

Optimization

Prof. James L. Frankel
Harvard University

Version of 5:34 PM 30-Mar-2022
Copyright © 2022, 2020, 2018, 2016, 2015 James L. Frankel. All rights reserved.

Reasons to Optimize

- Reduce execution time
- Reduce memory requirements
 - Not so important with very large memories
 - For large data structures, alignment may be an issue
- For multiprocessor systems with local memory, locality of data is extremely important

Optimization Constraints

- Preserve Semantics
- If performed at the IR level, then optimization is both language and machine independent
- Some new IR instructions may be introduced to allow machine-specific optimizations

Optimization Application

- It may be beneficial to apply any one optimization more than once
 - Perhaps both before and after a different optimization has been applied
- It is often important to consider the order in which optimizations are performed

Optimization In Our Projects

- In our implementations, all optimizations will be performed at the IR level
 - That is, each optimization will make a pass over the IR doubly-linked list and will modify the IR list
- In order to more easily generate specific MIPS instructions, create new machine-specific IR instructions
 - For example, these may be created for immediate MIPS instructions
- Always have a compiler mode where the IR can be pretty-printed before and after optimizations
 - Bugs are often introduced into the IR by faulty optimizations
- Allow differing levels of optimization
 - Compare code without optimization to code after differing optimization levels

Degree of Analysis Required

- Peephole Optimizations
 - Requires examining a short sequence (called the *peephole*) of instructions
 - Optimization is performed only within that sequence
 - No deep analysis required
- Optimizations within Basic Blocks
 - Use liveness, next-use, reaching definitions, and other information to perform the analysis
- Global Optimization
 - Across basic blocks
- Interprocedural Optimization

Reaching Definitions

- In many cases, an optimization can be performed only when the variables involved in an expression have not changed since a previous computation
- A *reaching definition* analysis may be useful in these and other cases
- Reaching definition analysis determines where in a program each variable x may have been defined when control reaches point p
- A definition d reaches a point p if there is a path from the point immediately following d to p and that d is not **killed** along that path
 - A definition of variable x is a statement that assigns, or may assign, a value to x

Elimination of Redundant Loads and Stores

- (addressOf, r0, a)
- (loadWord, r1, r0)
- (storeWord, r0, r1)

- The last storeWord is unnecessary
- This optimization may not be performed if the variable is tagged as **volatile**

Common Subexpression Elimination

- May need to apply commutativity and/or associativity to identify these
 - Beware, some operations that are mathematically associative may not be associative as an instruction on a computer (*e.g.*, floating point operations)
- Can arise from subscripting (*i.e.*, addition of an integral value to a pointer)

Copy Propagation

- $b = a$
- $c = b$

Dead/Unreachable Code Elimination

- Code may be present after a goto, return, break, or continue
- Code may be present after a conditional operator and the value of the operand may be determinable at compile time
 - The operand may be a preprocessor symbol
 - The operand may be a const-qualified identifier
 - The operand may be determined after constant folding or evaluation at compile time

Flow-of-Control Optimization

- Replace branches/jumps to branches/jumps with a direct branch/jump
 - Also called **jump threading**
- Remove unnecessary branches/jumps to next location
 - Example:
 - (goto, _UserLabel_function_target)
 - (label, _UserLabel_function_target)
 - becomes:
 - (label, _UserLabel_function_target)
 - Notes
 - label IRs do not produce any executable code
 - label IRs may be referred to elsewhere – hence, they cannot be removed

Machine-Specific Optimizations

- Use of branch rather than jump when appropriate
- Use of offset field in load and store instructions
 - Instead of load address followed by load or store
- Use of immediate instructions when possible
- Use of autoincrement or autodecrement addressing modes if they are present in the target instruction set
- Use of instructions to manipulate a hardware-supported stack if they are present in the target instruction set

Constant Folding

- Evaluate expressions composed of constants known at compile time

Code Motion

- Move code out of loops if it is loop invariant
- This may include expressions that are solely based on variables that do not change inside a loop

