# 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

# ◆ Introduction

- 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

- 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

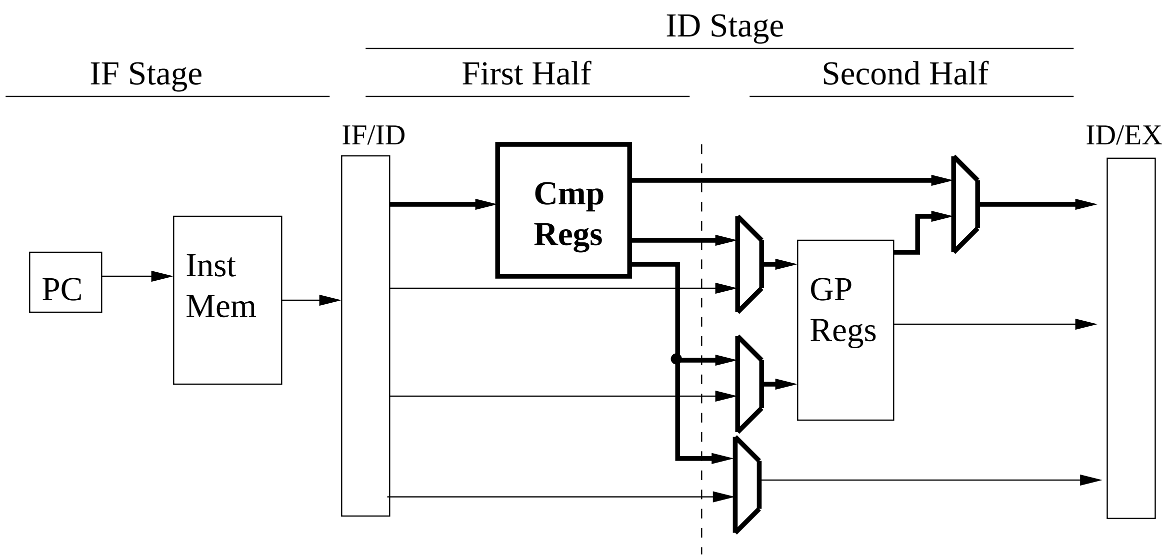
- Decouple the specification of the values to be compared with the actual comparison.
  - encoding flexibility of separate compare and branch instructions
  - efficiency of single compare and branch instruction

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

## ◆ New Hardware

- comparison register file
- read/write ports for this file
- forwarding hardware
  - cmpspec → cbranch
- separate adder for calculating branch target address

# ◆ Overview of Decode Stage




- 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.

# ◆ Experimental Environment

- VPO compiler
- classic five-stage in-order pipeline
- Arm port of the SimpleScalar Simulator
- modified GNU tools (assembler)

## ◆ Old Vs. New

```
1 r[2]=MEM;
2 IC=r[2]?r[3];
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

- (a) comparison on line 2, branch on line 3
- (b) cmpspec on line 2, cbranch on line 3

# ◆ Pipeline Diagrams

```
r[2]=MEM;
IC=r[2]?r[3];
PC=IC<0,L6;
```

(a) Original RTLs

```
r[2]=MEM;
c[0]=2,3;
PC=c[0]<,L6;
```

(b) New RTLs

Cycles

| | inst | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|
| 1) | load | IF | ID | EX | MEM | WB | | | |
| 2) | cmp | | IF | ID | stall | EX | MEM | WB | |
| 3) | branch | | | IF | stall | ID | EX | MEM | WB |

Cycles

| | inst | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|
| 1) | load | IF | ID | EX | MEM | WB | | | |
| 2) | cmpspec | | IF | ID | EX | MEM | WB | | |
| 3) | cbranch | | | IF | ID | EX | MEM | WB | |

# ◆ Loop-Invariant Code Motion

```
L3:
    r[2]=MEM;
    IC=r[1]?r[2];
    PC=IC<0,L3;
```

(a) Original Code

```
L3:
    r[2]=MEM;
    c[0]=1,2;
    PC=c[0]<,L3;
```

(b) Code with Cmpspec

```
    c[0]=1,2;
L3:
    r[2]=MEM;
    PC=c[0]<,L3;
```

(c) Cmpspec out of Loop

- 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

# ◆ Pipeline Diagram

```
    c[0]=1,2;
L3:
    r[2]=MEM;
    PC=c[0]<,L3;
```

(c) Cmpspec out of Loop

Cycles

| | inst | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|
| 1) | load | IF | ID | EX | MEM | WB | | |
| 2) | cbranch | | IF | ID | stall | EX | MEM | WB |

# ◆ Loop-Invariant Code Motion – cont

