

CS412/CS413

Introduction to Compilers Tim Teitelbaum

Lecture 32: More Instruction Selection 15 Apr 07

CS 412/413 Spring 2007

Introduction to Compilers

1

Instruction Selection

1. Translate low-level IR code into DAG representation
2. Then find a good tiling of the DAG
 - disjoint set of tiles that cover the DAG
 - Maximal munch algorithm
 - Dynamic programming algorithm

CS 412/413 Spring 2007

Introduction to Compilers

2

DAG Tiling

- **Goal:** find a good covering of DAG with tiles
- **Problem:** need to know what variables are in registers
- **Assume abstract assembly:**
 - Machine with infinite number of registers
 - Temporary/local variables stored in registers
 - Parameters/heap variables: use memory accesses

CS 412/413 Spring 2007

Introduction to Compilers

3

Problems

- **Classes of registers**
 - Registers may have specific purposes
 - Example: Pentium multiply instruction
 - multiply register eax by contents of another register
 - store result in eax (low 32 bits) and edx (high 32 bits)
 - need extra instructions to move values into eax
- **Two-address machine instructions**
 - Three-address low-level code
 - Need multiple machine instructions for a single tile
- **CISC versus RISC**
 - Complex instruction sets => many possible tiles and tilings
 - Example: multiple addressing modes (CISC) versus load/store architectures (RISC)

CS 412/413 Spring 2007

Introduction to Compilers

4

Pentium ISA

- **Pentium:** two-address CISC architecture
- **Multiple addressing modes:** source operands may be
 - Immediate value: imm
 - Register: reg
 - Indirect address: [reg], [imm], [reg+imm],
 - Indexed address: [reg+reg'], [reg+imm*reg'], [reg+imm*reg'+imm']
- Destination operands = same, except immediate values

CS 412/413 Spring 2007

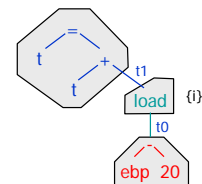
Introduction to Compilers

5

Example Tiling

- Consider: $t = t + i$
 - t = temporary variable
 - i = parameter
- Need new temporary registers between tiles (unless operand node is labeled with temporary)
- Resulting code:

```
mov %ebp, t0
sub $20, t0
mov 0(t0), t1
add t1, t
```

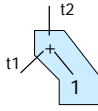


CS 412/413 Spring 2007

Introduction to Compilers

6

Tiles



mov t1, t2
add \$1, t2

- Tiles capture compiler's understanding of instruction set
- Each tile: sequence of machine instructions that match a subgraph of the DAG
- May need additional move instructions
- Tiling = cover the DAG with tiles

CS 412/413 Spring 2007

Introduction to Compilers

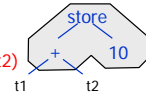
7

Some Tiles

mov t2, t1



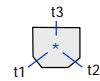
mov \$10, 0(t1,t2)



mov t2, t3
add t1, t3



mov t1, %eax
mul t2
mov %eax, t3



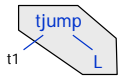
CS 412/413 Spring 2007

Introduction to Compilers

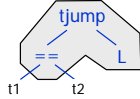
8

Conditional Branches

- How to tile a conditional jump?
- Fold comparison into tile



test t1,t1
jnz L



cmp t1,t2
je L

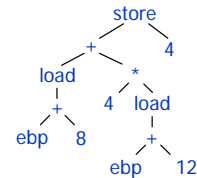
CS 412/413 Spring 2007

Introduction to Compilers

9

Maximal Munch Algorithm

- Maximal Munch = find largest tiles (greedy algorithm)
- Start from top of tree
- Find largest tile that matches top node
- Tile remaining subtrees recursively



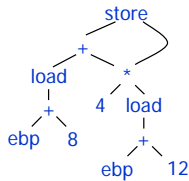
CS 412/413 Spring 2007

Introduction to Compilers

10

DAG Representation

- DAG: a node may have multiple parents
- Algorithm: same, but a node with multiple parents occurs inside a tile only if all its parents are in the tile



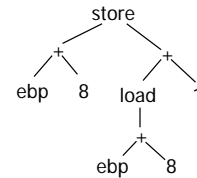
CS 412/413 Spring 2007

Introduction to Compilers

11

Example

x = x + 1;



