Exploiting Idle Floating-Point Resources for Integer Execution

Subramanya Sastry, Subbarao Palacharla, James E. Smith

University of Wisconsin, Madison
Motivation

• Partitioned integer and floating-point resources on current superscalar processors.

• Simplifies implementation.

• BUT....

Idle floating-point resources while executing integer code
Superscalar Microarchitecture

Floating-point subsystem

Integer subsystem

Branch outcomes

Communication datapath

Load/store data

Memory address
Contributions

• Compiler algorithms to exploit idle fp resources.
  - FP unit augmented to support integer operations.
  - Algorithms are simple, easy to implement, fast in practice.

• Results (on a 4-way issue superscalar machine):
  - 3%-23% perf. improvement on SpecINT95 programs.
  - FP programs do not experience slowdowns.
  - Occasional perf. improvements on FP programs.
Outline

• Motivation

• Hardware changes

• Preliminaries
  • Register Dependence Graph
  • Partitioning Heuristics

• Basic Partitioning Scheme
  • Algorithm
  • Results

• Advanced Partitioning Scheme
  • Copy instructions and Code duplication
  • Algorithm Overview
  • Results
Hardware Changes

Minimal hardware changes

• No extra buses, registers, register file ports required.

• Extra functional units for simple integer operations.
  - Assumes integer operation latency is not affected.
  - Integer multiply and divides not supported.

• Extend ISA to encode integer operations using fp regs.

• Changes in the spirit of Intel MMX, Sun VIS extensions.
For integer programs, provides:

- *Additional* issue and execution bandwidth.
- *Bigger* instruction window.
- *Larger* register file.
Terminology:
- Integer subsystem denoted as INT subsystem.
- Floating-point subsystem denoted as FP_a subsystem.

Identify integer code that can execute in FP_a subsystem.
- Divide code into FP_a and INT partitions.

Inter-partition communication:
- Through existing loads/stores.
- Through copy instructions.
- Avoid communication through code duplication.

Constraint: Only INT subsystem can execute loads/stores.
Register Dependence Graph (RDG)

• Graph representing pseudo-register dependences.
  - Computed by solving reaching-defns dataflow problem.

• Load/store instructions split into two nodes
  - Address nodes (assigned to INT).
  - Value nodes (could be assigned to $FP_a$).
RDG continued...

G = RDG of a program
LS(G) = Set of load/store address nodes
$LdSt$ slice = Instructions computing memory addresses

= $\bigcup_{v \in LS(G)} BackwardSlice(G, v)$

Branch slice & store-value slice are similarly defined.
I1: move $16, $0
$L5:
I2: lw $2, reg_mask
I3: sra $2, $2, $16
I4: andi $2, $2, 0x1
I5: beq $2, $0, $L4
I6: move $4, $16
I7: jal delete_equiv_reg
I8: lw $3, reg_tick
I9: sli $2, $16, 2
I10: addu $2, $2, $3
I11: lw $4, 0($2)
I12: bltz $4, $L4
I13: addu $4, $4, 1
I14: sw $4, 0($2)
$L4:
I15: addu $16, $16, 1
I16: slt $2, $16, 66
I17: bne $2, $0, $L5
Partitioning Heuristics

• Only INT subsystem can execute loads/stores => LS(G) is assigned to INT.

• Memory addressing/access on critical path.
  - Minimize communication overheads on these paths.
  - For integer programs, short addressing paths.
  - Entire LdSt slice assigned to INT partition.

• Branch and store-value slices can be assigned to FP_{a}.
Partitioning Heuristics (contd...) 

$LdSt$ slice close to 50% of dynamic instruction count.
- Use greedy strategy to maximize size of $FP_a$ partition.

Goal: *Maximize* size of the $FP_a$ partition.

*Minimize* instruction & communication overheads.
Roadmap ...

