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

Instruction Scheduling
- List Scheduling
  - forward
  - backward
- Alternative Approaches
- Resource Models
  - Reservation tables
  - Finite state automata

Scheduling for VLIW architectures
- Trace Scheduling
- Trace Selection
- Handling Loops
- Code Motion / Compensation Code

Different Types of Regions
- Superblocks
- Hyperblocks
- Treegions
Region Enlargement Techniques
Phases of an ILP Oriented Compiler

- Front-End
  - C
  - C++
  - Java
- Middle-End
  - LLIR
  - High Level Optimizer
  - OIR
- Back-End
  - Code Generator (B IRS)
  - LLIR
  - Low Level Optimizer
  - OIR
  - Region Scheduler
  - Sched LLIR
- Assembly
  - Assembly Prelink
  - Assembly Code

Instruction Scheduling

- Most fundamental ILP-oriented phase
- VLIW Scheduling
  - Identify and group operations that can be executed in parallel
  - Minimize schedule length
  - Obey data dependencies and resource limitations
  - Cyclic vs. acyclic scheduling
  - Phase ordering issues

Classification

- Regime
  - Operation
    - Cycle
- Search Strategy
  - Greedy
  - Backtracking
  - Exhaustive
- Flow Analysis
  - Linear
  - Graph
- Region Shape
  - Acyclic
  - Cyclic
- Blocks
  - Supra-Blocks
  - Trees
  - DGs

Scheduling Phases

- “Global” scheduling is too complex in general
- Basic blocks do not offer sufficient amounts of ILP for wide-issue machines
- VLIW architectures often include hardware support
- Phase Ordering
  - Region formation
  - Schedule compaction
Schedule Compaction

- List scheduling widely used in practice
- Operates on the data dependence graph
  1. Select and schedule a ready node from the DDG
  2. Repeat until the DDG is empty
- A node is available if all predecessors in the DDG have already been scheduled
- A node is ready if it is available, and all hardware constraints are met

List Scheduling - Limitations

- How to select from the list of ready nodes?
- List Schedulers often tend to be too greedy
  - Operations that occupy a resource for multiple cycles might be scheduled too early, preventing subsequent critical instructions to become ready
  - Operations scheduled too early might unnecessarily increase register pressure

Region Selection

- Iterative trace growing using the **mutual most likely** strategy
- Stops, whenever no mutual most likely blocks can be found or a backedge is encountered

Traces

- Definition
  - Linear sequence of blocks with possibly multiple entrances and exits
    - \( B_0, B_1, ..., B_n \)
  - Formal Properties
    1. Each basic block is a predecessor of the next on the list
    2. For any \( i \) and \( k \), there is no path \( B_i \to B_k \to B_j \), except for those passing through \( B_0 \)
  - Does not prohibit forward branches or control flow leaving and re-entering the region!
**Example Trace**

- [Diagram of example trace]

**Compensation Code**

- What happens if we move instructions across join/split points
- Preserve all paths from the original code sequence in the transformed control flow after scheduling
- Compensation code drastically complicates trace based schedulers

**Compensation Scenarios**

- [Four different compensation scenarios diagrammed]

**Superblocks**

- Superblocks are traces without **side entrances**
- Single-Entry Multiple-Exit (SEME)
- Formal Properties
  1. Each basic block is a predecessor of the next on the list
  2. For any i and k, there is no path \( B_i \rightarrow B_k \rightarrow B_j \) except for those passing through \( B_k \)
  3. There may be no branches into a block in the region, except to \( B_0 \)
Tail Duplication

- Additional restrictions facilitate superblock schedulers
- Stopping trace formation at every join point is — at the first glance — a severe restriction
- Tail duplication can be used to create a copy of the rest of the trace
- Tail duplication can be seen as a form of “compensation code” that is created before the schedule compaction phase

Superblock Formation

Experimental Results

Performance Improvement

Performance improvement for a k-issue processor in comparison to a scalar base processor model.
**Effect of Speculation**

Restricted Percolation: no support for disregarding exceptions for load, store, divide, and floating point instructions. Non-trapping versions of those instructions are included in the “General Percolation” model.

