Lecture 22-23
Optimization + MERL

CS 241: Foundations of Sequential Programs
Spring 2017

Troy Vasiga et al. (where Dan Holtby ∈ al)
University of Waterloo
Optimization

Large and interesting topics: we will just skim. Recall that we can have an infinite number of MIPS programs that are equivalent to a particular WLP4 program. What are the “good” MIPS programs that we would like to generate?

- Increasing speed of given WLP4 code:

  Hard. You need to know about pipelines and all that other CS251 stuff.

- Reducing number of instructions used for given WLP4 code:

  Easy..er. Easier.
Optimization: A10

You should probably finish A10 before starting on the bonus. That way you have all your tests prepared and can ensure your optimization strategies don’t break anything!

Two major approaches in our (syntax-driven) code generation

▶ Rewrite the parse tree (e.g. constant folding on next slide)
▶ Improve the code() function (e.g. register allocation)
▶ Rewrite the assembly you’ve already generated

Three, there are three major approaches
Optimization: Constant Folding

Consider an expression 5+3.

<table>
<thead>
<tr>
<th>What our compiler currently does</th>
<th>What it could do:</th>
</tr>
</thead>
<tbody>
<tr>
<td>lis $3</td>
<td>lis $3</td>
</tr>
<tr>
<td>.word 5</td>
<td>.word 8</td>
</tr>
<tr>
<td>.push $3</td>
<td></td>
</tr>
<tr>
<td>lis $3</td>
<td></td>
</tr>
<tr>
<td>.word 3</td>
<td></td>
</tr>
<tr>
<td>.word 3</td>
<td></td>
</tr>
<tr>
<td>.pop $5</td>
<td></td>
</tr>
<tr>
<td>add $3, $5, $3</td>
<td></td>
</tr>
</tbody>
</table>
Optimization: Constant Folding

This can be done to the parse tree itself.

It can be part of type checking. The type of “3” an “5” is int, and these expressions are “compile-time constants”.

An operation where both operands are compile-time constants is also a compile-time constant value (and can be replaced with its value).

You can replace the entire “3 + 5” sub-tree with a single “8” node.
int foo() {
    int x = 1;
    return x+x; }

In WLP4 all variables are initialized with compile-time constants. x is “constant” until it is first assigned to (in the example, it is never assigned to).

The parse tree can be rewritten as
int foo() {
    int x = 1;
    return 2;
}

Further, x is no longer used, so int foo() { return 2; }
Let’s say you had:
(a+b) * (a+b)

You could imagine this as

```
temp = a+b;
temp * temp;
```

Caution: Can’t do this if the expression has side effects (in WLP4, that means a procedure call or new[])

PRO-TIP: You can look for that sub-expression throughout the entire procedure, not just a single expression.
Caution: What if a or b changes? Need to refresh temp.
Optimization: Dead-code elimination

What makes code dead?
- It will never be executed
- It is executed but doesn’t alter the behaviour of the program

How do you get dead code in WLP4?
- if test that is always the same
- while test that is always false
- a calculation that is not used (e.g. \( x = \) (complex expression); but \( x \) is not used again)
- Watch for side-effects (\( x = f(y) \) might look “dead” if you never use \( x \), but \( f \) should still be called!)

We do not have true or false tokens in WLP4, but if you have a test where both operands are compile-time constants, you essentially have true / false
Optimization: Register allocation

- Notice that many registers (i.e., $14$ through $28$) are not used by our compilation system.
- Notice that using registers would save instructions.

To save/load a value in memory: `sw / lw` (1 instruction)

To save/load a value in a register: it’s already in a register! (0 instructions)

- Suppose we could store 15 variables. Which ones would we store?
  - Try your options and see which makes the most efficient code? If you have 30 variables and want to save 15, that’s 30 choose 15 = 155,117,520 options!
  - The most commonly used?
Optimization: Register allocation for temporaries

If we use registers for temporaries it saves us more instructions. Why?

Pushing a temporary value: 3 instructions
Popping a temporary value: 3 instructions
Using a register as a temporary: 2 instructions (move from $3$, move to $5$)
Or: None instructions? (modify code() to let you pick a different destination than $3$).

```c
int code(expr → expr PLUS term, dest, free_reg) {
    Register temp = something from free_reg
    return code(expr,dest,free_reg) +
        code(term,temp,free_reg minus dest) +
        "add $dest, $dest, $temp"
}
```

Problems? Improvements?
Optimization: Strength Reduction

- \{expr\} * 2
- Many instructions (lis to get the 2, mult and mflo to do the math, maybe some stack work for temporaries?)

- Consider \{expr\} + \{expr\} instead
- Isn’t that worse? Not if you did “common subexpression elimination! Now it’s “add $3, $3, $3”
### Optimization: Inlining Procedures

```c
int f(int x) { return x+x; }
int wain(int a, int b) { return f(a); }

int wain(int a, int b) { return a+a; }
```

**Pros:**
- No instructions for procedure call (no pushing / popping, saving $29, etc.)
- No instructions for saving callee-save registers
- Might mean you can skip saving $31 too

**Cons:**
- One copy of body per call, instead of one copy per program
- What if f contains statements? (Substantial rewrite of code needed?)
In WLP4, we must have exactly one `return` as the last statement in every function.

