Dynamic Instruction Reuse

Avinash Sodani
Guri Sohi

Computer Sciences Department
University of Wisconsin — Madison
Motivation

• Programs consist of static instructions
• Execution sees static instruction many times
  - often with same inputs
  → produces same result
  → no need to compute again

Exploited by

• Buffer results of instructions
• Reuse old result if input operands are same

Dynamic Instruction Reuse
Advantages

Instruction Reuse

+ permits dependent instructions to issue earlier
+ reduces resource contention
+ salvages useful work from misprediction squashes
+ completes chains of dependent instruction in single cycle
  ➔ potentially breaks dataflow limit
Outline

• Motivation
• What enables reuse?
• Implementing Reuse
• Three Reuse Schemes
• Some Results
• Summary
What enables reuse?

- **General Reuse**
  - due to the way computation is expressed
    ➔ same code visited with same data

- **Squash Reuse**
  - due to mis-speculation
General Reuse

\[
\text{find } (\text{key}, \text{list}) \\
\quad \text{foreach element in list} \\
\quad \quad \text{access element} \\
\quad \quad \text{if } (\text{key} == \text{element}) \text{ found} \\
\quad \text{not found}
\]

\[
\text{find } (a, \text{list}) \quad \text{find}(b, \text{list})
\]

\[
\begin{align*}
\text{iter 1} & \\
\text{iter 2} & \quad \text{same computation}
\end{align*}
\]
Squash Reuse

Dynamic Instruction Reuse

Avinash Sodani
U. of Wisconsin-Madison
Outline

• Motivation
• What enables reuse?
• Implementing Reuse
• Three Reuse Schemes
• Some Results
• Summary
Implementing Reuse

- Buffer results of instructions: **Reuse Buffer (RB)**
- Reuse if operands same as in previous execution: **Reuse Test**

**RB**
- Indexed by PC
- Reuse Test
- Selective invalidation

[Diagram of Reuse Buffer (RB) with events and Reuse Test]
Integrating RB in pipeline

- RB access begins in fetch stage
- Reuse happens in decode stage
Issues

• What information stored in RB?
• How is Reuse Test done?
• How is the information kept consistent?
Reuse Schemes

• Scheme $S_v$ : operand values
  + most aggressive
  - lot of bits

• Scheme $S_n$ : operand names
  + few bits
  - too conservative

• Scheme $S_{n+d}$ : operand names + dependences
  + improvement on $S_n$
  + $S_v$ performance at (near) $S_n$ cost
Scheme $S_v$

**What to store in RB?**

- Store results and operand values

**Reuse Test**

- Reuse result if operand values are same

**How to keep RB consistent?**

- loads invalidated when memory location overwritten.
- other instructions not invalidated
Scheme $S_V$ (cont’d)
Scheme $S_n$

What to store in RB?

• Store result and operand names

Reuse Test

• Reuse if result valid: valid bit

How to keep RB consistent?

• invalidate result when operand name overwritten
Scheme $S_n$ (cont’d)

<table>
<thead>
<tr>
<th>Time</th>
<th>Dynamic instructions</th>
<th>RB contents</th>
</tr>
</thead>
</table>
| T1   | $A : r1 \leftarrow r2 + 3$
     | $B : r3 \leftarrow r1 + 4$ | $A \quad r2$
     |                       | $B \quad r1$ |
| T2   | $R : r1 \leftarrow 4$  | $A \quad r2$
     |                       | $B \quad r1$ |
|      | invalidate            |             |
| T3   | $A : r1 \leftarrow r2 + 3$
     | $B : r3 \leftarrow r1 + 4$ | $A \quad r2$ |

B performs same computation — but not reused by $S_n$
What to store in RB?

• Store result, name and dependences

Reuse Test

• A reused if valid: valid bit
• B reused if A is latest producer of r1

How to keep RB consistent?

• Invalidate chain when inputs overwritten
**Scheme $S_{n+d}$ (cont’d)**

<table>
<thead>
<tr>
<th>Time</th>
<th>Dynamic instructions</th>
<th>RB contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>A : $r_1 \leftarrow r_2 + 3$</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>B : $r_3 \leftarrow r_1 + 4$</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$r_2$ A</td>
</tr>
<tr>
<td>T2</td>
<td>R : $r_1 \leftarrow 4$</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>(B not invalidated)</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$r_2$ A</td>
</tr>
<tr>
<td>T3</td>
<td>A : $r_1 \leftarrow r_2 + 3$</td>
<td>Reused</td>
</tr>
<tr>
<td></td>
<td>B : $r_3 \leftarrow r_1 + 4$</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$r_2$ A</td>
</tr>
</tbody>
</table>

-dependent chain reused — possibly in the same cycle
Experimental Evaluation

- OOO execution: window of 32 inst.: 4-way superscalar
- BTB: 2048 entries with 2-bit counters
- I-cache: 16K direct mapped, 32 byte line
- D-cache: 16K 2-way set assoc., 32 byte line
- Reuse Buffer
  - Size: 32, 128 and 1024 entries
  - Fully assoc. and 4-way set assoc. with FIFO replacement
  - 4 reads, 4 writes and 4 invalidations per cycle
- Benchmarks: Spec95 Int, Spec92 Int and others
Percentage Reuse ($S_v$)

Significant reuse observed
Dynamic Instruction Reuse

Avinash Sodani
U. of Wisconsin-Madison

Percentage Speedup ($S_v$)

- 32 RB entries
- 128 RB entries
- 1024 RB entries

Speedups significant too
Comparing Schemes

Harmonic mean

- Scheme $S_v$
- Scheme $S_n$
- Scheme $S_{n+d}$

Percent vs. RB Entries (32, 128, 1024)
Summary

Instruction Reuse

- reduces critical path of the computation
- reduces contention for resources
- reduces mis-prediction penalty

Significant instruction reuse

- in some cases > 50% : typically ~ 20%

Speedups also significant

- in several cases > 20% : typically ~ 10%
Dynamic Instruction Reuse

- Reuse prevalent among all instruction categories
- About 15% from load values
- About 25-35% from address calculation

40-50% of Reuse
Mean Speedups

Harmonic mean

<table>
<thead>
<tr>
<th>RB Entries</th>
<th>Scheme S_v</th>
<th>Scheme S_n</th>
<th>Scheme S_{n+d}</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>17</td>
<td>14</td>
<td>19</td>
</tr>
<tr>
<td>128</td>
<td>26</td>
<td>15</td>
<td>28</td>
</tr>
<tr>
<td>1024</td>
<td>43</td>
<td>17</td>
<td>30</td>
</tr>
</tbody>
</table>

Percent of max speedup:
- 17%
- 14%
- 19%
- 26%
- 15%
- 28%
- 43%
- 17%
- 30%
Squash Vs. General Reuse

- Recovering squashed work gives significant reuse.

- **Squash reuse** buys more — but general reuse important too
Collapsing true dependences

Data dependence resolution latency

- gcc
- compress
- eqntott
- espresso
- xilisp
- yacr2
- mpeg
- go
- m88ksim
- perl
- ijpeg
- vortex

Normalized resolution latency

32 RB entries
128 RB entries
1024 RB entries

Benmarks
Related Work

- Harbison’s value cache (*Tree machine*)
- Richardson’s result cache.
- Oberman and Flynn’s division and reciprocal caches

**Key Difference**

- They use address or operand values as index
  - limits the usefulness to long latency operations
  - Cannot reuse dependent chain in the same cycle
Squash Reuse

A = 0

A = 2

B = A + 1

Control Flow Graph