Compiler construction 2011

Lecture 10

More on code optimization

- IR code optimizations revisited: loops
- Native code generation revisited
  - Instruction selection
  - Instruction scheduling

A motivating example

A simple Javalette function (in extension arrays1)

```
int sum (int [] a) {
  int res=0;
  for (int x : a)
    res = res + x;
  return res;
}
```

What code would you generate?

Possible naive LLVM code, part 1

```
%arr = type { i32, [ 0 x i32 ] }*
define i32 @sum(%arr %__p__a) {
entry:  %a = alloca %arr
        store %arr %__p__a , %arr* %a
        %_res_t0 = alloca i32
        store i32 0 , i32* %_res_t0
        %_x_t1 = alloca i32
        %t2 = load %arr* %a
        %t3 = getelementptr %arr %t2 , i32 0, i32 0
        %t4 = load i32* %t3
        %_indexx_t5 = alloca i32
        store i32 0 , i32* %_indexx_t5
        br label %lab0
lab0:   %t6 = load i32* %_indexx_t5
        %t7 = icmp slt i32 %t6 , %t4
        br i1 %t7 , label %lab1 , label %lab2
```


Possible naive LLVM code, part 2

```
lab1: %t8 = getelementptr %arr %t2 , i32 0,  i32 1, i32 %t6
      %t9 = load i32* %t8
      store i32 %t9 , i32* %_x_t1
      %t10 = load i32* %_res_t0
      %t11 = load i32* %_x_t1
      %t12 = add i32 %t10 , %t11
      store i32 %t12 , i32* %_res_t0
      %t13 = add i32 %t6 , 1
      store i32 %t13 , i32* %_indexx_t5
      br label %lab0

lab2: %t14 = load i32* %_res_t0
      ret  i32 %t14
}
```

After `opt -mem2reg`

```
define i32 @sum(%arr %__p__a) {
entry: %t3 = getelementptr %arr %__p__a, i32 0, i32 0
       %t4 = load i32* %t3
       br label %lab0

lab0: %_res_t0.0 = phi i32 [ 0, %entry ], [ %t12, %lab1 ]
      %_indexx_t5.0 = phi i32 [ 0, %entry ], [ %t13, %lab1 ]
      %t7 = icmp slt i32 %_indexx_t5.0, %t4
      br i1 %t7, label %lab1, label %lab2

lab1: %t8 = getelementptr %arr %__p__a, i32 0, i32 1, i32 %_indexx_t5.0
      %t9 = load i32* %t8
      %t12 = add i32 %_res_t0.0, %t9
      %t13 = add i32 %_indexx_t5.0, 1
      br label %lab0

lab2: ret i32 %_res_t0.0
}
```


After `opt -std-compile-opts`

```
define i32 @sum(%arr nocapture %__p__a) nounwind readonly {
entry: %t3 = getelementptr %arr %__p__a, i32 0, i32 0
       %t4 = load i32* %t3
       %t71 = icmp sgt i32 %t4, 0
       br i1 %t71, label %bb.nph, label %lab2

bb.nph: %tmp = zext i32 %t4 to i64
        br label %lab1

lab1: %indvar = phi i64 [ 0, %bb.nph ], [ %indvar.next, %lab1 ]
      %_res_t0.02 = phi i32 [ 0, %bb.nph ], [ %t12, %lab1 ]
      %t8 = getelementptr %arr %__p__a, i64 0, i32 1, i64 %indvar
      %t9 = load i32* %t8
      %t12 = add i32 %t9, %_res_t0.02
      %indvar.next = add i64 %indvar, 1
      %exitcond = icmp eq i64 %indvar.next, %tmp
      br i1 %exitcond, label %lab2, label %lab1

lab2: %_res_t0.0.lcssa = phi i32 [ 0, %entry ], [ %t12, %lab1 ]
      ret i32 %_res_t0.0.lcssa
}
```


Generated x86 assembly (with `llc`)

```
_sum:   push    EDI
        push    ESI
        mov     ECX, DWORD PTR [ESP + 12]
        mov     EDX, DWORD PTR [ECX]
        test    EDX, EDX
        jg      LBB0_2
        xor     EAX, EAX
        jmp     LBB0_4
LBB0_2: xor     ESI, ESI
        add     ECX, 4
        xor     EAX, EAX
LBB0_3: add     EAX, DWORD PTR [ECX]
        add     EDX, -1
        adc     ESI, -1
        add     ECX, 4
        mov     EDI, EDX
        or      EDI, ESI
        jne     LBB0_3
LBB0_4: pop     ESI
        pop     EDI
        ret
```

Comments

- No local vars; no stack frame handling.
- Uses callee save registers EDI and ESI; note save/restore.
- ECX holds address of current array elem; increased by 4 in each iteration.
- EDX counts nr of elems remaining.
- Use of ESI in loop termination test??