```
L2:
    c[0]=2,3;
    PC=c[0]==,L6;
    ...
    c[0]=5,12;
    PC=c[0]!=,L5;
    ...
    // br L2;
```

(a) Before Renaming

```
L2:
    c[0]=2,3;
    PC=c[0]==,L6;
    ...
    c[1]=5,12;
    PC=c[1]!=,L5;
    ...
    // br to L2;
```

(b) After Renaming

```
    c[0]=2,3;
    c[1]=5,12;
L2:
    PC=c[0]==,L6;
    ...
    PC=c[1]!=,L5;
    ...
    // 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

```
IC=r[2]?r[3];
PC=IC<0,L5;
...
IC=r[2]?r[3];
PC=IC>0,L5;
```

(a) Original Instructions

```
c[0]=2,3;
PC=c[0]<,L5;
...
c[0]=2,3;
PC=c[0]>,L5;
```

(b) New Instructions

```
c[0]=2,3;
PC=c[0]<,L5;
...
PC=c[0]>,L5;
```

(c) After CSE

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

# CSE – Reversing Conditions

```
    c[2]=2,3;
    c[3]=3,2;
L2:
    PC=c[2]>,L6;
    ...
    PC=c[3]<,L5;
    ...
    // br to L2;
```

(a) Original Code

```
    c[2]=2,3;
    c[3]=2,3;
L2:
    PC=c[2]>,L6;
    ...
    PC=c[3]>,L5;
    ...
    // br to L2;
```

(b) Reversed Condition

```
    c[2]=2,3;
L2:
    PC=c[2]>,L6;
    ...
    PC=c[2]>,L5;
    ...
    // br to L2;
```

(c) After CSE

# ◆ CSE – Constant off by one

```
      c[2]=2,0;
      c[3]=2,1;
L2:
      PC=c[2]#>,L6;
      ...
      PC=c[3]#<,L5;
      ...
      // br to L2;
```

(a) Original Code

```
      c[2]=2,0;
      c[3]=2,0;
L2:
      PC=c[2]#>,L6;
      ...
      PC=c[3]#<=,L5
        ;
      ...
      // br to L2;
```

(b) After Modification

```
      c[2]=2,0;
L2:
      PC=c[2]#>,L6;
      ...
      PC=c[2]#<=,L5
        ;
      ...
      // br to L2;
```

(c) After CSE

# ◆ CSE – Identical Cmpspecs

```
c[4]=2,1;
PC=c[4]<=,L6;
...
c[4]=2,1;
PC=c[4]#==,L5;
...
```

(a) Identical Bit Pattern

```
c[4]=2,1;
PC=c[4]<=,L6;
...
PC=c[4]#==,L5;
...
```