Reduction in Strength

- If a lower-strength operator is faster and has the same semantics, use it instead
- $a*2$ becomes $a+a$ or $a<<1$
- unsigned int ui ;
 $ui/2$ becomes $ui>>1$
- $a*4$ becomes $a<<2$

Induction Variables

- An induction variable is a variable whose value is incremented or decremented by a constant value for each iteration of a loop
- Multiple induction variables may exist in a single loop

Induction Variables and Reduction in Strength

```
sum = 0;
for(i = 0; i < 100; i++) {
    sum += a[i];
}
```

becomes

```
sum = 0;
p = &a[0];
for(i = 0; i < 100; i++) {
    sum += *p++;
}
```

Induction Variables and Reduction in Strength

```
sum = 0;
p = &a[0];
for(i = 0; i < 100; i++) {
    sum += *p++;
}
```

becomes

```
sum = 0;
for(p = &a[0]; p < &a[100]; p++) {
    sum += *p;
}
```

Identities

- Operations on an identity for a particular operator can be removed
- $a+0$
- $a-0$
- $a*1$
- $a/1$
- $a\ll 0$
- $a\gg 0$
- $a\&-1$
- $a|0$
- a^0
- etc.

Algebraic Simplification

- Some operations can be performed at compile time
 - $a - a$
 - a / a
 - a^a
 - $a \& a$
 - $a | a$
 - etc.

Inlining of Functions

- Removes the overhead of calling and returning from the function
- Allows more straight-line code to be optimized without requiring interprocedural analysis
- Increases code size
 - This may reduce cache efficiency
- Not running code at the caller and at the callee may improve cache performance
 - It is possible that the caller's code and the callee's code are mapped to the same cache line

Loop Reordering

- Accessing data memory in a sequential order may decrease access time to those variables making the program run faster
- For example, accessing elements of an array in the order in which they are laid out in memory is very helpful
 - Row-major order
 - Rightmost indices vary faster in consecutive memory locations
 - C, C++, Python use this ordering
 - Column-major order
 - Leftmost indices vary faster in consecutive memory locations
 - OpenGL, MATLAB, R, Fortran use this ordering

Loop Unrolling

- Provides more straight-line code for optimization without requiring global analysis
- Decreases the penalty for branching back to the beginning of the loop

Array Alignment/Padding/Layout

- Makes access to array elements faster

Instruction Scheduling

- Pipelining can benefit from improved ordering of instructions

Tail Recursion Elimination

- A recursive call to a function that appears as the last operation may be able to be replaced by iteration within the function
- Present
 - factorialTailRecursive.c
 - factorialTailRecursiveOptimized.c

Low-Level MIPS Optimizations

- Even though we are performing our optimizations at the IR level, my example will show the resulting changes at the MIPS assembler code level

Utilize MIPS **addiu** Instruction

- Instead of using the **li** pseudo-instruction to load a constant into a register followed by accessing that register in the **addu** instruction, we can directly access a constant in an **addiu** instruction
 - The **li** pseudo-instruction is able to load a full 32-bit constant into a register by generating more than one MIPS instruction, if necessary
 - But, the range of the immediate field in **addiu** is limited – it's a 16-bit sign-extended immediate field – so this transformation can't be applied with constants that are out of range

- Using the **addiu** instruction, we can convert:

```
li      $s0, 5           # $s0 <- 5
addu    $s2, $s1, $s0    # $s2 <- $s1+5
```

- Into:

```
li      $s0, 5           # $s0 <- 5
addiu  $s2, $s1, 5      # $s2 <- $s1+5
```

- The code that has been changed is highlighted in **red**
- Note that in order to apply this optimization, **\$s0** cannot be modified between the **li** and the **addu** instructions
- The **li** instruction is not removed yet because we don't know if **\$s0** is used elsewhere

Example One of Low-Level MIPS Immediate Optimizations

- Start with the following C program:

```
int a, b;  
int main(void) {  
    ...  
    a = b+5;  
    ...  
}
```