Optimizations of loops

In computationally demanding applications, most of the time is spent in executing (inner) loops.

Thus, an optimizing compiler should focus its efforts in improving loop code.

The first task is to identify loops in the code. In the source code, loops are easily identified, but how to recognize them in a low level IR code?

A loop in a CFG is a subset of the nodes that

- has a **header** node, which dominates all nodes in the loop.
- has a **back edge** from some node in the loop back to the header. A back edge is an edge where the head dominates the tail.

Moving loop-invariant code out of the loop

A simple example

```
for (i=0; i<n; i++)
  a[i] = b[i] + 3*x;
```

should be replaced by

```
t = 3*x;
for (i=0; i<n; i++)
  a[i] = b[i] + t;
```

We need to insert an extra node (a **pre-header**) before the header.

Not quite as simple

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    a[i][j] = b[i][j]+10*i+3*x;
```

should be replaced by

```
t = 3*x;
for (i=0; i<n; i++) {
  u = 10*i + t;
  for (j=0; j<n; j++)
    a[i][j] = b[i][j] + u;
}
```


Induction variables

A **basic** induction variable is an (integer) variable which has a single definition in the loop body, which increases its value with a fixed (loop-invariant) amount.

Example: `n = n + 3`

A basic IV will assume values in arithmetic progression when the loop executes.

Given a basic IV we can find a collection of **derived** IV's, each of which has a single def of the form
`m` = a*n+b;
where `a` and `b` are loop-invariant.
The def can be extended to allow RHS of the form `a*k+b` where also `k` is an already established derived IV.

Strength reduction for IV's

`n` is a basic IV (only def is to increase by 1).
`k` is derived IV.

Replace multiplication involved in def of `k` by addition.

```
while (n<100) {
   k = 7*n + 3;
   a[k]++;
   n++;
}
```

Replace multiplication involved in def of derived IV by addition.

```
k = 7*n + 3;
while (n<100) {
   a[k]++;
   n++;
   k+=7;
}
```

Could there be some problem with this transformation?

Strength reduction for IV's, continued

The loop might not execute at all, in which case `k` would not be evaluated.
Better to perform loop inversion first.

```
if (n<100) {
   k = 7*n + 3;
   do {
     a[k]++;
     n++;
     k+=7;
   } while (n<100);
}
```

If `n` is not used after the loop, it can be eliminated from the loop

```
if (n<100) {
   k = 7*n + 3;
   do {
     a[k]++;
     k+=7;
   } while (k<703);
}
```

One more example

Sample loop

```
int sum = 0;
for(i=0; i<1000; i++)
  sum += a[i];
```

What can these techniques do for this loop?

Strength reduction/IV techniques

```
    %sum = 0
    %off = 0
    %addr = %addr.a
    %end = add %addr.a,4000
L1: %a.i = load %addr
    %sum = add %sum,%a.i
    %addr = add %addr, 4
    %stop = cmp lt %addr,%end
    br %stop, L1, L2
L2:
```

Naive assembler code

```
    %sum = 0
    %i = 0
L1: %off = mul %i, 4
    %addr = add %addr.a,%off
    %a.i = load %addr
    %sum = add %sum,%a.i
    %i = add %i, 1
    %stop = cmp lt %i,1000
    br %stop, L1, L2
L2:
```


Loop unrolling

