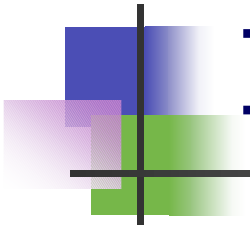
Instruction selection

Niels Janssen

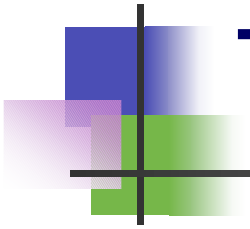
njanssen@cs.uu.nl

HPC 2001



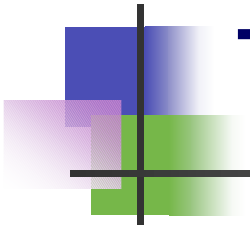
Introduction

- Tiger compiler
  - Tiger -> IR -> MIPS assembly
  - Today; IR -> MIPS assembly
    - Instruction Selection
    - Run MIPS assembly-code with SPIM
  - No FunDec's and VarDec's
- First a basic compiler
  - No optimalizations (Loop reduction etc)



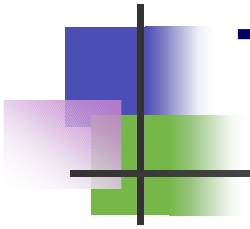
Tiling (1)

- Basic idea of instruction selection
- Find tiles in IR-tree
  - Patterns corresponding with instructions
  - Non-overlapping
- Algorithms
  - Maximal Munch
  - Dynamic Programming
- Optimal tiling vs the optimum



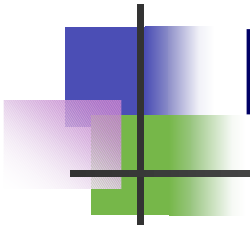
Tiling (2)

- Optimal tiling
  - Tiles can’t be split in smaller tiles
  - More than one optimal tiling possible
  - Maximal Munch
- Optimum
  - Tiling with lowest cost
  - Dynamic Programming



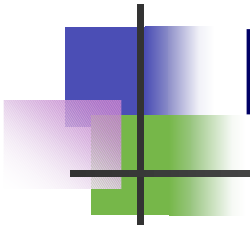
Tiling (3)

- A tile represents an instruction
  - RISC architecture
    - Small tiles
    - Just a few nodes
  - CISC architecture
    - Complex tiles
    - Could contain a bigger subtree
- Programmer designs tiles



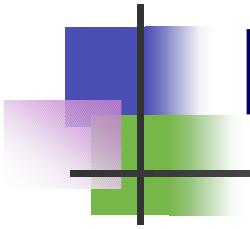
Maximal Munch (1)

- Algorithm finds an optimal tiling
- Topdown traversal
  - Starting at root; find largest pattern-match
  - Do the same for each subtree
- Largest pattern-match
  - Size depends on number of nodes per tile
  - More options mean deterministic behavior
- Greedy algorithm; not always optimum tiling



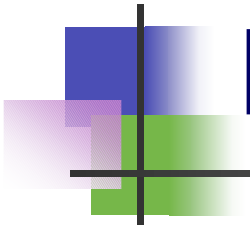
Maximal Munch (2)

- Code emission
  - Starts when entire tree is tiled
  - Emit code in reverse order
    - Invoke code emission when returning from recursion



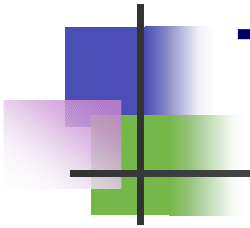
Dynamic Programming (1)

- Algorithm calculates an optimum tiling
- Bottomup traversal
  - First; calculate cost of (grand)children
  - Then; find cheapest tile for this node
- Every tile has zero or more leaves
  - Subtrees are connected to leave(s) of a tile
  - Cost of subtree rooted under current node
    - Cost of current pattern-match plus all subtrees



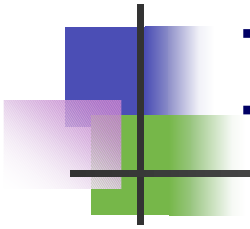
Dynamic Programming (2)

- Once cost of root node is calculated
  - Cost of entire tree calculated
  - Start of code-emission phase
- Code-emission
  - Reverse order
  - Emission; emit for all leaves then tile
  - Leaves are tiles rooted under current tile



