# CS412/CS413

# Introduction to Compilers

Tim Teitelbaum

## Lecture 32: More Instruction Selection

11 Apr 08

# Near Optimal Instruction Selection on DAGs

By David Ryan Koes and Set Copen Goldstein

Carnegie-Mellon University

CGO ’08 Boston, MA (April 7, 2008)

Please read paper (found on Koes web page)

# Optimal Tiling for Trees

- Pass 1. Ignore common subexpressions and find optimal covering for fully duplicated DAG

# **procedure** Select

fixedNodes := { }

BottomUpDP()

TopDownSelect()

ImproveCSEDecisions()

BottomUpDP()

TopDownSelect()

# **procedure** BottomUpDP

```
for n ∈ reverseTopologicalSort(DAG) do
    bestChoiceForNode[n].cost := ∞
    for t_n ∈ matchingTiles(n) do
        if not hasInteriorFixedNode(t_n, fixedNodes) then
            val := cost(t_n) +
                    Σ_{n' ∈ edgeNodes(tn)} bestChoiceForNode[n'].cost
            if val < bestChoiceForNode[n].cost then
                bestChoiceForNode[n]:cost := val
                bestChoiceForNode[n].tile := t_n
```

# **procedure** TopDownSelect

```
matchedTiles.clear()
coveringTiles.clear()
q.push(roots(DAG))
while not q.empty() do
    n := q.pop()
    bestTile := bestChoiceForNode[n].tile
    matchedTiles.add(bestTile)
    for every node n_t covered by bestTile do
        coveringTiles[n_t].add(bestTile)
    for n' ∈ edgeNodes(bestTile) do
        q.push(n')
```

# Optimal Tiling for Trees

- Pass 1. Ignore common sub-expressions and find optimal covering for fully duplicated DAG
- fixedNodes := set of shared nodes not allowed in interiors of tiles, i.e., nodes required to match roots of tiles and therefore represent common sub-expressions for which instruction selection can be done in a decomposed manner.
- Pass 2. Find optimal covering for decomposed DAG defined by fixedNodes

# **procedure** Select

```
fixedNodes := { }
BottomUpDP()
TopDownSelect()
ImproveCSEDecisions()
BottomUpDP()
TopDownSelect()
```

# **procedure** BottomUpDP [revised]

**for** n ∈ reverseTopologicalSort(DAG) **do**

  bestChoiceForNode[n].cost := ∞

  **for** $t_n$ ∈ matchingTiles(n) **do**

    **if not** hasInteriorFixedNode($t_n$, fixedNodes) **then**

      val := cost($t_n$) +

        $\Sigma_{n' \in \text{edgeNodes}(tn)}$ bestChoiceForNode[n'].cost

      **if** val < bestChoiceForNode[n].cost **then**

        bestChoiceForNode[n]:cost := val

        bestChoiceForNode[n].tile := $t_n$

# **procedure** TopDownSelect [revised]

```
matchedTiles.clear()
coveringTiles.clear()
q.push(roots(DAG))
while not q.empty() do
    n := q.pop()
    bestTile := bestChoiceForNode[n].tile
    matchedTiles.add(bestTile)
    for every node n_t covered by bestTile do
        coveringTiles[n_t].add(bestTile)
    for n' ∈ edgeNodes(bestTile) do
        q.push(n')
```

# **procedure** ImproveCSEDecision

```
for n ∈ sharedNodes(DAG) do
   if coveringTiles[n].size() > 1 then
      overlapCost :=  getOverlapCost(n, coveringTiles)
      cseCost := bestChoiceForNode[n].cost
      for t_n ∈ coveringTiles[n] do
         cseCost := cseCost + getTileCutCost(t_n, n)
      if cseCost < overlapCost then
         fixedNodes.add(n)
```

/* I.e., if the cost of the CSE + the cost of cutting the overlapping tiles < cost of the overlapping computation, decompose DAG at CSE. */

# **function** GetOverlapCost(n)

```
cost := 0; seen  := { }
for t ∈ coveringTiles[n] do q.push(t); seen.add(t)
while not q.empty() do
    t := q.pop()
    cost := cost + cost(t)
    for n’ ∈ edgeNodes(t) do
        if n’ is reachable from n then
            t’ := bestChoiceForNode[t].tile
            if coveringTiles[n’].size() = 1 then
                cost := cost + bestChoiceForNode[n’].cost
            else if t’ ∉ seen then
                seen.add(t’)
                q.push(t’)
                return cost
```

# **function** GetTileCutCost(t,n)

```
bestCost := ∞
r := root(t)
for t' ∈ matchingTiles(r) do
    if n ∈ edgeNodes(t') then
        cost := cost(t')
        for n' ∈ edgeNodes(t') ^ n' ≠ n do
            cost := cost + bestChoiceForNode[n'].cost
        if cost < bestCost then
            bestCost  := cost
for n' ∈ edgeNodes(t) do  //Subtract edge costs of original tile
    if path r → n' ∈ t does not contain n then
        bestCost := bestCost - bestChoiceForNode[n'].cost
return bestCost
```

# 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

# 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