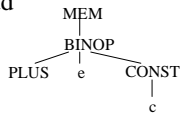


Instruction Selection

- A list of simplified IR trees -> need to map these to assembly instructions
- tree node \Leftrightarrow one operation
 - BINOP(...)
- machine instruction \Leftrightarrow multiple operations
 - e.g., load involves fetch + add

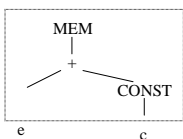


Instruction trees

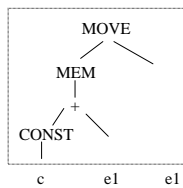
- Represent instructions as tree fragments – *tree patterns - tiles*
- Tile the IR trees with instruction tiles so that
 - no IR tree nodes are uncovered
 - no tiles overlap
- The tiling corresponds to selected instructions

Instruction tiles

- tile root node
 - a value in a register
- e and c
 - unresolved subtrees, leaves
- other instructions
 - no result in register
 - side effect only



load/lw tile



store/sw tile

Optimal & Optimum Tiling

- An instruction may have alternate patterns
- An IR can be tiled in different ways
- A tiling is *optimum* if the set of tiles have lowest cost (best)
 - cost – e.g., number of tiles used, sum of cycles for instructions selected
- A tiling is *optimal* if no two adjacent tiles can be combined into a single tile of lower cost (locally best)

optimum \Rightarrow optimal

is it really optimum? what about other optimizations – constant folding, code movement...

- How do we find optimum/optimal tiling?
 - maximal munch
 - dynamic programming
 - tree grammars

Maximal Munch

- choose “largest” tile that matches at the root of the IR tree – munch
 - this is instruction I
 - you still have subtrees T1, T2...
- Recursively apply maxmunch for T1...
- Generate instruction I with source/destination registers produced from subtrees T1...

- good to have large initial exp trees – can take large munches
- but... maybe an alternate smaller munch at the beginning can leave subtrees that can be tiled at lower cost/better...
 - greedy algorithm, uses local info only...

Dynamic Programming

- find optimum for entire problems based on optimums for each subproblem
 - bottom-up
- assume – each instruction has a cost value c
- at node n, each possible tiling leaves zero or more residual subtrees (leaves)
 - costs of leaves are already computed
 - for a specific tiling t, the cost of n will be the sum of costs of t and subtrees
- choose the tiling t with minimal cost
- When you get to root node
 - emit instructions for each leaf
 - emit instruction for selected tile for root

Tree Grammars

- What if we have to consider additional conditions when selecting a tile
 - e.g. different registers for address and data
 - ok – can label nodes in tiles with conditions
 - can return result in a , d , or none (statement s)
 - can represent them as grammars
 - a , d , s are nonterminals
 - grammar ambiguous – but ok, we want to consider different possibilities to find minimum cost

- dynamic programming algorithm still works
- at each node find minimum cost to produce a and minimum cost to produce d , and any other nonterminal...
 - e.g. different sized immediates
- programming can be hairy – tools exists
 - BURG – accepts tree grammars and produces a tree parser
 - ML-BURG overkill for Tiger parser

- For Tiger compiler:
 - Maximal Munch in ML
 - ML great for pattern patching
 - in C we'll have to encode fast matching algorithm
 - consider type of each node, existing patterns for that type, then children...

Implementing Maximal Munch

```
fun munchStm(MOVE(MEM(BINOP(PLUS,e1,CONST i)),e2)
  =(munchExp e1; munchExp e2; emit "STORE")
| munchStm(MOVE(MEM(BINOP(PLUS,CONST i,e1)),e2)
  =(munchExp e1; munchExp e2; emit "STORE")
| munchStm(MOVE(MEM e1,FETCH(MEM e2)))
  =(munchExp e1; munchExp e2; emit "STORE")
| ...

fun munchExp(BINOP(PLUS,e1,CONST i))
  =(munchExp e1; emit "ADDI")
| munchExp(BINOP(PLUS,CONST i,e1))
  =(munchExp e1; emit "ADDI")
| munchExp(BINOP(PLUS,e1,e2))
  =(munchExp e1; munchExp e2; emit "ADD")
| ...
```

- Register allocation – not yet
- use virtual registers
 - no need to tie reg. alloc to machine specific instruction selection
 - aside from number/kind of registers, allocation is machine-independent, so want to be able to reuse
 - in tree – register == root of tile
 - before IS, no information which IR nodes will registers
 - some may be internal to an inst. tile
 - we'll do reg. alloc after instruction selection

Representing Instructions

```
datatype instr
  = OPER of
    {assem: string,
     dst: temp list,
     src: temp list,
     jump: label list option}
  | LABEL of {assem: string, lab: label}
  | MOVE of
    {assem: string,
     dst: temp,
     src: temp}
```

