Reducing the Cost of Conditional Transfers of Control by Using Comparison Specifications

May 30, 2006
Authors and Affiliations

- William Kreahling - Western Carolina University
- Stephen Hines - Florida State University
- David Whalley - Florida State University
- Gary Tyson - Florida State University
• Conditional transfers of control are expensive.
  – consume a large number of cycles
  – cause pipeline flushes
  – inhibit other code improving transformations

• Conditional transfers of control can be broken into three portions.
  – comparison (boolean test)
  – calculation of branch target address
  – actual transfer of control

• Most work done focuses on branch target address or branch itself.

• This research focuses on the comparison portion of conditional transfers of control.
Separate Instructions

- comparison instruction sets a register
- accessed by the branch instruction
- advantage, freedom to encode all the necessary info

Disadvantages
- two instructions needed
- may stall at the comparison instruction
single instruction performs compare and branch

Advantages
- only one instruction
- branch reached sooner, prediction made sooner

Disadvantages
- less bits allocated for branch target address
- may limit constant that can be compared
Comparison Specifications with Cbranches

- Decouple the specification of the values to be compared with the actual comparison.
  - encoding flexibility of separate compare and branch instructions
  - efficiency of single compare and branch instruction

- New Instructions
  - comparison specification (cmpspec)
  - compare and branch (cbranch)
New Hardware

- comparison register file
- read/write ports for this file
- forwarding hardware
  - cmpspec $\rightarrow$ cbranch
- separate adder for calculating branch target address
- Comparison register file is accessed in first half of stage.
- GP register file accessed in second half of stage to get actual values.
- Values to be compared are passed to the execute stage.
- Constants may also stored in comparison register file.
• VPO compiler

• classic five-stage in-order pipeline

• Arm port of the SimpleScalar Simulator

• modified GNU tools (assembler)


Old Vs. New

1 \texttt{r[2]=MEM;}
2 \texttt{IC=r[2]?r[3];}
3 \texttt{PC=IC<0,L6;}

(a) Original RTLs

1 \texttt{r[2]=MEM;}
2 \texttt{c[0]=2,3;}
3 \texttt{PC=c[0]<,L6;}

(b) New RTLs

- (a) comparison on line 2, branch on line 3
- (b) cmpspec on line 2, cbranch on line 3
1. $r[2] = MEM$
3. $PC = IC < 0, L6$;

(a) Original RTLs

1. $r[2] = MEM$
2. $c[0] = 2, 3$
3. $PC = c[0] <, L6$

(b) New RTLs

<table>
<thead>
<tr>
<th>Inst</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>branch</td>
<td>IF</td>
<td>stall</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inst</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmpspec</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cbranch</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Loop-Invariant Code Motion

(a) Original Code
L3:
2 \( r[2] = \text{MEM}; \)
3 \( \text{IC} = r[1] \? r[2]; \)
4 \( \text{PC} = \text{IC} < 0, L3; \)

(b) Code with Cmpspec
L3:
2 \( r[2] = \text{MEM}; \)
3 \( c[0] = 1, 2; \)
4 \( \text{PC} = c[0] <, L3; \)

(c) Cmpspec out of Loop
L3:
2 \( r[2] = \text{MEM}; \)
3 \( c[0] = 1, 2; \)
4 \( \text{PC} = c[0] <, L3; \)

- cmpspecs within loops can typically be moved into loop preheaders
- pay cost once, when loop is entered
- values within registers being compared may change, cmpspec does not
1 \quad c[0]=1,2;
2 \quad L3:
3 \quad r[2]=MEM;
4 \quad PC=c[0]<,L3;

(c) Cmpspec out of Loop

<table>
<thead>
<tr>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>inst</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>load</td>
</tr>
<tr>
<td>cbranch</td>
</tr>
</tbody>
</table>

LCTES 2006
Loop-Invariant Code Motion – cont

1 L2:
2 \(c[0]=2,3\);
3 \(PC=c[0]==,L6;\)
4 ... 
5 \(c[0]=5,12;\)
6 \(PC=c[0]!=,L5;\)
7 ... 
8 // br L2;

(a) Before Renaming

1 L2:
2 \(c[0]=2,3;\)
3 \(PC=c[0]==,L6;\)
4 ... 
5 \(c[1]=5,12;\)
6 \(PC=c[1]!=,L5;\)
7 ... 
8 // br to L2;

(b) After Renaming

1 \(c[0]=2,3;\)
2 \(c[1]=5,12;\)
3 L2:
4 ... 
5 \(PC=c[0]==,L6;\)
6 ... 
7 \(PC=c[1]!=,L5;\)
8 // br to L2;

(c) After Code Motion

- cmpspecs usually reference \(c[0]\)
- conflict occurs rename a comparison register
- no free registers, cmpspec remains inside loop
**Common Subexpression Elimination**