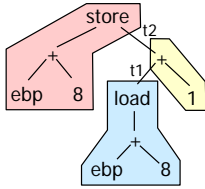
CS 412/413 Spring 2007

Introduction to Compilers

12

Example

$x = x + 1;$



```

mov 8(%ebp), t1
mov t1, t2
add $1, t2
mov t2, 8(%ebp)
    
```

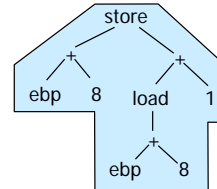
CS 412/413 Spring 2007

Introduction to Compilers

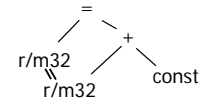
13

Alternate (CISC) Tiling

$x = x + 1;$



```
add $1, 8(%ebp)
```



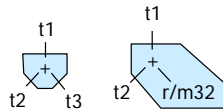
CS 412/413 Spring 2007

Introduction to Compilers

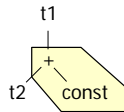
14

ADD Expression Tiles

```
mov t2, t1
add r/m32, t1
```



```
mov t2, t1
add imm32, t1
```



CS 412/413 Spring 2007

Introduction to Compilers

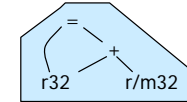
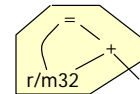
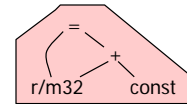
15

ADD Statement Tiles

Intel Architecture

```

add imm32, %eax
add imm32, r/m32
add imm8, r/m32
add r32, r/m32
add r/m32, r32
    
```



CS 412/413 Spring 2007

Introduction to Compilers

16

Designing Tiles

- Only add tiles that are useful to compiler
- Many instructions will be too hard to use effectively or will offer no advantage
- Need tiles for all single-node trees to guarantee that every tree can be tiled, e.g.

```
mov t2, t1
add t3, t1
```



CS 412/413 Spring 2007

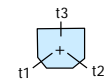
Introduction to Compilers

17

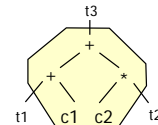
More Handy Tiles

lea instruction computes a memory address

```
lea (t1,t2), t3
```



```
lea c1(t1,t2,c2), t3
```



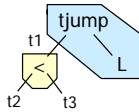
CS 412/413 Spring 2007

Introduction to Compilers

18

Matching Jump for RISC

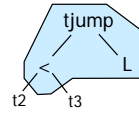
- As defined in lecture, have
 - tjump(cond, destination)
 - fjump(cond, destination)
- Our tjump/fjump translates easily to RISC ISAs that have explicit comparison result



```
MIPS
cmplt t2, t3, t1
br t1, L
```

Condition Code ISA

- Pentium: condition encoded in jump instruction
- cmp: compare operands and set flags
- jcc: conditional jump according to flags



```
cmp t1, t2
jl L
```

Annotations: "set condition codes" points to the cmp instruction, "test condition codes" points to the jl instruction.

Fixed-register instructions

mul r/m32

Multiply value in register eax
Result: low 32 bits in eax, high 32 bits in edx

jecxz L

Jump to label L if ecx is zero

add r/m32, %eax

Add to eax

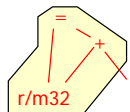
- No fixed registers in low IR except frame pointer
- Need extra move instructions

Implementation

- Maximal Munch: start from top node
- Find largest tile matching top node and all of the children nodes
- Invoke recursively on all children of tile
- Generate code for this tile
- Code for children will have been generated already in recursive calls
- How to find matching tiles?

Matching Tiles

```
abstract class LIR_Stmt {
    Assembly munch();
}
class LIR_Assign extends LIR_Stmt {
    LIR_Expr src, dst;
    Assembly munch() {
        if (src instanceof IR_Plus &&
            ((IR_Plus)src).lhs.equals(dst) &&
            is_regmem32(dst)) {
            Assembly e = ((LIR_Plus)src).rhs.munch();
            return e.append(new AddIns(dst,
                e.target()));
        }
        else if ...
    }
}
```



Tile Specifications

- Previous approach simple, efficient, but hard-codes tiles and their priorities
- Another option: explicitly create data structures representing each tile in instruction set
 - Tiling performed by a generic tree-matching and code generation procedure
 - Can generate from instruction set description: [code generator generators](#)
 - For RISC instruction sets, over-engineering

How Good Is It?