The algorithms

- Maximal Munch
  - Easy algorithm to implement
  - Tiger compiler
- Dynamic programming
  - Harder to implement
  - BURG; Bottomup rewriting system
  - Lecture also covers BURG later on



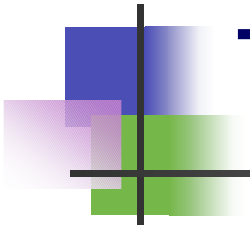
Implementation (1)

- Maximal Munch using Stratego
  - Traversal
    - MaximalMunch = topdown(repeat(Select))
  - Rules
    - Select: rewrite matching pattern of IR-tree to certain ASM-tree
  - ‘Code emission’ happens directly
    - Matching patterns are rewritten directly
    - No real code; an ASM-tree

Implementation (2)

- What happens to ASM-tree?
  - Pretty-print ASM-tree
  - Use overlays to emit MIPS Assembly-code

```
module MIPS
imports ASM

overlays

  add(rd, rs, rt) =
    OPER(["add ", D(1), ",", S(1), ",", S(2)], [rs, rt], [rd], [])
```



Tiger -> IR

- Tiger
  - Only knows integers and strings
  - Each element is 4 bytes long; one word
  - Can represent
    - An integer value
    - A pointer to some value
  - String is an allocation of (length * word)

IR expressions

```
Const(i)          : Integer constant i
MEM(e)            : Returns contents at address calculated with exp e
NAME(n)           : Address that corresponds to label n
TEMP(t)           : If t is temporary register; return its value
BINOP(op,e1,e2)   : Calculate e1, then e2; then op values e1 and e2
                    Abbreviated to op(e1,e2); e.g. +(e1,e2)
CALL(f,[e1..en]) : Evaluate e1 to en; then call function f with these
                    values as parameter(s)
ESEQ(s,e)         : Execute statement s then evaluate and return e
```

IR statements

```
MOVE(TEMP(t),e)    : store value of expression e in register t
MOVE(MEM(e1),e2)   : evaluate e1 (address); then store value of e2 in e1
EXP(e)             : evaluate e; discard result
JUMP(e,[l1..ln])   : evaluate e to get address (label) then jump to it
CJUMP(o,e1,e2,t,f): evaluate e1, then e2; If e1 op e2 results in true
                      then jump to label t else jump to label f
SEQ(s1,s2)         : Perform statement s1 and then s2
LABEL(n)           : Define name n to be address of this statement
```

Tiger -> IR (1)

```
// Tiger code; retrieve the i-th element of array a

a[i]

// TAS

Subscript(Var("a"),Var("i"))

// IR
// A: address of array a; e.g. BINOP(PLUS,TEMP(fp),CONST(2))
// I: value of i; e.g. MEM(BINOP(TEMP(fp),CONST(35)))

MEM(BINOP(PLUS,A,BINOP(MUL,I,4)))

// Appel's notation, which we use from now on

MEM(+(A,*(I,4)))
```

Tiger -> IR (2)

```
// Tiger code; retrieve x from record r

r.x

// TAS

FieldVar(Var("r"),"x")

// IR

MEM(+(R,CONST(X*4)))
```

Some tiles

	Tile (IR)	MIPS (ASM)		
0.	CONST(c)		li	‘d0, c
1.	+(e0,e1)		add	‘d0, ‘s0, ‘s1
2.	+(e0,c)		add	‘d0, ‘s0, c
3.	*(e0,e1)		mult	‘d0, ‘s0, ‘s1
4.	*(e0,CONST(2^k))		sll	‘d0, ‘s0, k
5.	MEM(e0)		lw	‘d0, (‘s0)
6.	MEM(+(e0,CONST(c)))		lw	‘d0, c(‘s0)
7.	MOVE(MEM(e0),e1)		sw	‘s1, (‘s0)
8.	MOVE(MEM(+(e0,CONST(c))),e1)		sw	‘s1, c(‘so)
9.	JUMP(NAME(a_0))		b	a_0
A.	JUMP(e0)		jr	‘s0
B.	LABEL(a_0)	a_0:	nop	

d0: destination address
s0: source address with value of e0
s1: source address with value of e1
c : constant/immediate