• Motivation

• Hardware changes

• Preliminaries
  • Register Dependence Graph
  • Partitioning Heuristics

• Basic Partitioning Scheme
  • Algorithm
  • Results

• Advanced Partitioning Scheme
  • Copy instructions and Code duplication
  • Algorithm Overview
  • Results
Basic Partitioning Scheme

Restriction:

- No extra communication instructions.
  => Existing loads/stores used for communication.

Partitioning condition:

- Consider undirected graph $G_u$ corresponding to $G$.
- $v \in F(G_u) \Rightarrow v$ is not reachable from any node in $I(G_u)$.

Algorithm:
1. Find connected components of $G_u$.
2. Components containing addr. nodes are assigned to $\text{INT}$.
3. Other components (containing only branch and store value computation) are assigned to $\text{FP}_a$. 
I1:    move  $16, $0

$L5:

I2:    lw      $2, reg_mask
I3:    sra     $2, $2, $16
I4:    andi   $2, $2, 0x1
I5:    beq     $2, $0, $L4
I6:    move    $4, $16
I7:    jal     delete_equiv_reg
I8:    lw      $3, reg_tick
I9:    sll     $2, $16, 2
I10:   addu    $2, $2, $3
I11:   lw      $f0, 0($2)
I12:   bltz,c  $f0, $L4
I13:   addu,c  $f0, $f0, 1
I14:   sw      $f0, 0($2)

$L4:

I15:   addu    $16, $16, 1
I16:   sll     $2, $16, 66
I17:   bne     $2, $0, $L5

INT PARTITION

FP_a PARTITION
Evaluation Methodology

Compiler:
- gcc-2.7.1 modified to do code partitioning.
- Generates code for an extended SimpleScalar ISA.
- Integer multiply & divide not supported in fp subsystem.

Benchmarks:
- SPECint95 programs.

Simulation Environment:
- Timing simulator based on the SimpleScalar toolset.
- Models both a conventional and an augmented arch.

Evaluation Metric:
- All benchmarks run to completion.
- Speedups based on cycles to completion.
Size of $\text{FP}_a$ partition

<table>
<thead>
<tr>
<th>Application</th>
<th>% Instructions in $\text{FP}_a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>perl</td>
<td>5.7</td>
</tr>
<tr>
<td>go</td>
<td>8.9</td>
</tr>
<tr>
<td>gcc</td>
<td>14.7</td>
</tr>
<tr>
<td>li</td>
<td>16.2</td>
</tr>
<tr>
<td>compress</td>
<td>10.9</td>
</tr>
<tr>
<td>jpeg</td>
<td>10.8</td>
</tr>
<tr>
<td>m88ksim</td>
<td>29.5</td>
</tr>
</tbody>
</table>
Performance Improvements on a 4-way issue (2 int + 2 fp) machine

0 1.6 3.7 4.3 5.0 7.0 5.2 20.8
perl go gcc li compress jpeg m88ksim

Performance improvements

Exploiting Idle FP Resources for Integer Execution
Advanced Partitioning Scheme

Limitations of the earlier scheme:

• Partitioning conditions force some branch and store-value computation to INT.

• Calling convention limitations:
  Int arguments & Int return values must be in int registers.
  => Argument/return-value slices assigned to INT.

Solutions:
- Introduce instructions to copy values.
- Duplicate code.
- Have to evaluate benefit of these extra instructions since they introduce overhead in the program.
I1: move $16, $0

$L5:
I2: lw $2, reg_mask
I3: sra $2, $2, $16
I4: andi $2, $2, 0x1
I5: beq $2, $0, $L4
I6: move $4, $16
I7: jal delete_equiv_reg
I8: lw $3, reg_tick
I9: sll $2, $16, 2
I10: addu $2, $2, $3
I11: lw $4, 0($2)
I12: bltz $4, $L4
I13: addu $4, $4, 1
I14: sw $4, 0($2)