- Very rough approximation on modern pipelined architectures: execution time is number of tiles
- Maximal munch finds a locally optimal (two adjacent tiles can never be combined into one) but not necessarily globally optimum tiling (least cost of all covers)
- Metric used: tile size

CS 412/413 Spring 2007

Introduction to Compilers

25

Improving Instruction Selection

- Because it is greedy, Maximal Munch does not necessarily generate best code
 - Always selects largest tile, but not necessarily the fastest instruction
 - May pull nodes up into tiles inappropriately – it may be better to leave below (use smaller tiles above and larger, or faster tiles below)
- Better to use *dynamic programming*, an optimization technique that uses *memoization* to assure that subproblems are never solved more than once.

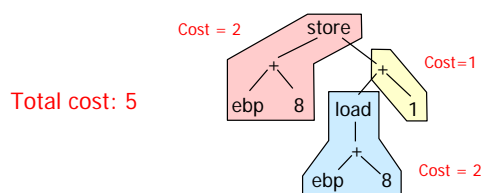
CS 412/413 Spring 2007

Introduction to Compilers

26

Timing Cost Model

- **Idea:** associate cost with each tile (say proportional to number of cycles to execute)
 - may not be a good metric on modern architectures
- Total execution time is sum of costs of all tiles



CS 412/413 Spring 2007

Introduction to Compilers

27

Finding globally optimum tiling

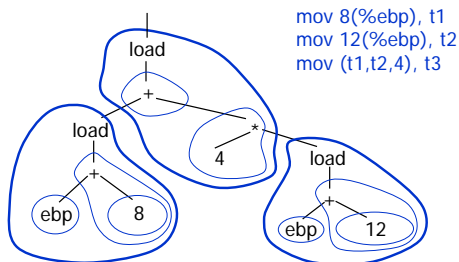
- **Goal:** find minimum total cost tiling of DAG
- **Algorithm:** for every node, find minimum total cost tiling of that node and subgraph below it
- **Lemma:** Given minimum cost tiling of all nodes in subgraph, we can find minimum cost tiling of the node by trying out all possible tiles matching the node
- **Therefore:** start from leaves, work upward to top node

CS 412/413 Spring 2007

Introduction to Compilers

28

Dynamic Programming: a[i]



CS 412/413 Spring 2007

Introduction to Compilers

29

Recursive Implementation

- Traverse DAG recursively, and for each node n , record $\langle t, c \rangle$, where
 - t is the best tile to use for subgraph rooted at n ,
 - c is the total cost of tiling the subgraph rooted at n if t is chosen.
- To compute $\langle t, c \rangle$ for node n
 - Consider every tile t' that matches rooted at n , and compute total cost $c' = \text{cost of tile } t' + \text{sum of the costs of tiling the subgraphs rooted at the leaves of } t'$ (which costs can be computed recursively and memoized)
 - Store lowest-cost tile t' and its total cost c'
- To emit code, traverse least-cost tiles recursively and emit code in postorder

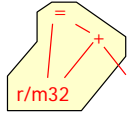
CS 412/413 Spring 2007

Introduction to Compilers

30

Memoization

```
class IR_Move extends IR_Stmt {
  IR_Expr src, dst;
  Assembly best; // initialized to null
  int optTileCost() {
    if (best != null) return best.cost();
    if (src instanceof IR_Plus &&
        ((IR_Plus)src).lhs.equals(dst) && is_regmem32(dst)) {
      int src_cost = ((IR_Plus)src).rhs.optTileCost();
      int cost = src_cost + CISC_ADD_COST;
      if (best == null || cost < best.cost())
        best = new AddIns(dst, e.target); }
    ...consider all other tiles...
    return best.cost();
  }
}
```



CS 412/413 Spring 2007

Introduction to Compilers

31

Problems with Model

- Modern processors:
 - execution time not sum of tile times
 - instruction order matters
 - Processors pipeline instructions and execute different pieces of instructions in parallel
 - bad ordering (e.g. too many memory operations in sequence) stalls processor pipeline
 - processor can execute some instructions in parallel (super-scalar)
 - cost is merely an approximation
 - instruction scheduling needed

CS 412/413 Spring 2007

Introduction to Compilers

32

Summary

- Can specify code generation process as a set of tiles that relate low IR trees (DAGs) to instruction sequences
- Instructions using fixed registers problematic but can be handled using extra temporaries
- Maximal Munch algorithm implemented simply as recursive traversal
- Dynamic programming algorithm generates better code, can be implemented recursively using memoization
- Real optimization will also require instruction scheduling

CS 412/413 Spring 2007

Introduction to Compilers

33