Sample instructions

```
sw $ta 4($tb)
  OPER{assem = "sw `s0, 4(`s1)",
       dst = [],
       src = [tempa,tempb],
       jump = NONE}

L0 :
  LABEL{assem = "L0 :", lab = labelL0}

move $ta, $tb
  MOVE{assem = "move `d0, `s0",
       dst = tempa,
       src = tempb}
```

```
structure A = Assem

fun munchStm(T.SEQ(a,b) = (munchStm a; munchStm b)
  | munchStm(T.MOVE(T.MEM(T.BINOP(T.PLUS,e1,T.CONST i)),e2))
  = emit(A.OPER{assem="sw `s0, " ^ (intToString i) ^
    "`s1)",
    src=[munchExp e2, munchExp e1],
    dst=[],
    jump=NONE}) (if i within no. of bits)
  | munchStm(T.MOVE(T.MEM(T.BINOP(T.PLUS,T.CONST i,e1)),e2))
  = emit(A.OPER{assem="sw `s0, " ^ (intToString i) ^ ...
  ...
  | munchStm(T.LABEL lab)
  = emit(A.LABEL{assem=Symbol.name lab ^ ":", lab=lab})
```

```

and munchExp(T.BINOP(T.PLUS,T.CONST i, e2)) =
  result(fn r=emit(A.OPER{
    assem="addiu `d0, `s0, " ^ (intToString i),
    src=[munchExp e2],
    dst=[r],
    jump=NONE}) (but if i >>, then multiple inst)
...
| munchExp(T.MEM(T.BINOP(T.PLUS,e1,T.CONST i))) =
  result(fn r=emit(A.OPER{
    assem="lw `d0, " ^ (intToString i) ^ "`s0)",
    src=[munchExp e1],
    dst=[r],
    jump=NONE})

```

- `munchExp: Tree.exp -> Temp.temp`
- `munchStm: Tree.stm -> unit`

in codegen

- `emit -> instruction list`
- `results -> newtemp`

in assem

- `format(m)(i) -> string`
 - `m` - function that translates temp registers to appropriate strings
 - `i` - the instruction

```

...
| munchStm(A.JUMP(T.NAME lab, lablist)) =
  emit(A.OPER{assem = "j `j0",
    src = [],
    dst = [],
    jump = SOME lablist})
| munchStm(A.CJUMP(relop,e1,e2,t,f) =
  case relop of
  T.EQ => (emit(A.OPER{assem = "beq `s0, `s1, `j0",
    src = [munchExp e1, munchExp e2],
    dst = [],
    jump = SOME([t, f])})
...

```

what if we have instruction for relop – need a newtemp for intermediate results

`EXP(CALL(f,arglist))` or `MOVE(TEMP t, CALL(f, arglist))`

```

| munchStm(T.EXP(T.CALL(e1,arglist)) =
  emit{assem="jalr `s0",
    src=[munchExp e1:: munchArgs... (or something)
    dst=calldefs,
    jump=NONE})

```

`munchArgs` – generates code for movement of argument to correct position (reg, memory)

- results in list of source registers for the CALL instruction
- list in not really an arg, but needed for liveness analysis

`calldefs` – all registers ‘trashed’ by the call, the caller-save registers, `$ra`, `$rv...`

- To construct the register list for arguments, and calldefs, need to distinguish between the classes of registers:
 - special – RV, FP, ...
 - argregs – 4, 5, 6, 7
 - calleesaves – 1, 3, 8, 9...
 - callersaves – 16, 17, ...
- have a register map to translate registers to “\$...” strings – use register int 0 - 31 as key

Complex instruction sets

- RISC:
 - 32 regs, of one class,
 - arithm ops only between regs,
 - three-address inst, M[reg+const] addressing mode,
 - inst are 32bits long, and produce one result

- CISC:
 - fewer registers: ok – assume infinite set of temps, and reg. allocator will figure it out.
 - classes of registers: and operations which operate only on specific class: - have to do register movement, and hope that reg. alloc. will do a good job!


```
t1 <- t2 * t3 on Pentium, * needs eax
move eax, t2      eax <- t2
mul t3           eax <- eax*t3, edx <- garbage
mov t1, eax      t1 <- eax
```

- two-address instructions:


```
t1 <- t2 + t3
mov t1, t2
add t1, t3
```
- arithm ops can address memory


```
mov eax, [ebp, -8]
add eax, ecx          add [ebp, -8], ecx
mov [ebp, -8], eax
```

 - same thing, same cyclecount, but no trashing of eax, and shorter encoding
- multiple addressing modes
 - introduces lots of tiles
- var. length insts
 - no problem – IS doesn't care
- insts. with side effects
 - e.g., autoincrement – multiple results $r2 <- M[r1]; r1 <- r1 + 4$