Tiger -> IR -> ASM (1)

```
// Tiger code; Assign x to the i-th element of array a

a[i] := x

// TAS

Assign(Subscript(Var("a"),Var("i")),Var("x"))

// IR
// i : Temporary register
// a : stored at offset 20 of the framepointer
// x : stored at offset 10 of the framepointer

MOVE(
        MEM(+(MEM(+(TEMP(fp),CONST(20))),*(TEMP(i),CONST(4)))),
        MEM(+(TEMP(fp),CONST(10)))
)
```

Tiger -> IR -> ASM (2)

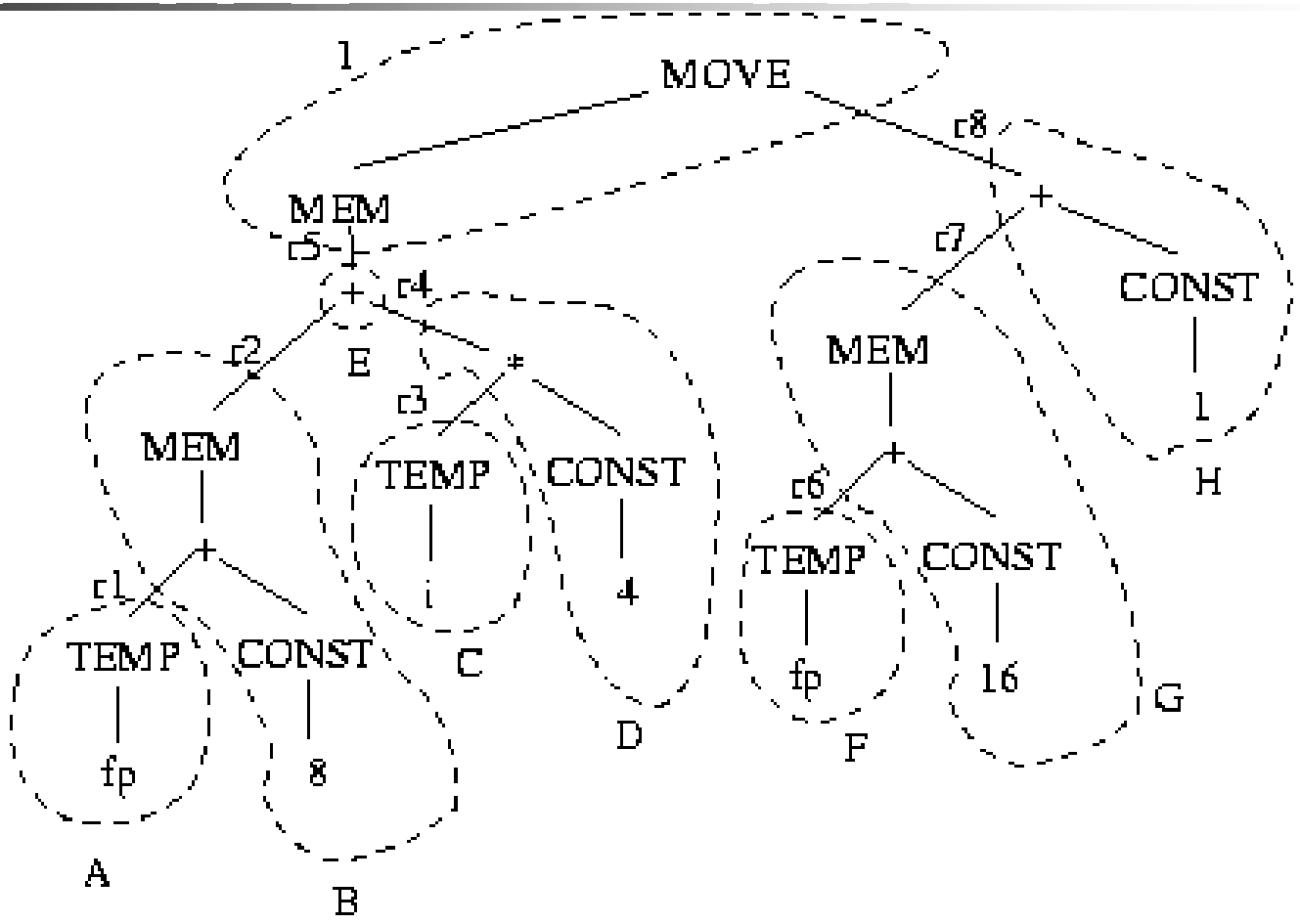
```
// IR
// i : Temporary register
// a : stored at offset 20 of the framepointer
// x : stored at offset 10 of the framepointer

MOVE( MEM(+(MEM(+(TEMP(fp),CONST(20))),*(TEMP(i),CONST(4)))),
      MEM(+(TEMP(fp),CONST(10))) )

// Possible tilings:

6. lw   r1, 20($fp)          2. add  r1, $fp, 20
4. sll  r2, r1, 2            5. lw   r1, (r1)
1. add  r1, r1, r2           4. sll  r2, r1, 2
6. lw   r2, 10($fp)          1. add  r1, r1, r2
7. sw   r2, (r1)             2. add  r2, $fp, x
                             5. lw   r2, (r2)
                             7. sw   r2, (r1)
```

