


3

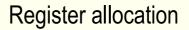
4

## 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



# <section-header><list-item><list-item><list-item><list-item><list-item><list-item><list-item><list-item><list-item><list-item><list-item><list-item><list-item><list-item><list-item><list-item>



- 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

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

## 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:

Pi/d

- two constants Pi and d
- tmp = Pi/d is evaluated at compile time
- the compiler uses the value tmp in all subsequent expressions containing v\*tmp;

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

- Explicitly 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)

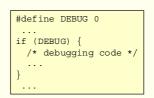
x z	= =	у; с+х;
x	=	у;
Z	=	с+у;

Reduces register pressure and eliminates redundant registerto-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



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



## 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<sup>n</sup> with a shift operation
- replace integer division by 2<sup>n</sup> with a shift operation, for positive values

Expression	Replaced by
x*2	x+x
x <sup>2</sup>	x*x
x <sup>2.5</sup>	x²*√x
x/n	x*(1/n)
k*2 <sup>n</sup>	k< <n< th=""></n<>
k/2 <sup>n</sup>	k>>n (k>0)
k%2	k&1

- replace integer modulo-2 division by masking out the least significant bit
- Some transformations may affect the precision of floatingpoint calculations

13

# Induction variable optimization

Simplify expressions that change as a linear function of the loop index

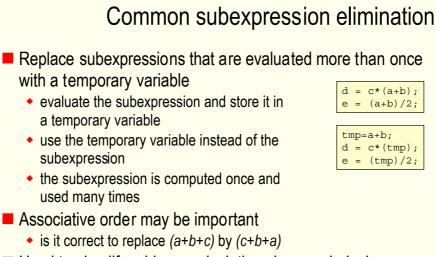
 for (i=0; i<N; i++) {</td>

- the loop index is multiplied with a constant
- replaces a multiplication with a number of additions

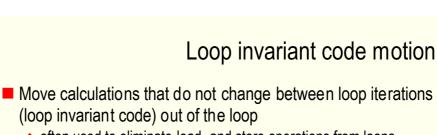
<pre>for (i=0; i<n; i++)="" k="4*i+m;" pre="" }<=""></n;></pre>	{
<pre>} k=m; for (i=0; i<n; i++)<="" pre=""></n;></pre>	{
k=k+4; }	ſ

Used in array address calculations for iteration over an array

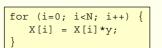
```
adr = base_address(A) - sizeof_datatype(A)
L1:
    ...
    adr = adr + sizeof_datatype(A)
    ...
    jcc L1
```



Used to simplify address calculations in array indexing or pointer de-referencing



- 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

for	(i=0;	i <n;< th=""><th>i++)</th><th>{</th></n;<>	i++)	{	
s = s + X[i];					
}					

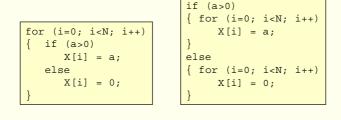
 example: load value of s into a register before the loop store value of register into s after the loop



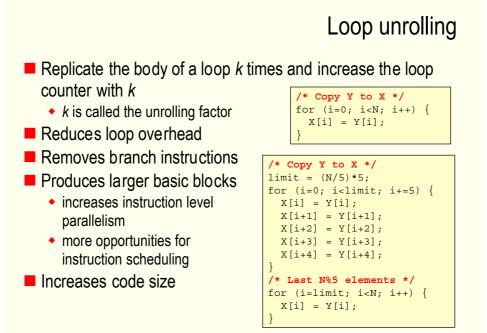
## Loop unswitching

### 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



17



## **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 double max(double a, double b) { upper limit on the size and return ((a>b) ? a : b); } complexity of functions that . . . can be inlined 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];19

# 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

## 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

# 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)