- Straight-forward code generation will yield:

```
la    $s0, _Global_a # $s0 -> a  
la    $s1, _Global_b # $s1 -> b  
li    $s2, 5          # $s2 <- 5  
lw    $s3, ($s1)      # $s3 <- b  
addu  $s4, $s3, $s2   # $s4 <- b+5  
sw    $s4, ($s0)      # a <- b+5
```

Low-Level Optimization Using **addiu** Instruction

- If we use the **addiu** instruction, we have:

```
la    $s0, _Global_a    # $s0 -> a
la    $s1, _Global_b    # $s1 -> b
li    $s2, 5             # $s2 <- 5
lw    $s3, ($s1)        # $s3 <- b
addiu $s4, $s3, 5      # $s4 <- b+5
sw    $s4, ($s0)        # a <- b+5
```

Apply Dead/Unreachable Code Elimination

- If we apply dead code elimination, we have:

```
la    $s0, _Global_a    # $s0 -> a
la    $s1, _Global_b    # $s1 -> b
lw    $s3, ($s1)        # $s3 <- b
addiu $s4, $s3, 5       # $s4 <- b+5
sw    $s4, ($s0)        # a <- b+5
```

- The **li** instruction is now removed

Example Two of Several Low-Level MIPS Optimizations with Local Variables

- Start with the following C program:

```
int a, b, c;  
b = a;  
c = a;
```

- Lets assume stack offsets of 96 for a, 100 for b, and 104 for c
- Straight-forward code generation will yield:

```
la    $s0, 100($fp)    # $s0 -> b  
la    $s1, 96($fp)     # $s1 -> a  
lw    $s2, ($s1)       # $s2 <- a  
sw    $s2, ($s0)       # b <- a  
la    $s3, 104($fp)   # $s3 -> c  
la    $s4, 96($fp)    # $s4 -> a  
lw    $s5, ($s4)       # $s5 <- a  
sw    $s5, ($s3)       # c <- a
```

Low-Level MIPS Optimizations – Using the Offset Field in **lw** Instructions

- Using the offset field in **lw** instructions, we can convert:

```
la    $s1, 96($fp)    # $s1 -> a
lw    $s2, ($s1)      # $s2 <- a
```

- Into:

```
la    $s1, 96($fp)    # $s1 -> a
lw    $s2, 96($fp)    # $s2 <- a
```

- The code that has been changed is highlighted in **red**
- Note that in order to apply this optimization, \$s1 cannot be modified between the **la** and the **lw** instructions
- The **la** instruction is not removed yet because we don't know if \$s1 is used elsewhere

Apply Using the Offset Field in **lw** Instructions

- If we apply using the offset field in **lw** instructions, we have:

```
la    $s0, 100($fp) # $s0 -> b
la    $s1, 96($fp)  # $s1 -> a
lw    $s2, 96($fp)  # $s2 <- a
sw    $s2, ($s0)    # b <- a
la    $s3, 104($fp) # $s3 -> c
la    $s4, 96($fp)  # $s4 -> a
lw    $s5, 96($fp)  # $s5 <- a
sw    $s5, ($s3)    # c <- a
```

- The code that has been changed is highlighted in **red**

Apply Using the Offset Field in **sw** Instructions

- If we apply a similar transformation by using the offset field in **sw** instructions, we have:

```
la    $s0, 100($fp) # $s0 -> b
la    $s1, 96($fp)  # $s1 -> a
lw    $s2, 96($fp)  # $s2 <- a
sw    $s2, 100($fp) # b <- a
la    $s3, 104($fp) # $s3 -> c
la    $s4, 96($fp)  # $s4 -> a
lw    $s5, 96($fp)  # $s5 <- a
sw    $s5, 104($fp) # c <- a
```

- The code that has been changed is highlighted in **red**

MIPS code produced from **la** Pseudo-Instruction (1 of 2)

- If the form of an **la** pseudo-instruction is:

```
la    $s0, 100($fp)
```

- Then, SPIM will generate the following MIPS instruction to implement it:

```
addi  $s0, $fp, 100
```