```
for (i=0; i<100; i++)
   a[i] = a[i] + x[i]
```

```
for (i=0; i<100; i=i+4) {
   a[i] = a[i] + x[i]
   a[i+1] = a[i+1] + x[i+1]
   a[i+2] = a[i+2] + x[i+2]
   a[i+3] = a[i+3] + x[i+3]
}
```

- In which ways is this an improvement?
- What to do if upper bound is `n`?
- Is unrolling four steps the best choice?
- What could be the disadvantages?

Native code generation, revisited

More complications

So far, we have ignored some important concerns in code generation:

- The instruction set in real-world processors typically offer many different ways to achieve the same effect. Thus, when translating an IR program to native code we must do **instruction selection**, i.e. choose between available alternatives.
- Often an instruction sequence contain independent parts that can be executed in arbitrary order. Different orders may take very different time; thus the code generator must do **instruction scheduling**.

Both these task are complex and interact with **register allocation**.

In LLVM, these tasks are done by the native code generator `llc` and the JIT compiler in `lli`.

Instruction selection

Further observations

- Instruction selection for RISC machines generally simpler than for CISC machines.
- The number of translation possibilities grow (combinatorially) as one considers larger chunks of IR code for translation.

Pattern matching

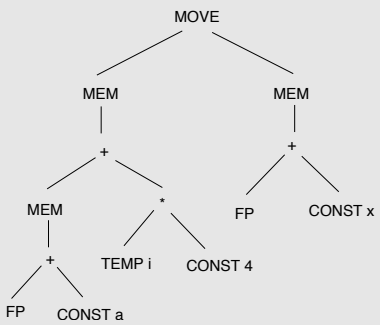
The IR code can be seen as a pattern matching problem: The native instructions are seen as patterns; instruction selection is the problem to cover the IR code by patterns.

Two approaches

- Tree pattern matching. Think of IR code as tree.
- Peephole matching. Think of IR code as sequence.

Tree pattern matching, an example

a[i] := x as tree IR code (from Appel)



a and x local vars, i in register.
a is pointer to first array element.

Algorithm outline

- Represent native instructions as **patterns**, or tree fragments.
- **Tile** the IR tree using these patterns so that all nodes in the tree are covered.
- Output the sequence of instructions corresponding to the tiling.

Two variants

- Greedy algorithm (top down).
- Dynamic programming (bottom up); based on cost estimates.

A simple instruction set

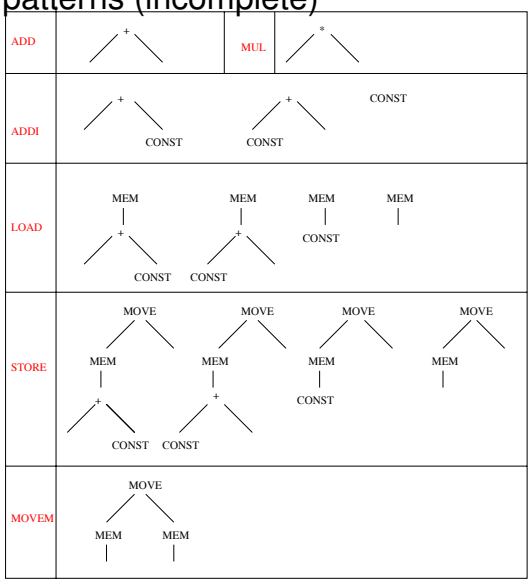
ADD	$r_i \leftarrow r_j + r_k$
MUL	$r_i \leftarrow r_j * r_k$
SUB	$r_i \leftarrow r_j - r_k$
DIV	$r_i \leftarrow r_j / r_k$
ADDI	$r_i \leftarrow r_j + c$
SUBI	$r_i \leftarrow r_j - c$
LOAD	$r_i \leftarrow M[r_j + c]$
STORE	$M[r_j + c] \leftarrow r_i$
MOVEM	$M[r_j] \leftarrow M[r_i]$

Notes

- We consider only arithmetic and memory instructions (no jumps!).
- Assume special register r_0, which is always 0.

Example done in class.

Identifying patterns (incomplete)



Peephole matching

Recall: peephole optimization

Code improvement by local simplification of the code within a small sliding window of instructions.

Can be used for instruction selection

Often one further intermediate language between IR and native code; peephole simplification done for that language.

Retargetable compilers

Instruction selection part of compiler generated from description of target instruction set (**code generator generators**).

Instruction scheduling, background

Simple-minded, old-fashioned view of processor

Fetch an instruction, decode it, fetch operands, perform operation, store result. Then fetch next operation, . . .

Modern processors

- Several instructions under execution concurrently.
- Memory system cause delays, with operations waiting for data.
- Similar problems for results from arithmetic operations, that may take several cycles.

Consequence

Important to understand data dependencies and order instructions advantageously.

Instruction selection, example

Example (from Cooper)

```
w = w * 2 * x * y * z
```

Memory op takes 3 cycles, mult 2 cycles, add one cycle.
One instruction can be issued each cycle, if data available.

Schedule 1

```
r1 <- M [fp + @w]
r1 <- r1 + r1
r2 <- M [fp + @x]
r1 <- r1 * r2
r2 <- M [fp + @y]
r1 <- r1 * r2
r2 <- M [fp + @z]
r1 <- r1 * r2
M [fp + @w] <- r1
```

Schedule 2

```
r1 <- M [fp + @w]
r2 <- M [fp + @x]
r3 <- M [fp + @y]
r1 <- r1 + r1
r1 <- r1 * r2
r2 <- M [fp + @z]
r1 <- r1 * r3
r1 <- r1 * r2
M [fp + @w] <- r1
```


Instruction scheduling

Comments

- Problem is NP-complete for realistic architectures.
- Common technique is **list scheduling**: greedy algorithm for scheduling a basic block.
  Builds graph describing data dependencies between instructions and schedules instructions from ready list of instructions with available operands.

Interaction

Despite interaction between selection, scheduling and register allocation, these are typically handled independently (and in this order).

Summing up

On optimization

We have only looked at a few of many, many techniques.

Modern optimization techniques use sophisticated algorithms and clever data structures.

Frameworks such as LLVM make it possible to get the benefits of state-of-the-art techniques in your own compiler project.