VU2 185.324

Compilation Techniques for VLIW Architectures

Dietmar Ebner `ebner@complang.tuwien.ac.at`
Florian Brandner `brandner@complang.tuwien.ac.at`

`http://complang.tuwien.ac.at/cd/vliw`

Last Lectures (1)

- Traditional Scalar Optimizations
  - Common subexpression elimination
  - Copy propagation
  - Copy elimination
  - Dead-code elimination
  - Strength reduction
- Function Inlining

Last Lectures (2)

- Dependencies
  - Control Dependencies
  - Data Dependencies
    - Read-after-write (true dependence)
    - Write-after-Read (anti dependence)
    - Write-after-write (output dependence)
- Alias Analysis
  - Flow-sensitive vs. flow-insensitive
  - Inter- / Intraprocedural

Last Lectures (3)

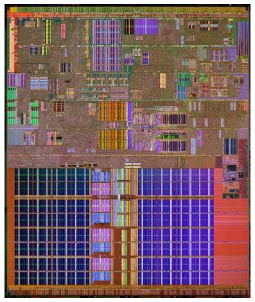
- Natural loops
- Dominance relation
- Backedges
- Reducible control flow graphs
- Loop carried dependencies

Last Lectures (4)

- Loop Transformations
  - Scalar expansion
  - Loop distribution
  - Loop interchange
  - Loop fusion
  - Loop peeling
  - Loop blocking
  - ...

In Today's Lecture

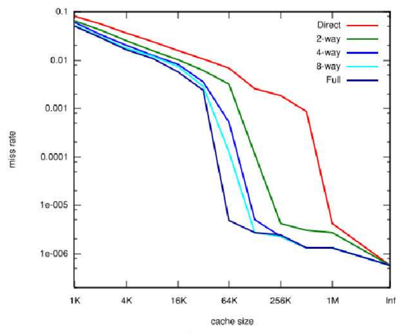
- Code Layout
  - Block / function placement
- Instruction Selection
  - Translates a compilers IR to machine code
  - BURS systems
  - DAG based approaches



Code Layout Techniques

- Embedded processors usually adopt simple cache structures (direct mapped / low associativity)
- Instruction cache fetch path is usually among the critical drivers of overall clock cycle
- Sources of i-cache misses
  - capacity misses
  - conflict misses ⇦ **Code Layout Techniques**
  - compulsory (cold) misses

Excursion: Cache Organization



Hill, Cantin: Cache Performance for SPEC CPU2000 Benchmarks

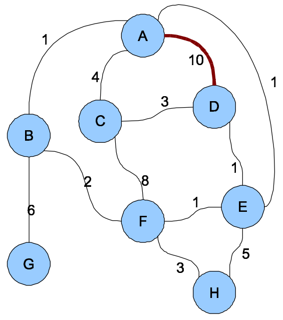
Placement Techniques - Motivation

- Default Code Layout is often bad
  - Instructions and Procedures are usually placed according to source order
- Rearraging can lead to a reduced miss rate
- "***Closest is best***" strategy
  - procedures calling each other frequently wind up close to each other
  - reduces working set

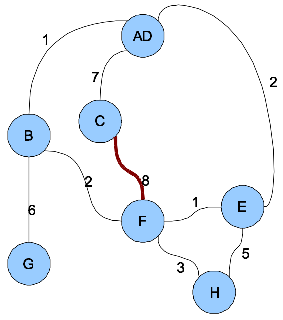
Pettis and Hansen: Procedure Positioning

- Construct a *weighted* call graph
  - Nodes correspond to procedures
  - Edge label denote the total number of **dynamic** calls
- Iteratively merge both nodes incident to edges with highest dynamic weight
- Keep a "**chain**" within those merged nodes corresponding to the link order

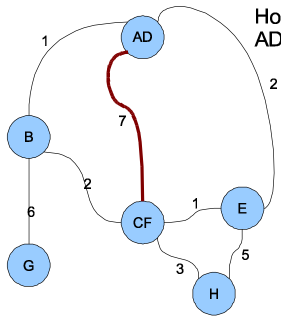
Procedure Positioning



Procedure Positioning



Procedure Positioning



How to merge chains for AD and CF?

closest is best strategy

Pick either DACF or FCAD

(C and A should be adjacent)

Pettis and Hansen: Basic Block Ordering

- Defines the order of blocks within a procedure
- Weighted control flow graph
- Layout blocks such that the “normal” flow of control is in a straight line
- Two step approach:
  - Identify chains of blocks
    - top-down
    - bottom-up
  - Define a precedence relation among those chains

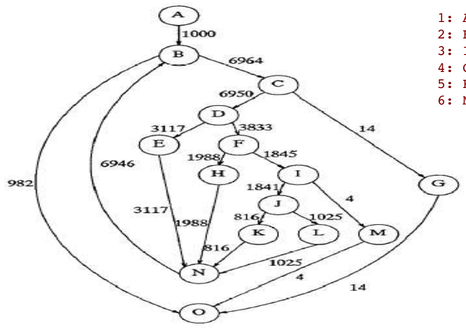
Chain Formation: Top-Down

1. Place the entry block of the procedure
2. Among all unplaced successors, select and append the one with the largest dynamic count
3. If all successors have been selected, pick among the unselected blocks with the largest connection to the already selected blocks
4. Continue until all blocks are placed.

