Last Lectures (1)

- Code layout techniques
  - Basic block placement
  - Function placement
  - Cache-line coloring
- Instruction selection
  - Transform the high-level IR to low-level IR
  - Optimize for size and speed
  - Scope: Expression, Statement, Block/Function

Last Lectures (2)

- Tree pattern matching
  - Two-phase approach (label & reduce)
- DAG based
  - Generally NP complete
  - Solved using heuristics, linear programming (LP)
- Function global
  - Large problem size
  - Solved using heuristics, or LP
  - Quadratic optimization (PBQP)

In Today's Lecture

- List Scheduling
  - Forward
  - Backward/Delay Slots
- Resource Models
  - Reservation Tables
  - Automatons
- Trace Scheduling
  - Trace formation
  - Compensation code
Assignments

- Small error in the makefile for example 1
  - Causes the decoder to use the floating-point IDCT
  - Should have used the integer based IDCT
- Please download the corrected makefile from our homepage
  http://www.complang.tuwien.ac.at/cd/vliw

Instruction Scheduling

- Reorder instructions to
  - Minimize the execution time
  - Maximize the utilization of computational resources
  - Data dependencies need to be preserved
- For VLIWs
  - Essential for correct code & high performance
  - Often combined with instruction bundling
  - Accounting for clusters

Phases of an ILP Oriented Compiler

Phase Ordering

- Interacts heavily with other optimizations
  - Cluster Assignment
    - Limits freedom of the scheduler
  - Register allocation (RA)
    - Before scheduling
      - May limit freedom of the scheduler
      - Introduces false dependencies
    - After scheduling
      - May increase register pressure
      - May lead to spilling
Scheduling and Clusters

- Clustering adds significant complexity to scheduling
  - Usually done beforehand
  - Reminisce a min-cut problem
    - But different constraints
    - Do not minimize the moves, but execution time
- A promising approach
  - Color the DDG and find partial connected components (PCC)
  - Iterative refinement by reassigning PCCs

Scheduling and Register Allocation

- Typically schedule pre & post RA
- Integrated Prepass Scheduling
  - Two scheduling modes
  - CSR: Schedule to minimize register usage
  - CSP: Schedule to minimize pipeline delays
  - Estimate the current register usage
- Integrated techniques
  - Solve both problems simultaneously

Scheduling Outline

Visit each basic block of a function
1. Load a model of the architecture
   - Functional units
   - Encoding constraints
   - Pipeline information
2. Calculate the data dependence graph
3. Reorder the instructions

List Scheduling

- Operates on the data dependence graph
  1. Select a ready node of the DDG
  2. Schedule the associated instruction
  3. Remove the node from the DDG
  4. Repeat until the DDG is empty
- A node is available if all predecessors in the DDG already have been scheduled
- A node is ready if it is available, and all latency constraints are met
Ready Queue

- Selection of the next ready node to be scheduled is crucial
  - Store ready nodes in a priority queue
  - Important priority criteria
    - Instruction latency and type
    - Longest path to a root node of the DDG
    - Number of predecessors in the DDG
    - etc.
Example: List Scheduling (4)

\[ (0) \ i = 0; \ \\
(1) \ \text{sum} = 0; \ \\
(2) \ L: \ t1 = i \times 4 \ \\
(3) \ i = i + 1; \ \\
(4) \ b1 = i < n; \ \\
(5) \ t2 = ld[&a + t1]; \ \\
(6) \ \text{sum} = \text{sum} + t2; \ \\
(7) \ \text{if} (b1) \ \text{goto} \ L \ \\
(8) \ \text{return} \]

Schedule: 2-5-3

Example: List Scheduling (5)

\[ (0) \ i = 0; \ \\
(1) \ \text{sum} = 0; \ \\
(2) \ L: \ t1 = i \times 4 \ \\
(3) \ i = i + 1; \ \\
(4) \ b1 = i < n; \ \\
(5) \ t2 = ld[&a + t1]; \ \\
(6) \ \text{sum} = \text{sum} + t2; \ \\
(7) \ \text{if} (b1) \ \text{goto} \ L \ \\
(8) \ \text{return} \]

Final schedule: 2-5-3-5-4-6-7

Some steps skipped!

Example: Backward Scheduling (1)

\[ (0) \ i = 0; \ \\
(1) \ \text{sum} = 0; \ \\
(2) \ L: \ t1 = i \times 4 \ \\
(3) \ i = i + 1; \ \\
(4) \ b1 = i < n; \ \\
(5) \ t2 = ld[&a + t1]; \ \\
(6) \ \text{sum} = \text{sum} + t2; \ \\
(7) \ \text{if} (b1) \ \text{goto} \ L \ \\
(8) \ \text{return} \]

Assuming one branch delay slots!

Forward vs. Backward Scheduling

- The DDG can be processed in both directions
- Backward scheduling
  - Allows to handle (branch/load) delay slots naturally
  - Different handling of pipeline stalls
    - Once a stall is recognized the instruction that actually stalls has already been scheduled
Example: Backward Scheduling (2)

1. \( i = 0; \)
2. \( \text{sum} = 0; \)
3. L: \( t1 = i \times 4 \)
4. \( i = i + 1; \)
5. \( t2 = \text{ld}[a + t1]; \)
6. \( \text{sum} = \text{sum} + t2; \)
7. if ( \( b1 \) goto L)
8. return

Schedule 1-6
Assuming one branch delay slot!