In other programming languages, you can say something like:

```c
int fact(int n, int a) {
    if (n==0) return a;
    else return fact(n-1, n*a);
}
```

Notice that the last thing the function does is returning a value: there is no additional work to be done. Thus, the frame can be reused!
MERL is our flavour of object file.

MIPS
Executable
Relocatable
Linkable

- Mostly machine code (MIPS executable)
- Has a table of "\.word LABEL" that can be adjusted if the program is not loaded at address 0 (relocatable)
- Has a table of uses of .imported and .exported labels (linkable)
A MERL file has 3 parts

- **Header** = 3 words
  - Magic number 0x10000002 a.k.a `beq $0, $0, 2`
  - length of code + header
  - length of code + header + footer

- **Body** = MIPS machine code assembled to be loaded at address 0xC (12).
  - Or, assembled to be loaded at address 0x0, but including the header

- **Footer** = table of interesting words in the body
Relocation

0x00    lis $11
0x04    .word 1
0x08    lis $20
0x0C    .word top
        top:
0x10    slt $7, $1, $11
0x14    beq $7, $0, end
0x18    mult $1, $3
0x1C    mflo $3
0x20    sub $5, $5, $11
0x24    jr $20
end:

Q: If this was meant to be loaded at address $\alpha$ instead of 0, which words change?
A: Just 0x0C. (0x14 is a relative reference to a label, so it doesn’t depend on $\alpha$)

0x00000001 ; REL
0x0000000C ; address (except not)
## Relocation - cont.

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>;;; MERL</td>
<td>0x1000 0002</td>
</tr>
<tr>
<td>0x04</td>
<td>;;; HEADER</td>
<td>0x0000 003C</td>
</tr>
<tr>
<td>0x08</td>
<td>;;; WILL GO HERE</td>
<td>0x0000 0034</td>
</tr>
<tr>
<td></td>
<td>__________________________</td>
<td>-------------</td>
</tr>
<tr>
<td>0x0C</td>
<td>lis $11</td>
<td>0x0000 5814</td>
</tr>
<tr>
<td>0x10</td>
<td>.word 1</td>
<td>0x0000 0001</td>
</tr>
<tr>
<td>0x14</td>
<td>lis $20</td>
<td>0x0000 A014</td>
</tr>
<tr>
<td>0x18</td>
<td>.word top</td>
<td>0x0000 001C</td>
</tr>
<tr>
<td></td>
<td><strong>top:</strong></td>
<td></td>
</tr>
<tr>
<td>0x1C</td>
<td>slt $7, $1, $11</td>
<td>0x002B 382A</td>
</tr>
<tr>
<td>0x20</td>
<td>beq $7, $0, end</td>
<td>0x10E0 0004</td>
</tr>
<tr>
<td>0x24</td>
<td>mult $1, $3</td>
<td>0x0023 0018</td>
</tr>
<tr>
<td>0x28</td>
<td>mflo $3</td>
<td>0x0000 1812</td>
</tr>
<tr>
<td>0x2C</td>
<td>sub $1, $1, $11</td>
<td>0x002B 0822</td>
</tr>
<tr>
<td>0x30</td>
<td>jr $20</td>
<td>0x0280 0008</td>
</tr>
<tr>
<td></td>
<td><strong>end:</strong></td>
<td></td>
</tr>
<tr>
<td>0x34</td>
<td>;;; MERL FOOTER</td>
<td>0x0000 0001</td>
</tr>
<tr>
<td>0x38</td>
<td>;;; WILL GO HERE</td>
<td>0x0000 00018</td>
</tr>
</tbody>
</table>
Lets say you have a MERL file (you have a MERL file). Your OS can’t load it address 0, instead it must find free memory where the program will fit.

read header
α = findFreeRAM(codeLength)
for each instruction
    MEM[α+i] = instruction
for each relocation entry
    MEM[α+location] += α
place α into $19
jalr $19
A Relocating Assembler - cs241.relasm

You can change your A4 code so that it emits MERL instead of raw MIPS code.

Changes to pass 1:

- Start counting at 12 instead of 0 (the first instruction will be loaded at address 0xC after all)
- Keep a “reference list”, a list (or vector, or set) of the locations of all “.word {label}” instructions

Changes to pass 2:

- Emit the header before emitting the machine code
  - The “Code length” is the number of instructions in the body + 3 for the header (then all times 4 to convert from words to bytes)
  - The “File length” is the code length plus the footer length
  - The footer length is 8 times the number of entries in the “reference list”

- Emit the footer after emitting the machine code
  - For each element of the “reference list”, emit the word 1, then the element of the list
Combining Programs

- Why? (Modularization, see CS136)
- When would we want to assemble? (Independent from other modules)
- Consider:
  - main calls fred and derf
  - main contains x (needed by both fred and derf)
  - derf contains y (needed by both fred and derf)
We could combine 3 assembly files by concatenating them together.
That’s easy. But while an assembler is pretty fast, it’s still a waste of
effort to keep re-assembling a bunch of modules that haven’t changed!
Assemble each separately and then put all the MERL files together

This means relocating, because code for the second file will be below code for the first.