Tiger -> IR -> ASM (3)

- Using the Maximal Munch algorithm

```
// Tiger code; Assign value of (x+1) to the i-th element of array a

a[i] := (x+1)

// TAS

Assign(Subscript(Var("a"),Var("i")),BinOp(PLUS,Var("x"),Int("1")))

// IR
// i : Temporary register
// a : stored at offset 8 of the framepointer
// x : stored at offset 16 of the framepointer

MOVE(
        MEM(+(MEM(+(TEMP(fp),CONST(8))),*(TEMP(i),CONST(4)))),
        MEM(+(TEMP(fp),CONST(16))),CONST(1)))
)
```

Tiger -> IR -> ASM (4)



Tiger -> IR -> ASM (5)

- Result of tiling using Maximal Munch
  - [I,E,H,B,D,G,A,C,F]
  - Order determined by traversal (topdown)
  - Use unique registers between tiles
    - [r1,r2,r3,r4,r5,r6,r7,r8]
- Code emission
  - [A,B,C,D,E,F,G,H,I]
  - Reverse order (returning from recursion)

Tiger -> IR -> ASM (6)

- Code emitted after Maximal Munch

```
// Code emission (MIPS Assembly) of selected tiles in IR-tree

A (5)     lw   r1, $fp
B (6)     lw   r2, 8(r1)
C (5)     lw   r3, $i
D (4)     sll  r4, r3, 2
E (1)     add  r5, r2, r4
F (5)     lw   r6, $fp
G (6)     lw   r7, 16(r6)
H (2)     add  r8, r7, 1
I (7)     sw   r8, (r5)
```

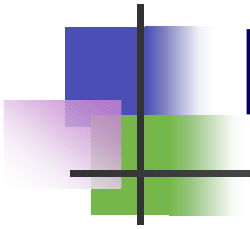
Tiger -> IR -> ASM (7)

- Function calls

```
// Tiger code; call function f with arguments e1 and e2

f(e0,e1)

// TAS

Call(Var("f"),[e0,e1])

// IR

MOVE(CALL(NAME(f),[e0,e1]),TEMP(a_0))

// or when function f only produces a side-effect

EXP(CALL(NAME(f),[e0,e1]))
```

Tiger -> IR -> ASM (8)

```
// IR

MOVE(CALL(NAME(f),[e0,e1]),TEMP(a_0))

// or when function f only produces a side-effect

EXP(CALL(NAME(f),[e0,e1]))

// ASM for first option

sw      $a0, (s0)          # assign result of e0 (register s0) to $a0
sw      $a1, (s1)          # assign result of e1 (register s0) to $a1
jal     F                  # call procedure f
lw      a_0, ($v0)         # assign returnvalue $v1 to temporary $t0
```



BURG (1)