- CSE eliminates instructions that compute values already available
- normally, cannot eliminate comparison instructions
- in contrast, cmpspecs can often be eliminated
CSE – Reversing Conditions

(a) Original Code

1  
2  
3  \text{L2:}
4  \text{PC}=c[2]>, L6;
5  ... 
6  \text{PC}=c[3]<, L5;
7  ... 
8  // \text{br to L2};

(b) Reversed Condition

1  
2  
3  \text{L2:}
4  \text{PC}=c[2]>, L6;
5  ... 
6  \text{PC}=c[3]>, L5;
7  ... 
8  // \text{br to L2};

(c) After CSE

1  
2  
3  \text{L2:}
4  \text{PC}=c[2]>, L6;
5  ... 
6  \text{PC}=c[2]>, L5;
7  ... 
8  // \text{br to L2};

LCTES 2006
(a) Original Code

1. \(c[2] = 2, 0;\)
2. \(c[3] = 2, 1;\)
3. \(L2: \)
4. \(PC = c[2] \#>, L6;\)
5. ...
6. \(PC = c[3] \#<, L5;\)
7. // br to L2;

(b) After Modification

1. \(c[2] = 2, 0;\)
2. \(c[3] = 2, 0;\)
3. \(L2: \)
4. \(PC = c[2] \#>, L6;\)
5. ...
6. \(PC = c[3] \#<=, L5;\)
7. // br to L2;

(c) After CSE

1. \(c[2] = 2, 0;\)
2. \(L2: \)
3. \(PC = c[2] \#>, L6;\)
4. ...
5. \(PC = c[2] \#<=, L5;\)
6. ...
7. // br to L2;
(a) Identical Bit Pattern

1. \( c[4] = 2, 1 \);
2. \( \text{PC}=c[4] \leq, L6; \)
3. ...
4. \( c[4] = 2, 1 \);
5. \( \text{PC}=c[4] \#==, L5; \)
6. ...

(b) After CSE

1. \( c[4] = 2, 1 \);
2. \( \text{PC}=c[4] \leq, L6; \)
3. ...
4. \( \text{PC}=c[4] \#==, L5; \)
5. ...

## Register Encoding & New Instructions

### Comparison Register

<table>
<thead>
<tr>
<th>15-12</th>
<th>11-4</th>
<th>3-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg num</td>
<td>unused</td>
<td>reg num</td>
</tr>
<tr>
<td>reg num</td>
<td>constant</td>
<td></td>
</tr>
</tbody>
</table>

### New Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpspec <code>&lt;creg&gt;,idx1,val;</code></td>
<td>Assigns an index and an index or a constant.</td>
</tr>
<tr>
<td>cbr <code>&lt;creg&gt;&lt;rel_op&gt;,&lt;label&gt;;</code></td>
<td>Comparison register contains indices</td>
</tr>
<tr>
<td>cbri <code>&lt;creg&gt;&lt;rel_op&gt;,&lt;label&gt;;</code></td>
<td>Comparison register contains an index and a constant</td>
</tr>
<tr>
<td>[l/s]cfd <code>&lt;reg&gt;,{register list};</code></td>
<td>CISC inst - stores/loads comparison registers to/from stack</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>adpcm</td>
<td>adaptive pulse modulation encoder</td>
</tr>
<tr>
<td>bitcount</td>
<td>bit manipulations</td>
</tr>
<tr>
<td>crc32</td>
<td>cyclic redundancy check</td>
</tr>
<tr>
<td>fft</td>
<td>fast Fourier transform</td>
</tr>
<tr>
<td>ispell</td>
<td>spell checker</td>
</tr>
<tr>
<td>patricia</td>
<td>routing using reduced trees</td>
</tr>
<tr>
<td>rsynth</td>
<td>text-to-speech analysis</td>
</tr>
<tr>
<td>stringsearch</td>
<td>search words</td>
</tr>
<tr>
<td>tiff</td>
<td>convert a color TIFF image to b/w</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>basicmath</td>
<td>simple math calculations</td>
</tr>
<tr>
<td>blowfish</td>
<td>block encryption</td>
</tr>
<tr>
<td>dijkstra</td>
<td>shortest path problem</td>
</tr>
<tr>
<td>jpeg</td>
<td>image compression</td>
</tr>
<tr>
<td>lame</td>
<td>MP3 encoder</td>
</tr>
<tr>
<td>qsort</td>
<td>quick sort of strings</td>
</tr>
<tr>
<td>sha</td>
<td>exchange of cryptographic keys</td>
</tr>
<tr>
<td>susan</td>
<td>image recognition</td>
</tr>
</tbody>
</table>
Dynamic Micro-Op Counts

- Average savings 5.6%
  - Greatest savings came from \textit{adpcm} at roughly 18%.
  - \textit{ispell} was around 4\% worse.

- lack of profile data
  - saves and restores of comparison registers
  - loop preheader executing more than loop body

- Majority of savings comes from loop-invariant code motion 5.3%.