(b) After CSE

# ◆ Register Encoding & New Instructions

Comparison Register

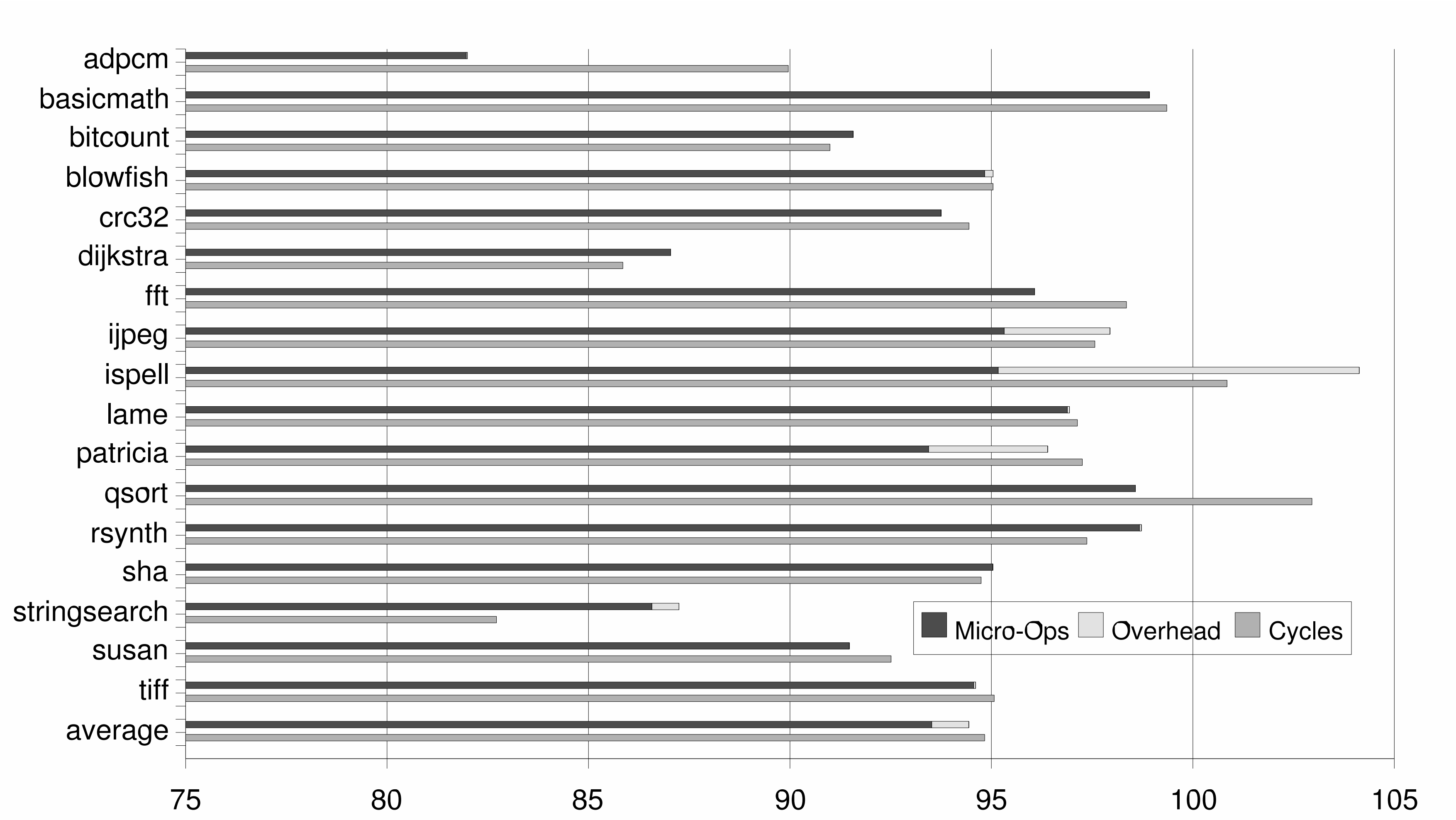
| 15-12 | 11-4 | 3-0 |
|---|---|---|
| reg num | unused | reg num |
| reg num | constant | |

| New Instructions | |
|---|---|
| cmpspec <creg>,index1,val; | Assigns an index and an index or a constant. |
| cbr <creg><rel_op>, <label>; | Comparison register contains indices |
| cbri <creg><rel_op>, <label>; | Comparison register contains an index and a constant |
| [l/s]cfd <reg>,{register list}; | CISC inst - stores/loads comparison registers to/from stack |

# ◆ Benchmarks Tested

| Name | Description | Name | Description |
|---|---|---|---|
| adpcm | adaptive pulse modulation encoder | basicmath | simple math calculations |
| bitcount | bit manipulations | blowfish | block encryption |
| crc32 | cyclic redundancy check | dijkstra | shortest path problem |
| fft | fast Fourier transform | ijpeg | image compression |
| ispell | spell checker | lame | MP3 encoder |
| patricia | routing using reduced trees | qsort | quick sort of strings |
| rsynth | text-to-speech analysis | sha | exchange of cryptographic keys |
| stringsearch | search words | susan | image recognition |
| tiff | convert a color TIFF image to b/w | | |

# ◆ Results



adpcm
basicmath
bitcount
blowfish
crc32
dijkstra
fft
ijpeg
ispell
lame
patricia
qsort
rsynth
sha
stringsearch
susan
tiff
average
75
80
85
90
95
100
105
Micro-Ops
Overhead
Cycles

## ◆ Dynamic Micro-Op Counts

- Average savings 5.6%
  - Greatest savings came from *adpcm* at roughly 18%.
  - *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%.

## ◆ Execution Cycles

- 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

# ◆ Mispredictions Rates

| | bimodal-128 | gshare-256 | gshare-512 | gshare-1024 |
|---|---|---|---|---|
| Micro-ops Reduced | 5.6% | 5.7% | 5.7% | 5.8% |
| Cycles Reduced | 5.2% | 5.2% | 5.4% | 6.0% |
| Misprediction Rate | 10% | 9.9% | 8.1% | 6.9% |

## ◆ 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

- Contributions
  - 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 dynamic instruction counts
  - 5.2% reduction in execution cycles

## ◆ The End

# Questions?

## ◆ . . .

## ◆ Delayed Branches

- 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.