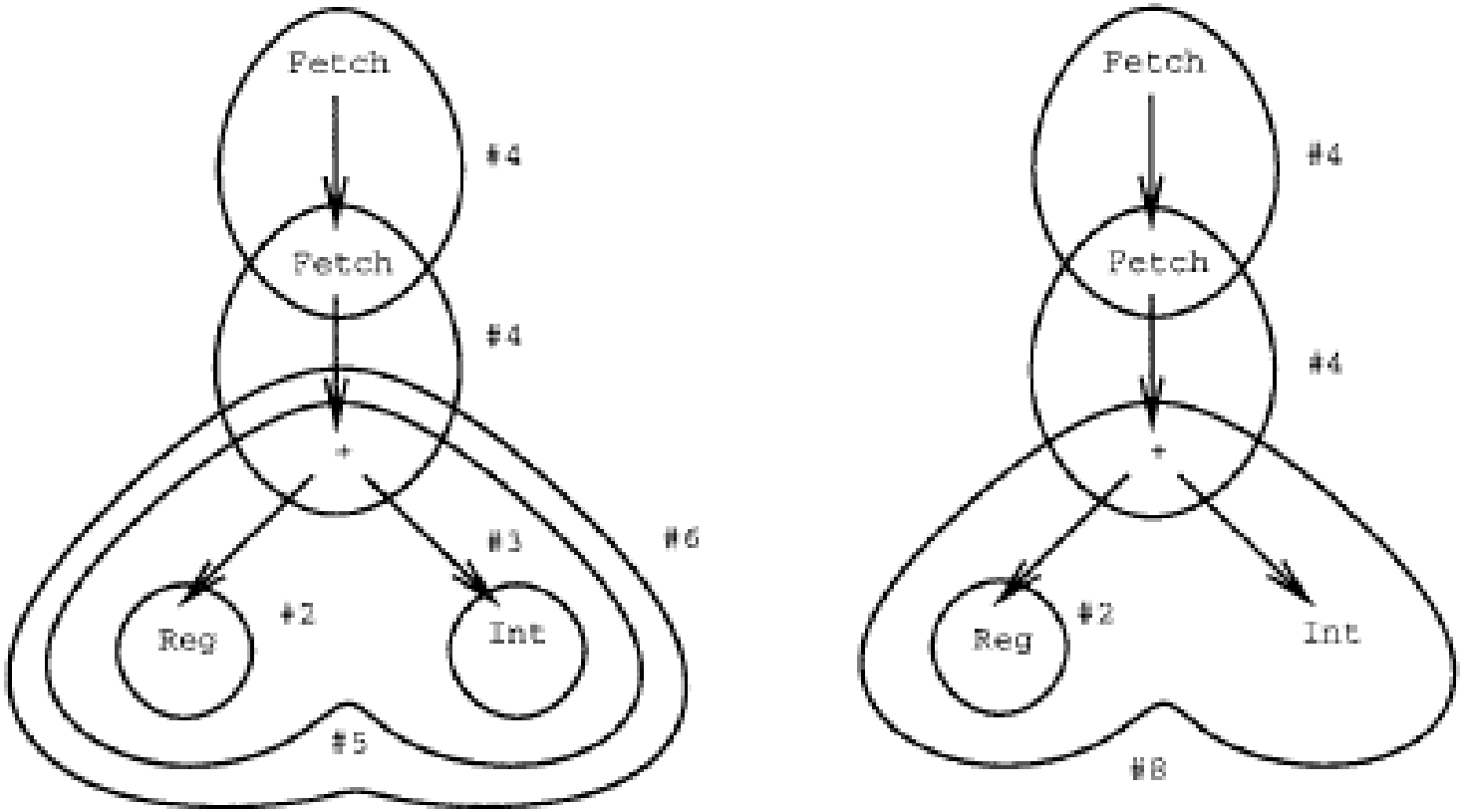
- BottomUp Rewrite Generator
- Code-generator generator system
  - Todd Proebsting (Microsoft Research) e.a.
- Uses IR-trees
- Dynamic programming
  - Goal of system; find lowest cost tiling
- IR -> ASM using tree grammar
  - ‘n -> t (c)’

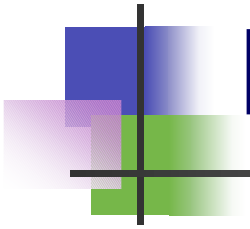
BURG (2)

- Tree grammar example

```
// Code emission (MIPS Assembly) of selected tiles in IR-tree

[1] goal -> reg            (0)  [5] reg  -> Plus(reg, reg) (2)
[2] reg  -> Reg            (0)  [6] addr -> reg            (0)
[3] reg  -> Int            (1)  [7] addr -> Int            (0)
[4] reg  -> Fetch(addr)    (2)  [8] addr -> Plus(reg, Int) (0)

// Term;

Fetch(Fetch(Plus(Reg,Int)))

// Two tilings possible;

1st. 4(4(6(5(2,3))))
2nd. 4(4(8(2)))
```

BURG (3)





BURG (4)

- Based on BURS; Bottomup Rewrite System
  - Bottomup traversal
- Tiling algorithm
  - Compute and associate cheapest tiles for each node
    - By matching right-hand side of productions
    - Consider that subtree may be tiled already
- Instructions are selected afterwards
  - topdown