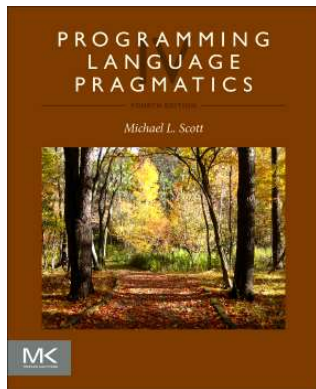


Code Improvement

17-363/17-663: Programming Language Pragmatics



Reading: PLP chapter 17



Prof. Jonathan Aldrich



Introduction

- We discussed target code generation
 - Typically produces correct but highly suboptimal code
 - redundant computations
 - inefficient use of the registers, multiple functional units, and cache
- This chapter takes a look at *code improvement*: the phases of compilation devoted to generating *good* code
 - we interpret “good” to mean *fast*
 - occasionally we also consider program transformations to decrease memory requirements
 - we sometimes say “optimization,” but the code produced is rarely truly optimal

Introduction

- In a very simple compiler, we can use a *peephole optimizer* to peruse already-generated target code for obviously suboptimal sequences of adjacent instructions
- At a slightly higher level, we can generate near-optimal code for *basic blocks*
 - a basic block is a maximal-length sequence of instructions that will always execute in its entirety (assuming it executes at all)
 - in the absence of hardware exceptions, control never enters a basic block except at the beginning, and never exits except at the end

Introduction

- Code improvement at the level of basic blocks is known as *local* optimization
 - elimination of redundant operations (unnecessary loads, common sub-expression calculations)
 - effective instruction scheduling and register allocation
- At higher levels of aggressiveness, compilers employ techniques that analyze entire subroutines for further speed improvements
- These techniques are known as *global* optimization
 - multi-basic-block versions of redundancy elimination
 - instruction scheduling, and register allocation
 - code modifications designed to improve the performance of loops