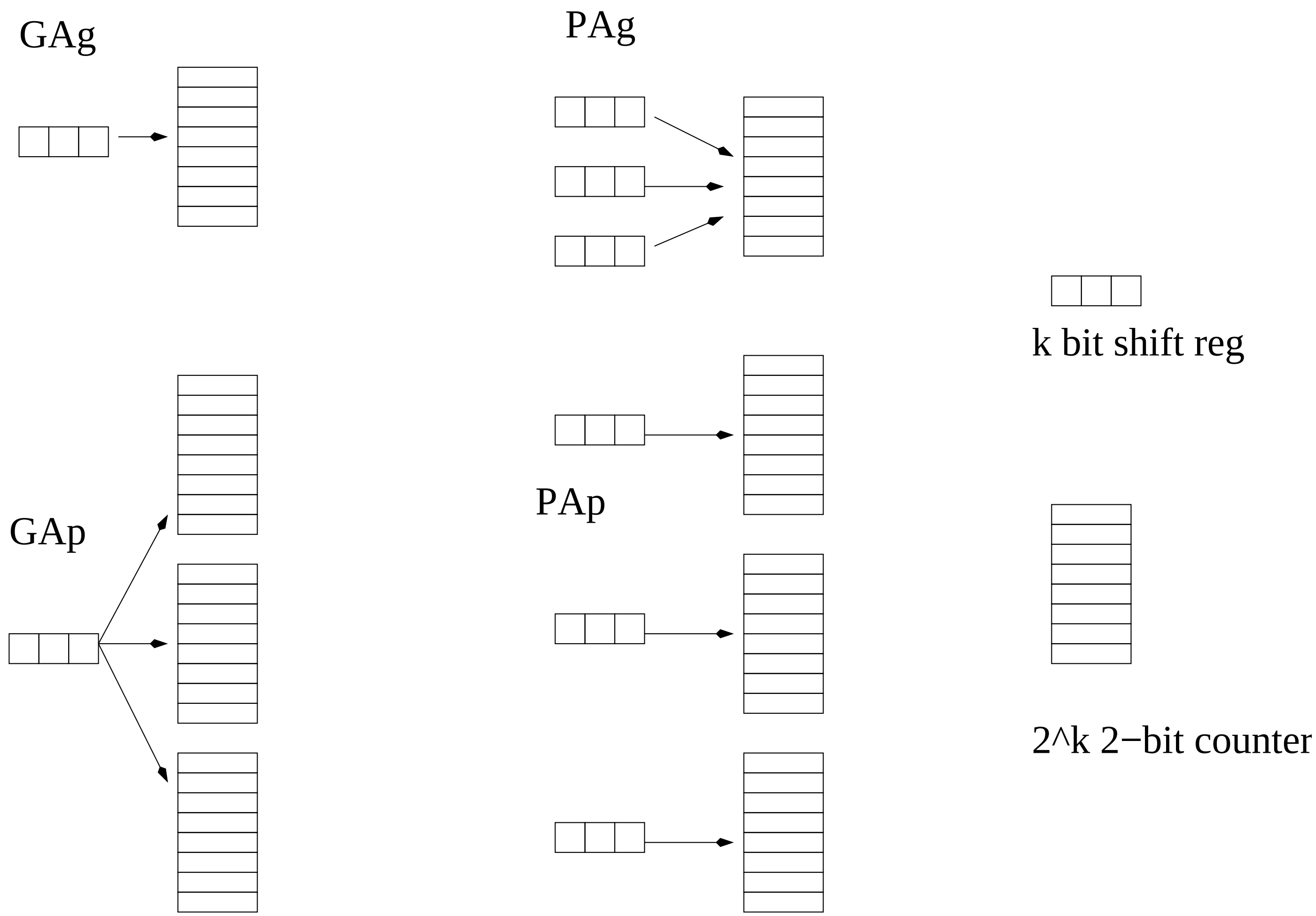
## ◆ Branch Prediction

- 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.

# ◆ Correlating or 2-level predictors

- 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



# ◆ GShare

- 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 Predictors

- Tournament or hybrid predictors combine two or more prediction methods.
- Different methods 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.

# ◆ Neural Methods for Branch Predictions

- 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.

## ◆ Branch Target Buffer

- 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)

## ◆ Predication

- 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