$L4:
I15: addu $16, $16, 1
I16: slt $2, $16, 66
I17: bne $2, $0, $L5
I1:  move     $16, $0
I1c:  cp_to_fp $16, $f2

$L5:
I2:  lw         $f4, reg_mask
I3:  sra,c     $f4, $f4, $f2
I4:  andi,c    $f4, $f4, 0x1
I5:  beq,c     $f4, $0, $L4
I6:  move      $4, $16
I7:  jal       delete_equiv_reg
I8:  lw         $3, reg_tick
I9:  sll       $2, $16, 2
I10: addu      $2, $2, $3
I11: lw         $f0, 0($2)
I12: bltz,c    $f0, $L4
I13: addu,c    $f0, $f0, 1
I14: sw         $f0, 0($2)

$L4:
I15: addu      $16, $16, 1
I15c: cp_to_fp $16, $f2
I16: slt,c     $f4, $f2, 66
I17: bne,c     $f4, $0, $L5

LdSt Slice

Copies
I1:    move     $16, $0
I1d:   move,c  $f2, $0
$L5:
I2:    lw     $f4, reg_mask
I3:    sra,c  $f4, $f4, $f2
I4:    andi,c $f4, $f4, 0x1
I5:    beq,c  $f4, $0, $L4
I6:    move     $4, $16
I7:    jal    delete_equiv_reg
I8:    lw     $3, reg_tick
I9:    sll    $2, $16, 2
I10:   addu   $2, $2, $3
I11:   lw     $f0, 0($2)
I12:   bltz,c $f0, $L4
I13:   addu,c $f0, $f0, 1
I14:   sw     $f0, 0($2)
$L4:
I15:   addu   $16, $16, 1
I15d:  addu,c $f2, $f2, 1
I16:   slt,c  $f4, $f2, 66
I17:   bne,c  $f4, $0, $L5
Copying vs Duplication

**Duplication:**
+ No communication between register files.
– Requires copy/duplication of parents.
  => Effect might fan out along backward slice.

**Copying:**
– Requires communication between register files.
+ Does not affect parents.

**Implications:**
- Optimal decisions cannot be made using only local info.
- Heuristics used to pick between the two.
Let $G_u$ be the undirected RDG.

1. Assign $LdSt$ slice to INT.
2. Assign connected components in $G_u$ containing only branch and store-value computation to $FP_a$.
3. Make copying/duplication decisions for all nodes in $G_u$.
4. For other connected components of $G_u$, determine where to introduce copies/duplicates.
5. Insert copies/duplicates.
Step 4 of the algorithm
Step 4 of the algorithm
Step 4 of the algorithm
Size of $FP_a$ partition

<table>
<thead>
<tr>
<th>Program</th>
<th>Basic scheme</th>
<th>Advanced scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>perl</td>
<td>5.7</td>
<td>8.3</td>
</tr>
<tr>
<td>go</td>
<td>8.9</td>
<td>14.7</td>
</tr>
<tr>
<td>gcc</td>
<td>15.4</td>
<td>16.4</td>
</tr>
<tr>
<td>li</td>
<td>21.4</td>
<td>26.2</td>
</tr>
<tr>
<td>compress</td>
<td>25.3</td>
<td>32.1</td>
</tr>
<tr>
<td>jpeg</td>
<td>29.5</td>
<td>41.6</td>
</tr>
</tbody>
</table>

Subramanya
Sastry

Exploiting Idle FP Resources for Integer Execution
Performance Improvements

Performance improvements on a 4-way issue (2 int + 2 fp) machine

Exploiting Idle FP Resources for Integer Execution
Conclusions

• Can exploit idle fp resources for integer execution.

• Minimal hardware changes to support integer execution in the floating-point subsystem.

• Code partitioning done by the compiler.

• Copy instructions and code duplication are useful in getting good $FPA$ partitions.

• 9%-41% of dynamic instructions execute in $FPA$.

• 3%-23% performance improvements on a 4-way issue m/c.