Chain Formation: Bottom-Up

1. Consider each block to be the head an the tail of a new chain
2. Consider the edges from largest to smallest weight. Two different chains are merged if the arc connects the tail of one chain to the head of the other

Chain Formation: Example



```
1: A
2: E N B C D F H
3: I J L
4: G O
5: K
6: M
```

Precedence Relation

- Order chains such that non-taken conditional branches point forward (branch prediction)
- Not always possible
  - Prefer the edge with highest weight
- 6 conditional branches in our example

B to C/O	C to D/G	D to E/F
F to H/I	I to J/M	J to K/L

- Final order: A , E-N-B-C-D-F-H, I-J-L, G-O, K, M

More Sophisticated Techniques

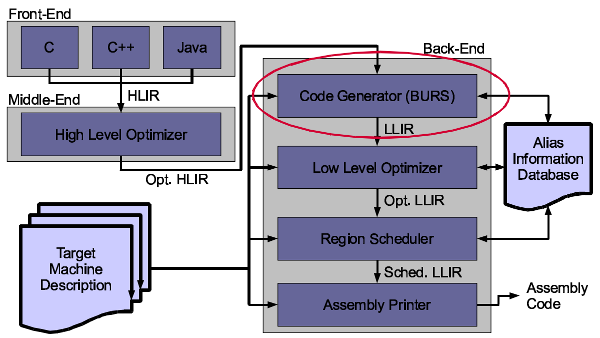
- Procedure Splitting
- Procedure Inlining (IMPACT compiler)
- Cache Line Coloring
  - Assign each cache line a different color
  - A parent function gets a color different from its descendants, as an attempt to prevent cache conflicts when they call each other
- Temporal Order Placement

Instruction Selection

***Task**: Translate the abstract syntax tree (AST) to concrete machine instructions supported by the target architecture*

- In general, many different combinations of machine instructions are semantically equivalent
- Usually, a cost model is used to balance among different optimization goals (performance, code size, energy).

Phases of an ILP Oriented Compiler



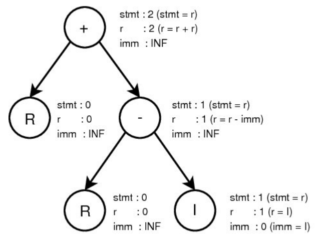
Scope

- Single (abstract) instruction
  - peephole approach
  - simple / efficient
- Statements (expression trees)
  - efficient (tree pattern matching)
  - optimal for each statement
- Functions / blocks
  - NP complete in general

Tree Pattern Matching

- Machine instructions represented by tree patterns
- Patterns have associated *costs* and *semantic actions*
- Two-phase approach
  - labeling phase: find a min-cost cover of the AST
  - reduction phase: apply semantic actions bottom-up
- Linear in the size of the tree

Example



RuleNr	Pattern	Cost	Emit
0	`stmt = r`	0	
1	`r = r + r`	1	add
2	`r = r - imm`	1	subi
3	`r = r - r`	1	sub
4	`r = R`	0	
5	`imm = I`	0	
6	`r = I`	1	li

Result:
```
subi r1 = R - I
add r2 = r1 + R
```

Limitations

- Limited scope; global flow of information is not visible to the matcher
- Cannot cope with general DAG patterns
- More sophisticated approaches:
  - *DAG based techniques*
    - NP complete in general (Ertl99)
    - Linear programming (Wilson95, LeupersBashford00)
  - *SSA-graph based techniques*
    - Model data and control flow of a whole function
    - Sound transition to PBQP

Partitioned Boolean Quadratic Programming (PBQP)

- Quadratic optimization problem

$$\min f = \sum_{1 \leq i < j \leq n} x_i . C_{i,j} . x_j^T + \sum_{1 \leq i \leq n} c_i . x_i^T$$

$$\text{s.t.} \forall i \in \{1 \ldots n\} : x_i . 1^T = 1$$

- Equivalent graph theoretic interpretation
  - boolean vectors are represented as nodes
  - edges represent cost matrizes

PBQP Based Instruction Selection

- Scholz, Eckstein (2003)