- CSE contributes another 0.3%.
• Large portion of savings from not-stalling at cmpspec, 5.2%.
  – Greatest savings came from stringsearch at roughly 18%.
  – Loss of roughly 3% with qsort.

• Loop-invariant code motion contributes around 0.9%.

• CSE contributes about 0.1%.
Branch Prediction

- higher misprediction penalty for cbranches (like implicit branches)
- benefits of new instructions outweigh misprediction penalty
- modern more efficient branch predictors can be used
<table>
<thead>
<tr>
<th></th>
<th>bimodal-128</th>
<th>gshare-256</th>
<th>gshare-512</th>
<th>gshare-1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Micro-ops Reduced</td>
<td>5.6%</td>
<td>5.7%</td>
<td>5.7%</td>
<td>5.8%</td>
</tr>
<tr>
<td>Cycles Reduced</td>
<td>5.2%</td>
<td>5.2%</td>
<td>5.4%</td>
<td>6.0%</td>
</tr>
<tr>
<td>Misprediction Rate</td>
<td>10%</td>
<td>9.9%</td>
<td>8.1%</td>
<td>6.9%</td>
</tr>
</tbody>
</table>
Future Work

- Profiling could be better used to guide optimizations like loop-invariant code motion.
  - cases where loop header is executed more frequently than the loop body

- With better analysis there should be more opportunities for CSE on cmpspecs.

- Implement technique on the Thumb.

- Implement loop unrolling in VPO.
Conclusions

- Specification of the comparison is decoupled from the comparison itself.
- Execution cycles are decreased because processor does useful work during the cmpspec.
- Optimizations that cannot be applied to traditional comparisons can be applied to cmpspecs.

Summary

- 5.6% reduction in dynamic instruction counts
- 5.2% reduction in execution cycles
Questions?
• One or more instructions following the branch are executed regardless of whether branch is taken or not taken.

• Compiler needs to fill the delay slots.

• Filled with no-ops if cannot find an instruction.

• Moving a instruction from before the branch always does useful work.

• Instruction from after the branch is more tricky.

• In some architectures, instructions in delay slots can be nullified.
Branch Prediction

- Process of deciding which instruction to execute following a branch, before the outcome of a branch is known.
- Branch prediction buffer – low order bits of an instruction used to index into a table.
- Prediction bit used to predict outcome of branch.
Use the behavior of multiple instances of previous branches to make prediction.

Generalized: use the behavior of the last $m$ branches to choose among $2^m$ predictors each having $n$ bits.

GAg, PAg, GAp, PAp, Gshare
- G: Global, P: Per-address (1st level)
- A: Adaptive
- g: global, p: per-address (2nd level)
2-level Predictors

GA\textsubscript{g} → 

GA\textsubscript{p} → 

PA\textsubscript{g} → 

PA\textsubscript{p} → 

k bit shift reg

2^k 2-bit counter
• Most recent branch outcomes are recorded in BHR - Branch History register

• BHR is a single shift-register shared by all branches

• BHR xor’d with branch address to find entry in Pattern History Table
• Tournament or hybrid predictors combine two or more prediction methods.

• Different methods work better for different branches.

• Array of two bit saturating counters used to determine which branch method to use.

• Each branch prediction make prediction each time.

• McFarling conducted experiments (bimodal and gshare) in combination worked better then either separately.
Markov Predictors

- Techniques common in the field of data compression used in branch prediction.

- Work done by Chen, et. al., shows that correlating predictors are a simplification of an optimal predictor used in data compression.

- Predication by Partial Matching.

- Not feasible to build optimal predictors given the current level of technology.
• Simple neural methods use as alternatives to commonly used 2-bit counters.

• Preceptron predictors consider longer histories than 2-bit predictors using the same resources.

• Experiments show better results than McFarling style hybrid predictors.

• Very complex hardware needed, feasibility in question.
• Delay can occur while calculating the address of the branch target.

• BTB acts as a small cache of branch target addresses.

• Branch instruction’s address not in the BTB, prediction of not-taken occurs.
Branch Registers

• Uses traditional registers to hold the branch target address.

• Calculation of the branch target address is separated from the instruction that uses it.

• This new instruction exposed to other compiler optimizations (loop-invariant code motion)
• Conditional execution of an instruction based upon a boolean source operand.

• Predicated instructions are fetched regardless of their predicate value.

• Reduce the number of branches.

• Eliminate frequently mispredicted branches.
Loop Transformations

- Loop Unrolling
  - replicate loop $n$ number of times
  - reduce overhead of loop, reduce number of branches

- Loop Unswitching
  - applied to loop that have a branch with invariant conditions
  - loop is replicated inside forks of the branch
  - reduce loop overhead and enable parallelization
• Avoiding Conditional Branches

  • compiler tries to determine if branches can be avoided

  • find path from point after a conditional branch, back to branch where comparison is not affected.

  • intraprocedurally interprocedurally