Example: Backward Scheduling (3)

1. \( i = 0; \)
2. \( \text{sum} = 0; \)
3. L: \( t1 = i \times 4 \)
4. \( i = i + 1; \)
5. \( t2 = \text{ld}[a + t1]; \)
6. \( \text{sum} = \text{sum} + t2; \)
7. if ( \( b1 \) goto L)
8. return

Schedule 1-6
Assuming one branch delay slot!

Example: Backward Scheduling (4)

1. \( i = 0; \)
2. \( \text{sum} = 0; \)
3. L: \( t1 = i \times 4 \)
4. \( i = i + 1; \)
5. \( t2 = \text{ld}[a + t1]; \)
6. \( \text{sum} = \text{sum} + t2; \)
7. if ( \( b1 \) goto L)
8. return

Schedule 1-6
Assuming one branch delay slot!
Alternatives to List Scheduling

- Linear programming (LP)
  - Very flexible
  - Allows to model additional constraints
    - e.g. utilize vertical & horizontal NOP features
  - Slow and complex
  - Many decision variables required
  - VLIW bundling adds extra overhead
- Constraint solving

Reservation Tables

- List of resources (units)
  - Virtual resources for other constraints
  - Max. number of concurrent uses
- Record for each instruction
  - Which resources are used
  - When are these resources used
  - How many cycles does each use take

Example: Reservation Tables

<table>
<thead>
<tr>
<th>Available Resources</th>
<th>Resources per Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1xMUL</td>
<td>* &lt;ESGES1.MUL&gt;</td>
</tr>
<tr>
<td>2xALU</td>
<td>* &lt;ESGES1.ALU&gt;</td>
</tr>
<tr>
<td>1xLD</td>
<td>* &lt;ESGES1.ALU&gt;</td>
</tr>
<tr>
<td>1xBRA</td>
<td>* &lt;ESGES1.ALU&gt;</td>
</tr>
<tr>
<td>1xESF*</td>
<td>id &lt;ESGES1.LD&gt;</td>
</tr>
<tr>
<td>1xESI*</td>
<td>gote &lt;ES6.BRANCH&gt;</td>
</tr>
</tbody>
</table>

* Virtual resources for encoding tasks
Finite State Automatons

- States of the automaton
  - Capture the architecture state
  - Other constraints (encoding, etc.)
- Transitions
  - Model the scheduling of instructions
  - Prohibit transition for instructions causing hazards or violating some constraint

Example: Finite State Automatons

Scheduling for VLIWs

- We already know
  - Branches limit the amount of achievable ILP
- Consequently
  - We would like to schedule across branches
  - The scope of basic blocks is too limited

Trace Scheduling (1)

- Extend instruction scheduling to traces
  - Traces allow to schedule across basic blocks
  - Developed for micro-code compaction by J. Fisher
- A sequence of basic blocks form a trace
  - Build a linear path through the CFG
  - May contain several entries (joins)
  - May contain several exits (splits)
Trace Scheduling (2)

- Code moved across a branch
  - May violate control dependencies
  - May violate data dependencies
- Insertion of compensation code is required
  - Causes some overhead in the compiler
  - May have negative impact on execution time
    - The compensation code slows down other paths

Example: Not a Traces

- This is not a valid trace!
- The path formed by BB0, BB2, BB3 is not linear

Example: Traces

Trace Scheduling Outline

1. Trace selection
   - Combine profitable blocks into traces
2. Trace buffering
   - Save variables live at the entries and exits of the trace (required for compensation code)
3. Scheduling
   - Invoke some scheduling algorithm
     (e.g. list scheduling)
4. Bookkeeping
   - Insert compensation code (if required)
Trace Selection (1)

1. Select a *hot* basic block
   i. Select heuristically some predecessor or successor
      * Append the block to the trace
      * Repeat this step
   ii. If no profitable candidate is available
      * Stop the trace formation
      * Start a new trace with the next hottest block available
2. Repeat until all blocks are assigned to a trace

Trace Selection (2)

- Forming profitable traces depends heavily on profiling information
  - Selection of *hot* blocks
  - Rating candidate blocks during trace expansion
- It is beneficial if the program executes mostly the same (few) paths and is predictable
- It is hard to find traces if all execution paths have equal frequencies

Example: Traces (1)

```plaintext
Traces:
1: [BB1, BB3, BB4, BB5]
```

Example: Traces (2)

```plaintext
Traces:
1: [BB1, BB3, BB4, BB5]
2: [BB2]
3: [BB5]
```
Example: Traces (3)

Which trace should be selected?

Handling Loops (1)

- Hierarchically decompose the CFG
  - Start with innermost loop
  - Schedule the loop
  - Replace the original loop with the scheduled trace
  - Schedule its surrounding loop
- Repeat until all loops have been processed

Handling Loops (2)
Limitations of Trace scheduling

- Sensitive to static branch prediction accuracy
  - Off-trace paths are penalized
  - Favor shorter traces in these cases
- Code size growth
  - Caused by compensation code
  - May hurt instruction cache
  - Can be controlled using thresholds
- Handling loops
  - Trace scheduling is an acyclic technique

Outlook

- Other forms of regions
  - SEME vs MEME regions
  - Superblocks vs Traces
  - Non-linear regions
  - Hyperblocks, Tregions
- Region enlargement techniques
- Cyclic scheduling
  - Software pipelining