- For the following **la** pseudo-instruction:

```
la    $s0, 65536($fp)
```

- SPIM will generate the following MIPS instructions to implement it:

```
lui   $1, 1  
add   $s0, $fp, $1
```

MIPS code produced from **la** Pseudo-Instruction (2 of 2)

- For the following **la** pseudo-instruction:

```
la    $s0, 65540($fp)
```

- SPIM will generate the following MIPS instructions to implement it:

```
lui   $1, 1  
ori   $1, $1, 4  
add   $s0, $fp, $1
```

MIPS code produced from **lw** Instruction

- If the form of an **lw** pseudo-instruction is:

```
lw    $s0, 65540($fp)
```

- Then, SPIM will generate the following MIPS instructions to implement it:

```
lui   $1, 1  
addu  $1, $1, $fp  
lw    $s0, 4($1)
```

- Note that even though **lw** is *not* a pseudo-instruction, SPIM may generate more than one instruction to implement it

Constraint on the Offset Field in **la**, **lw**, and **sw** Instructions

- The MIPS **la** pseudo-instruction is able to generate more than one MIPS instruction in order to load the address of its second operand into a register
 - The **la** pseudo-instruction is able to produce code even with an offset that is out of range for a 16-bit field
- For the **lw** or **sw** instruction to perform the same functionality *without requiring more than one instruction*, the memory address of the data that is being loaded or stored must be accessible through the 16-bit offset field
 - This may limit the stack frame size accessible through an offset in the **lw** or **sw** instructions

Apply Common Subexpression Elimination

- If we apply common subexpression elimination, we have:

```
la    $s0, 100($fp)    # $s0 -> b
la    $s1, 96($fp)     # $s1 -> a
lw    $s2, 96($fp)     # $s2 <- a (line 3)
sw    $s2, 100($fp)    # b <- a
la    $s3, 104($fp)   # $s3 -> c
la    $s4, 96($fp)     # $s4 -> a
lw    $s5, 96($fp)    # $s5 <- a
sw    $s2, 104($fp)   # c <- a (line 8)
```

- The code that has been changed is highlighted in red
- Note that in order to apply this optimization, the user variable **a** cannot be modified between line 3 and line 8 and also that register **\$s2** cannot be modified between the **lw** instruction in line 3 and the **sw** instruction in line 8

Apply Dead/Unreachable Code Elimination

- If we apply dead code elimination, we have:

```
lw    $s2, 96($fp)  # $s2 <- a
sw    $s2, 100($fp) # b <- a
sw    $s2, 104($fp) # c <- a
```

Assign Final Registers to the Resulting Code

- If we apply some register assignment algorithm – perhaps using graph coloring, we have:

```
lw    $s0, 96($fp)  # $s0 <- a
sw    $s0, 100($fp) # b <- a
sw    $s0, 104($fp) # c <- a
```

- Now eight lines of code has been optimized to three
- Now only one register is required
- This code is much more efficient than our initial straight-forward code

Example Three of Several Low-Level MIPS Optimizations with Global Variables

- Start with the following C program:

```
int a, b, c;
int main(void) {
    b = a;
    c = a;
    ...
}
```

- Straight-forward code generation will yield:

```
la    $s0, _Global_b # $s0 -> b
la    $s1, _Global_a # $s1 -> a
lw    $s2, ($s1)      # $s2 <- a
sw    $s2, ($s0)      # b <- a
la    $s3, _Global_c # $s3 -> c
la    $s4, _Global_a # $s4 -> a
lw    $s5, ($s4)      # $s5 <- a
sw    $s5, ($s3)      # c <- a
```

Comparing Example Three to Example Two

- The assembly code is nearly identical, but Example Three references the global variables by name whereas Example Two references the local variables through offsets off the \$fp register
- In order to produce efficient optimized code, the \$gp (global pointer) register has to be utilized to access global variable
 - The \$gp register will be initialized to point into the static data segment
 - For maximum addressability, \$gp would point in the middle of a 64K byte memory region
 - For simplicity in our code, I will make \$gp point to the beginning of the static data segment