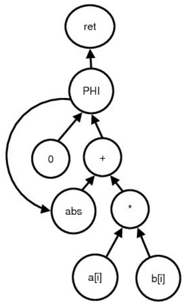
1. Construct the SSA graph
2. Transform the graph to an instance of PBQP
3. Solve the (NP complete) problem (heuristically)
4. Back-propagate the solution

SSA graphs

- Describe the computational flow of a whole function
- Based on static single assignment (SSA) form
  - Each variable is defined exactly once
  - Each use is dominated by its definition
  - If there are multiple definitions, an artificial Φ-function is inserted

Example: SSA graph

```
int f(short *a, short *b)
{
  int s1=0;
  loop(i) {
    s2 = PHI(s1, s3)
    s3 = abs(s2) + a[i] * b[i]
  }
  return(s2);
}
```



Rule Grammar

- Additional rules for matching Φ-nodes
- Two types of rules
  - base rules $nt_0 \leftarrow P(nt_1, ..., nt_n)$
  - chain rule $nt_0 \leftarrow nt_1$
- Straight forward automatic normalization

```
r ← +(r, *(r, r)) : c
=> t ← *(r, r) : 0
   r ← +(r, t) : c
```

Problem Transformation

- Main idea: PBQP and SSA graphs are structurally equivalent
- For each node, the domain for the decision vectors is defined by the set of applicable base rules
- Cost matrizes represent the least transition costs among the particular nonterminals
  - costs for transitions of the form $nt_0 \leftarrow nt_1$ are 0
  - costs for impossible transitions are set to infinity

Heuristic PBQP Solver

- Reduction Phase
  - **Reduce I**: eliminates a node i of degree 1 by transferring costs c_i and matrix costs $C_{i,j}$ to the adjacent node j
  - **Reduce II**: for degree 2 nodes, cost vector c_i and the two adjacent cost matrices $C_{i,j}$ and $C_{i,k}$ are merged into a new cost matrix among j and k
  - **Reduce N**: heuristically select a local minimum for a node of degree > 2 and eliminate the node
- Reconstruction Phase

  reconstruct the graph in inverse order and select the corresponding rules

Comparison to Tree Pattern Matching

- Both methods can use the same grammar (automatic normalization, implizit rules for Φ nodes)
- Operates on the scope of a whole function rather than statements
- When applied to trees, the heuristic solver acts almost like a tree pattern matcher (what is the difference?)
- Heuristic solver terminates with a provable optimal solution in most cases

Dependence Testing

- Data Dependencies
  - Read-after-write (true/flow dependence)
  - Write-after-Read (anti dependence)
  - Write-after-write (output dependence)
- Data Dependence Graph (DDG)
  - Vertizes represent instructions
  - There is an edge among u and v, if u has to precede v due to data dependencies
- Acyclic, if loop carried dependencies are left out

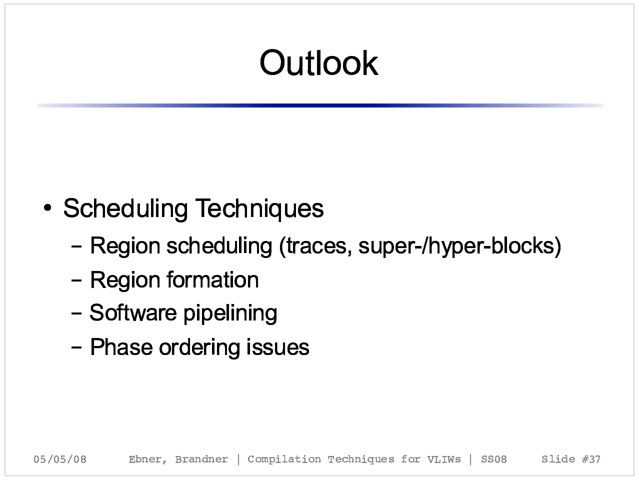
Simple Algorithm

- Backward scan over all instructions
- Maintain two maps
  - `def[r]`: last instruction that defines register r
  - `uses[r]`: list of all uses that are still “pending”
- Simplified memory model

  Ignore memory disambiguation by considering loads/stores to use/define an artificial memory resource

build_ddg

```
foreach i in I in reversed order:
  /* create a new node */
  n = new_node(i)
  /* insert edges */
  foreach definition o in i
    if(def[o]) new_edge(n, def[o], WAW)
    foreach p in uses[o]
      new_edge(n, p, RAW)
  foreach use u in i
    if(def[u]) new_edge(n, def[u], WAR)
  /* update temporary data structures */
  foreach definition o in i
    def[o] = n
    uses[o] = {}
  foreach use u in i
    uses[u].insert(n)
```



Outlook

- Scheduling Techniques
  - Region scheduling (traces, super-/hyper-blocks)
  - Region formation
  - Software pipelining
  - Phase ordering issues