**Code Size Expansion**

**Hyperblock**

- SEME regions with internal control flow
- Relaxed variant of superblocks
- Employ predication to fold multiple control paths into a single superblock
  - Allows to create traces with higher execution probability
  - Removes side exits and its schedule constraints

**Hyperblock Example**
Treegions

- Tree of basic blocks within the CFG
- List of blocks $B_i, B_j, ..., B_k$ such that each $B_i$ except for $B_j$ has exactly one predecessor $B_i$ with $i < j$
- Any path through the treegion yields a superblock
- Tail duplication is used to enlarge treegions just like in the superblock case
- Often referred as "non-linear regions" (cf. "linear regions" for traces, superblocks)

Region Comparison

<table>
<thead>
<tr>
<th>Year Proposed</th>
<th>Race</th>
<th>Superblock</th>
<th>Hyperblock</th>
<th>Treegion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1979</td>
<td>true way, most likely</td>
<td>true way, most likely</td>
<td>predicated when possible</td>
<td>both ways</td>
</tr>
<tr>
<td>Policy at splits</td>
<td>enforce</td>
<td>apply</td>
<td>stop</td>
<td>stop</td>
</tr>
<tr>
<td>Policy at packedges</td>
<td>stop, but apply region enlargement techniques</td>
<td>stop, but apply region enlargement techniques</td>
<td>stop, but apply region enlargement techniques</td>
<td>stop, but apply region enlargement techniques</td>
</tr>
<tr>
<td>Proposed measures to increase region size</td>
<td>stop unrolling</td>
<td>stop duplication, peeling, unrolling, and expansion</td>
<td>stop duplication, peeling, unrolling, and expansion</td>
<td>stop duplication, peeling, unrolling, and expansion</td>
</tr>
</tbody>
</table>

Region Enlargement

- Can be used to trade code size for performance
  - Make extra copies of highly iterated code
- Most common techniques
  - Loop unrolling
  - Loop peeling
  - Branch target expansion
  - (reverse) if-conversion (hyperblocks)
Loop Unrolling

- Duplicate a loop body several times
  - preconditioning / postconditioning to handle trip counts mod n
  - preconditioning is not possible for data dependent loop exits
- Small loops are often completely unrolled
- Can be performed both before and after region formation

Loop Unrolling: Increasing Parallelism

for(int i=0; i<=N; ++i) {
  <<body(i)>>
}

i=0;
L:
E:  i++;     goto L;

i++; if(i>N) goto E; <<body(i)>>

i++; if(i>N) goto E; <<body(i)>>

i++; if(i>N) goto E; <<body(i)>>

...
**Dependence Removing Optimizations**

- Eliminate data dependencies among instructions within frequently executed regions
  - Register Renaming
  - Operation Migration
  - Induction Variable Expansion
  - Accumulator Variable Expansion
  - Operation Combining

**Register Renaming**

- Eliminates anti- and output-dependencies by assigning unique registers to different definitions of the same register.
- Important within individual bodies of an unrolled loop

**Operation Migration**

- Moves an instruction from a region where its result is not used to less frequently used regions
- A copy has to be placed at the target region of each exit where the defined variable is live
- All of the data dependencies associated with that instruction can be eliminated from the region

**Induction Variable Expansion**

- Eliminates redefinitions of induction variables within an unrolled loop
- Each definition of the induction variable is given a new register
- Patch code has to be inserted if the induction variable is used outside the region to recover the proper value
Accumulator Variable Expansion

- Accumulation operations often define the critical path within loops
- Very similar to induction variable expansion
- Replaces each definition of an accumulator variable with a new register
- Requires additional initialization in the preheader and code that accumulates the partial results at region exits
- Often unsafe for floating point operations

Operation Combining

- Eliminates flow dependencies among pairs of instructions with the same precedence and with a compile-time source operand
- Often arise between address calculations and memory access instructions

Outlook

- Cyclic Scheduling
  - Software Pipelining
  - Modulo Scheduling
- Data- / Control Speculation
- Predicated Execution