Layout of Variables in the Static Data Segment

- There are three global **int** variables: **a**, **b**, and **c**
- They are laid out in memory in ascending locations

Global variable	Offset in Data Segment
a	0
b	4
c	8

Low-Level MIPS Optimizations – Using the Offset Field in **la** & **lw** Instructions

- Using the `$gp` register and the offset field in **lw** instructions, we can convert:

```
la    $s1, Global_a    # $s1 -> a
lw    $s2, ($s1)       # $s2 <- a
```

- Into:

```
la    $gp, Global_a    # $gp -> static data segment
la    $s1, 0($gp)       # $s1 -> a
lw    $s2, 0($gp)       # $s2 <- a
```

- The code that has been changed is highlighted in **red**
- Register `$gp` should be loaded with the address of the static data segment ***only once at the beginning of main***
- Note that in order to apply this optimization, `$s1` cannot be modified between the **la** and the **lw** instructions
- The **la** instruction for `$s1` is not removed yet because we don't know if `$s1` is used elsewhere

Apply Using the Offset Field in **la** & **lw** Instructions

- If we apply using the offset field in **la** & **lw** instructions, we have:

```
la    $gp, _Global_a      # $gp -> static data segment
la    $s0, 4($gp)         # $s0 -> b
la    $s1, 0($gp)         # $s1 -> a
lw    $s2, 0($gp)         # $s2 <- a
sw    $s2, ($s0)          # b <- a
la    $s3, 8($gp)         # $s3 -> c
la    $s4, 0($gp)         # $s4 -> a
lw    $s5, 0($gp)         # $s5 <- a
sw    $s5, ($s3)          # c <- a
```

- The code that has been changed is highlighted in **red**

Apply Using the Offset Field in **sw** Instructions

- If we apply a similar transformation by using the offset field in **sw** instructions, we have:

```
la    $gp, _Global_a      # $gp -> static data segment
la    $s0, 4($gp)         # $s0 -> b
la    $s1, 0($gp)         # $s1 -> a
lw    $s2, 0($gp)         # $s2 <- a
sw    $s2, 4($gp)        # b <- a
la    $s3, 8($gp)         # $s3 -> c
la    $s4, 0($gp)         # $s4 -> a
lw    $s5, 0($gp)         # $s5 <- a
sw    $s5, 8($gp)        # c <- a
```

- The code that has been changed is highlighted in **red**

Apply Common Subexpression Elimination

- If we apply common subexpression elimination, we have:

```
la    $gp, _Global_a           # $gp -> static data segment
la    $s0, 4($gp)              # $s0 -> b
la    $s1, 0($gp)              # $s1 -> a
lw    $s2, 0($gp)              # $s2 <- a (line 4)
sw    $s2, 4($gp)              # b <- a
la    $s3, 8($gp)              # $s3 -> c
la    $s4, 0($gp)              # $s4 -> a
lw    $s5, 0($gp)              # $s5 <- a
sw    $s2, 8($gp)              # c <- a (line 9)
```

- The code that has been changed is highlighted in **red**
- Note that in order to apply this optimization, the user variable **a** cannot be modified between line 4 and line 9 and also that register **\$s2** cannot be modified between the **lw** instruction in line 4 and the **sw** instruction in line 9

Apply Dead/Unreachable Code Elimination

- If we apply dead code elimination, we have:

```
la    $gp, _Global_a    # $gp -> static data segment
lw    $s2, 0($gp)       # $s2 <- a
sw    $s2, 4($gp)       # b <- a
sw    $s2, 8($gp)       # c <- a
```

Assign Final Registers to the Resulting Code

- If we apply some register assignment algorithm – perhaps using graph coloring, we have:

```
la    $gp, _Global_a      # $gp -> static data segment
lw    $s0, 0($gp)         # $s0 <- a
sw    $s0, 4($gp)         # b <- a
sw    $s0, 8($gp)         # c <- a
```

- Now eight lines of code has been optimized to three
- Now only one register is required
- This code is much more efficient than our initial straight-forward code