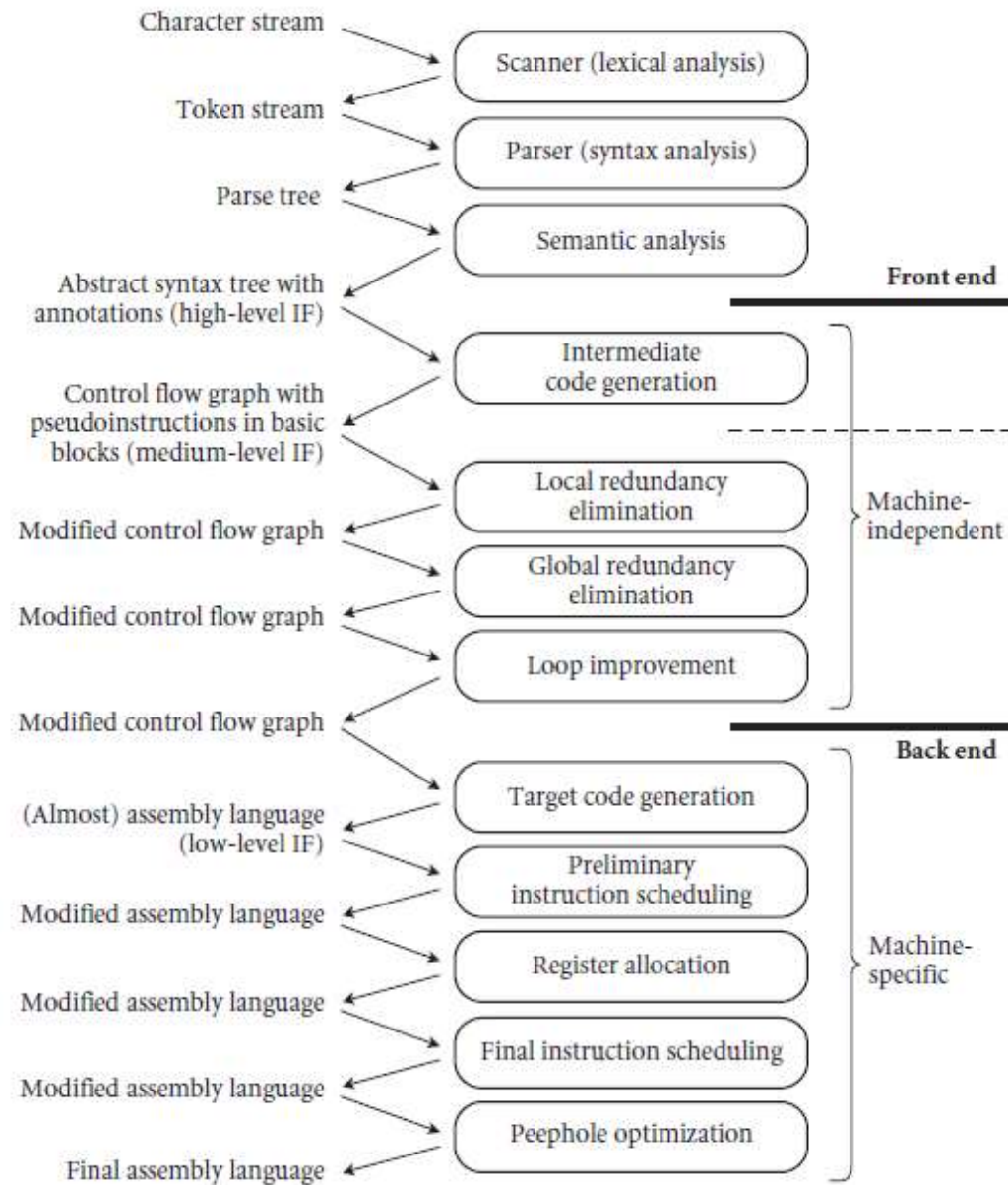
Introduction

- Both global redundancy elimination and loop improvement typically employ a *control flow graph* representation of the program
 - Use a family of algorithms known as *data flow analysis* (flow of information between basic blocks)
- Recent compilers perform various forms of *interprocedural* code improvement
- Interprocedural improvement is difficult
 - subroutines may be called from many different places
 - hard to identify available registers, common subexpressions, etc.
 - subroutines are separately compiled

Phases of Code Improvement

- We will concentrate in our discussion on the forms of code improvement that tend to achieve the largest increases in execution speed, and are most widely used
 - Compiler phases to implement these improvements is shown in Figure 17.1

Phases of Code Improvement



Phases of Code Improvement

- The *machine-independent* part of the back end begins with intermediate code generation
 - identifies fragments of the syntax tree that correspond to basic blocks
 - creates a control flow graph in which each node contains a sequence of three-address instructions for an idealized machine (unlimited supply of *virtual registers*)
- The *machine-specific* part of the back end begins with target code generation
 - strings the basic blocks together into a linear program
 - translates each block into the instruction set of the target machine and generating branch instructions that correspond to the arcs of the control flow graph

Phases of Code Improvement

- Machine-independent code improvement has three separate phases
 1. Local redundancy elimination: identifies and eliminates redundant loads, stores, and computations within each basic block
 2. Global redundancy elimination: identifies similar redundancies across the boundaries between basic blocks (but within the bounds of a single subroutine)
 3. Loop improvement: effects several improvements specific to loops
 - these are particularly important, since most programs spend most of their time in loops.
 - Global redundancy elimination and loop improvement may actually be subdivided into several separate phases

Phases of Code Improvement

- Machine-specific code improvement has four separate phases
 - Preliminary and final instruction scheduling are essentially identical (Phases 1 & 3)
 - Register allocation (Phase 2) and instruction scheduling tend to interfere with one another
 - the instruction schedules minimize pipeline stalls which tend to increase the demand for architectural registers (*register pressure*)
 - we schedule instructions first, then allocate architectural registers, then schedule instructions again
 - If it turns out that there aren't enough architectural registers, the register allocator will generate additional load and store instructions to *spill* registers temporarily to memory
 - the second round of instruction scheduling attempts to fill any delays induced by the extra loads

Peephole Optimization

- A relatively simple way to significantly improve the quality of naive code is to run a *peephole optimizer* over the target code
 - works by sliding a several instruction window (a peephole) over the target code, looking for suboptimal patterns of instructions
 - the patterns to look for are heuristic
 - patterns to match common suboptimal idioms produced by a particular front end
 - patterns to exploit special instructions available on a given machine
- A few examples are presented in what follows

Peephole Optimization

- *Elimination of redundant loads and stores*
 - The peephole optimizer can often recognize that the value produced by a load instruction is already available in a register

```
r2 := r1 + 5
```

```
i := r2
```

```
r3 := i
```

```
r3 := r3 × 3
```

becomes

```
r2 := r1 + 5
```

```
i := r2
```

```
r3 := r2 × 3
```

Peephole Optimization

- *Constant folding*
- A naive code generator may produce code that performs calculations at run time that could actually be performed at compile time
 - A peephole optimizer can often recognize such code

$r2 := 3 \times 2$

becomes

$r2 := 6$

Peephole Optimization

- *Constant propagation*

- Sometimes we can tell that a variable will have a constant value at a particular point in a program
- We can then replace occurrences of the variable with occurrences of the constant

r2 := 4

r3 := r1 + r2

r2 := . . .

becomes

r2 := 4

r3 := r1 + 4

r2 := . . .

and then

r3 := r1 + 4

r2 := . . .