- MERL has REL entries that can be used to solve that problem.

Not quite enough. In the previous example, we need to where the main.merl file uses the derf symbol so we can fill the right value in after linking.

We also need to know where the derf label is in derf.merl so that we know what the “right value” is.
Linking

There are two other kinds of table entry. That means a total of three magic numbers to look for:

- **REL - Relocation Entry**
  - Indicates that a `.word` is a reference to a label and must be relocated
  - Magic number is 0x1
  - Followed by location of `.word`

- **ESR - External Symbol Reference Entry**
  - Indicates that a `.word` is a reference to a symbol that has been imported from another file
  - There will be one ESR entry per reference to that symbol
  - Magic number is 0x11 (17 decimal)
  - Followed by location of `.word`, then length of label’s name, then ASCII for name

- **ESD - External Symbol Definition Entry**
  - Indicates that an instruction is labelled with a label that has been exported from this file
  - There will be one ESD entry per exported symbol
  - Magic number is 0x5
  - Followed by location of `label`, then length of label’s name, then ASCII for name
Linking - ESR

0x00
0x04
0x08
--------
0x0C
0x10
0x14
0x18
0x1C
0x20
--------
0x24
0x28
0x2C
0x30
0x34
0x38
0x3C
0x40

;;; HEADER

.import print
lis $3
.word print
add $5, $31, $0
jalr $3
add $31, $5, $0
jr $31

--------

;;; FOOTER

0x0000 0024
0x0000 0044
0x0000 0010 ; Address
0x0000 0005 ; Length
0x0000 0011 ; ESR
0x0000 0009
0x0000 0008
0x0000 0000
0x0000 1814
0x03E0 2820
0x0060 0009
0x00A0 F820
0x03E0 0008
0x0000 0002
0x0000 0000
0x03E0 2820
0x0060 0009
0x00A0 F820
0x03E0 0008
0x0000 0011 ; ESR
0x0000 0010 ; Address
0x0000 0005 ; Length
0x0000 0070 ; 'p'
0x0000 0072 ; 'r'
0x0000 0069 ; 'i'
0x0000 006e ; 'n'
0x0000 0074 ; 't'

24
<table>
<thead>
<tr>
<th>Address</th>
<th>Assembler Code</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>;;; HEADER</td>
<td>0x1000 0002</td>
</tr>
<tr>
<td>0x04</td>
<td>;;; GOES</td>
<td>0x0000 0028</td>
</tr>
<tr>
<td>0x08</td>
<td>;;; HERE</td>
<td>0x0000 0010</td>
</tr>
<tr>
<td>0x0C</td>
<td>.export foo</td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td>foo: jr $31</td>
<td>0x03E0 0008</td>
</tr>
<tr>
<td>0x14</td>
<td>;;; FOOTER</td>
<td></td>
</tr>
<tr>
<td>0x18</td>
<td>;;; GOES</td>
<td></td>
</tr>
<tr>
<td>0x1C</td>
<td>;;; HERE</td>
<td></td>
</tr>
<tr>
<td>0x20</td>
<td>;;;</td>
<td></td>
</tr>
<tr>
<td>0x24</td>
<td>;;;</td>
<td></td>
</tr>
</tbody>
</table>

Note: The above table shows the hexadecimal addresses and corresponding assembly code. The ESD and other annotations are included for reference.
A Linkable Assembler

Pass 1 Changes:

- The “reference list” from the relocating assembler should also keep track of which symbol (string) it is a reference to.
- Keep track of the set of .imported symbols.
- Keep track of the set of .exported symbols.
- If a symbol is .imported and also defined, that’s a duplicate definition error.

Pass 2 Changes:

- A symbol is only undefined if it is not in the symbol table AND not in the list of .imported symbols.
- A .word {label} has the value 0 if the label is .imported (really you can use any dummy value here, but ours uses 0).
- For each pair (location, symbol) in the reference list, emit a REL entry if the symbol is in the symbol table, and an ESR entry if it is .imported.
- For each symbol in the list of .exported symbols, emit an ESD entry (you can get the location from the symbol table).
  - If it’s not in the symbol table, it’s an undefined label error.
A linker

To link two MERL files together:

1. Place the code (including header) from file 1 first
2. Place the code (without header) from file 2 after
3. Use REL entries from file 2 to relocate all references in file 2 (since it was just relocated!)
4. Update REL entries from file 2 (they’ll need to go into the new footer)
5. Update ESR and ESD entries from file 2, too
6. For each pair of ESR and ESD entries that refer to the same symbol:
   6.1 That symbol is defined at ESD.location
   6.2 There is a reference to that symbol at ESR.location
   6.3 The word at ESR.location should be replaced with the value ESD.location
   6.4 Now that this is a regular label reference, the ESR entry should be replaced with a REL entry
   6.5 (Leave the ESD entry alone, it may be needed for future linking)
7. Now that you’ve replaced all satisfied ESR entries with REL entries you update the header with the correct sizes.

See CS241 website for a more detailed description of the algorithm