Optimization During Tree-Parsing Code Selection

M. Anton Ertl

anton@mips.complang.tuwien.ac.at

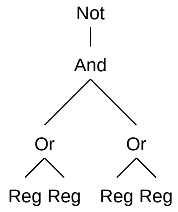
http://www.complang.tuwien.ac.at/anton/

Institut für Computersprachen

Technische Universität Wien

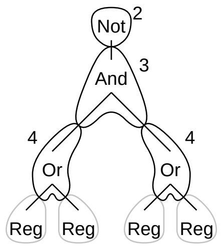
Intermediate representation tree

`~((a|b)&(c|d))`



MIPS code selection grammar

```
# Production             Cost  Action
1 reg: Reg               # 0 #
2 reg: Not(reg)          # 1 # gen("not ...
3 reg: And(reg,reg)      # 1 # gen("and ...
4 reg: Or(reg,reg)       # 1 # gen("or ...
5 reg: Not(Or(reg,reg))  # 1 # gen("nor ...
```




```
or  $1,$4,$5
or  $2,$6,$7
and $1,$1,$2
not $1,$1
```

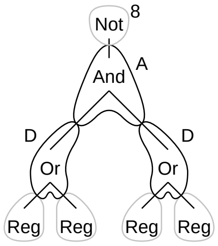
Optimizations expressed in tree grammars

- Introduce new nonterminals
- They correspond to new data representations (e.g., after the application of operations)

Optimizing `Not`

`notreg` represents a register containing the ones-complement of the ordinary value

```
# Production                 Cost  Action
6 notreg: Not(reg)           # 0 #
7 notreg: reg                # 1 # gen("not ...
8 reg:    Not(notreg)        # 0 #
9 reg:    notreg             # 1 # gen("not ...
A notreg: And(notreg,notreg) # 1 # gen("or ...
B notreg: Or(notreg,notreg)  # 1 # gen("and ...
C reg:    And(notreg,notreg) # 1 # gen("nor ...
D notreg: Or(reg,reg)        # 1 # gen("nor ...
```




```
nor $1,$4,$5
nor $2,$6,$7
or  $1,$1,$2
```

Discussion

+ Optimal for the machine (for trees)

+ No compile-time overhead

− Finite number of nonterminals/representations + data in actions

− Optimality limited to trees

− Limited to single-entry regions

− Large grammars

Optimizing unary operators

- unary −, +1, ×2

Advanced constant folding

`rpc` represents register + constant (val)

```
rpc: reg              # 0 # val=0;
rpc: Plus(rpc,const)  # 0 # val=kid0.val+kid1.val;
rpc: Plus(const,rpc)  # 0 # val=kid0.val+kid1.val;
rpc: Plus(rpc,rpc)    # 1 # val=kid0.val+kid1.val; gen("addu ...
reg: rpc              # 1 # gen("addiu ...
```

- `3+x+4` produces `addiu $1,$2,7`

- Variants:
  any associative and commutative operation
  multiply and add

Flag optimization

- Canonical flags: 0/1

- Conditionals: zero/non-zero

`zflag` represents a canonical flag as zero/non-zero flag

```
root:  CBranch(zflag)    # 1 # gen("beq ...
zflag: Differs(reg,Zero) # 0 #
zflag: Or(zflag,zflag)   # 1 # gen("or ...
reg:   zflag             # 1 # gen("sltiu ...
```

- `if ((a!=0)|(b!=0))` produces

  ```
  or  $1,$4,$5
  beq $1,$2,...
  ```

- Similar:
  sign bit vs. canonical flag for < 0
  tagged vs. untagged
  sign/zero-extended vs. garbage-extended

Further work: Factoring

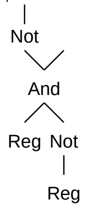
- Combining optimizations → rule explosion
- Similar grammar extensions on all machines
- Equivalence rules specific to intermediate representation, not machine-specific
- Ideal: equations + simple machine grammars

```
Not(Not(a))=a
Not(And(Not(a),Not(b)))=Or(a,b)
...
```

- Realistic: Explicitly specify how to generate new nonterminals (macro-like).

Further work: Beyond trees

Problem: shared nodes, nodes in other basic blocks




```
not $1, $4
or  $1, $1, $5 #for left parent
not $2, $5
and $2, $4, $2 #for right parent
```

- Ignore problem; or
- Force canonical nonterminal; or
- Force cheapest or canonical nonterminal

Conclusion

- Additional nonterminals correspond to a finite number of data representations
- Optimizes for the machine
- No compile-time overhead