Peephole Optimization

- *Common subexpression elimination*
 - When the same calculation occurs twice within the peephole of the optimizer, we can often eliminate the second calculation:
$$\begin{aligned} r2 &:= r1 \times 5 \\ r2 &:= r2 + r3 \\ r3 &:= r1 \times 5 \end{aligned}$$

becomes

$$\begin{aligned} r4 &:= r1 \times 5 \\ r2 &:= r4 + r3 \\ r3 &:= r4 \end{aligned}$$
 - Often, as shown here, an extra register will be needed to hold the common value

Peephole Optimization

- It is natural to think of common subexpressions as something that could be eliminated at the source code level, and programmers are sometimes tempted to do so
- The following, for example,

```
x = a + b + c;
```

```
y = a + b + d;
```

could be replaced with

```
t = a + b;
```

```
x = t + c;
```

```
y = t + d;
```


Peephole Optimization

- *Copy propagation*

- Even when we cannot tell that the contents of register b will be constant, we may sometimes be able to tell that register b will contain the same value as register a

- replace uses of b with uses of a , so long as neither a nor b is modified

$r2 := r1$

$r3 := r1 + r2$

$r2 := 5$

becomes

$r2 := r1$

$r3 := r1 + r1$

$r2 := 5$

and then

$r3 := r1 + r1$

$r2 := 5$

Peephole Optimization

- *Strength reduction*

- Numeric identities can sometimes be used to replace a comparatively expensive instruction with a cheaper one
 - In particular, multiplication or division by powers of two can be replaced with adds or shifts:

$r1 := r2 \times 2$

becomes

$r1 := r2 + r2$ or $r1 := r2 \ll 1$

$r1 := r2 / 2$

becomes

$r1 := r2 \gg 1$

Peephole Optimization

- *Filling of load and branch delays*

- For example, a value that is loaded may not be usable for several cycles

$r2 := r1 + r2$

$r3 := A$ — load

$r3 := r3 + r2$ — pipeline stall before r3 can be used

- Since different registers are used, we can schedule the load earlier, avoiding the pipeline stall

$r3 := A$ — load

$r2 := r1 + r2$

$r3 := r3 + r2$ — use is late enough to avoid stall

- This optimization is unnecessary on machines with out of order execution

- Most computers and smartphones, but not necessarily embedded devices

Peephole Optimization

- *Elimination of useless instructions*
 - Instructions like the following can be dropped entirely:
$$r1 := r1 + 0$$
$$r1 := r1 \times 1$$
- *Exploitation of the instruction set*
 - Particularly on CISC machines, sequences of simple instructions can often be replaced by a smaller number of more complex instructions

Redundancy Elimination in Basic Blocks

- Let's look at improving intermediate code generated from this C program for binomial coefficients:

```
combinations(int n, int *A) {  
    int i, t;  
    A[0] = 1;  
    A[n] = 1;  
    t = 1;  
    for (i = 1; i <= n/2; i++) {  
        t = (t * (n+1-i)) / i;  
        A[i] = t;  
        A[n-i] = t;  
    }  
}
```

Redundancy Elimination in Basic Blocks

- We employ a medium level intermediate form (IF) for control flow
 - Every calculated value is placed in a separate register
 - To emphasize virtual registers (of which there is an unlimited supply), we name them $v1, v2, \dots$.
 - We use $r1, r2, \dots$ to represent architectural registers in Section 17.8.

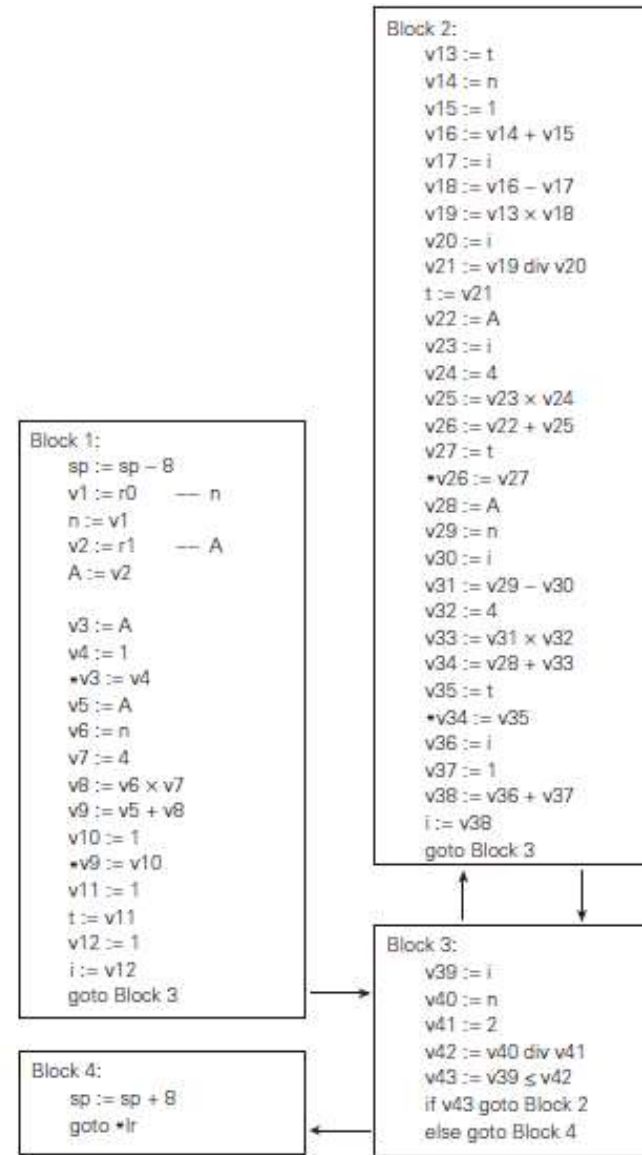


Figure 17.3 Naive control flow graph for the combinations subroutine. Note that reference parameter A contains the address of the array into which to write results; hence we write $v3 := A$ instead of $v3 := \&A$.

Redundancy Elimination in Basic Blocks

- To improve the code within basic blocks, we need to
 - minimize loads and stores
 - identify redundant calculations
- There are two techniques usually employed
 1. translate the syntax tree for a basic block into an *expression DAG* (directed acyclic graph) in which redundant loads and computations are merged into individual nodes with multiple parents
 2. similar functionality can also be obtained without an explicitly graphical program representation, through a technique known as *local value numbering*
- We describe the last technique below

Redundancy Elimination in Basic Blocks

- Value numbering assigns the same name (a “number”) to any two or more symbolically equivalent computations (“values”), so that redundant instances will be recognizable by their common name
- Our names are virtual registers, which we merge whenever they are guaranteed to hold a common value
- While performing local value numbering, we will also implement
 - local constant folding
 - constant propagation, copy propagation
 - common subexpression elimination
 - strength reduction
 - useless instruction elimination

Value Numbering

- Keep track of a table: replace e with reg/imm
 - Replacements are *virtual registers* or *immediates*
 - Virtual registers are numbered v_1, v_2, v_3, \dots
 - Origin of the term “value numbering” – we give each virtual register a number
 - Immediate values
 - i.e. value small enough to fit in the immediate operand of an instruction
 - MIPS architecture: ≤ 16 bits (unsigned value smaller than 65536)
 - Expressions e to replace include:
 - Program variables that are already in a register ($x \rightarrow v_1$)
 - An operand applied to small constants or register ($v_1 + 3 \rightarrow v_2$)
 - A register that duplicates another register ($v_3 \rightarrow v_2$) or holds a small constant value ($v_4 \rightarrow 1$)
 - “large” constants ($100000 \rightarrow v_5$)
 - i.e. too big to fit in the immediate operand of an instruction

Value Numbering

- Keep track of a table: replace e with reg/imm
 - Invariants:
 - Large (non-immediate) values appear alone only on the left
 - Small (immediate) values appear in an expression on the left, or alone on the right
 - No virtual register appears alone on both the left and right
- Procedure
 - Replace expressions with reg/imm according to the table
 - Add what we learn to the table
 - Perform simple optimizations along the way
 - constant folding, strength reduction, useless instruction removal
 - Delay stores (mark variable dirty in table)
 - At end of basic block, store dirty variables
 - Rationale: avoid double-stores

Redundancy Elimination in Basic Blocks

- Let's do value numbering for a simple example:

$v1 := x$

$v2 := 1$

$v3 := v1 + v2$

$y := v3$

$v4 := x$

$v5 := 1$

$v6 := v4 + v5$

$v7 := 3$

$v8 := 1$

$v9 := v7 + v8$

$v10 := v6 * v9$

$v11 := 1$

$v12 := v11 * v10$

$v13 := 100000$

$v14 := v12 + v13$

$y := v14$

What the source might look like:

$y := x + 1;$

$y := (x+1) * (3+1) * 1 + 100000;$

Your Turn: Value Numbering

- Perform value numbering optimization on the following:

$v1 := x$

$v2 := 3$

$v3 := v1 + v2$

$y := v3$

$v4 := 1$

$v5 := x$

$v6 := 2$

$v7 := v4 + v6$

$v8 := v5 + v7$

$v9 := v8 - v3$

$y := v9$