Improving Processor Efficiency by Statically Pipelining Instructions

Ian Finlayson, Brandon Davis, Peter Gavin, Gang-Ryung Uh*, David Whalley, Magnus Själander, Gary Tyson

Computer Science Department
Florida State University

*Computer Science Department
Boise State University

March 22, 2018

- Several inefficiencies with instruction pipelining.
- Pipeline registers are getting updated every cycle.
- Unnecessary register file accesses due to forwarding.
Motivation

Traditional Instruction Pipeline Example Datapath

- Constantly checking for hazards and forwarding.

Traditional vs Static Pipelining

- Unnecessary branch target addr calcs and BPB/BTB accesses.

Static Pipeline Example Datapath

- Internal registers can be explicitly accessed in SP instructions.
- A register file register is loaded into an RS register.

- The SE register contains a sign-extended value.

- A value from the L1 DC is loaded into the LV register.

- The OPER registers contain the results of ALU operations.
Static Pipeline Example Datapath

- Internal SP registers can be copied to a CP register.

- The TARG, SEQ, and PTB are used for transfers of control.

Overview of Compilation Process

Example

Source

```c
for (i = 0; i < 100; i++)
  a[i] += m;
```

MIPS

```mips
r[6]=<value m>;
r[9]=<address a>;
L2:
M[r[9]] = r[2];
PC = r[9] != r[5], L2;
```
### Initial Static Pipelined Code

```c
r[6] = LV;
r[9] = OPER2;
L2:
    # r[3] = M[r[9]];
    RS1 = r[9];
    LV = M[RS1];
    r[3] = LV;
    RS1 = r[3];
    RS2 = r[6];
    OPER2 = RS1 + RS2;
    r[2] = OPER2;
    # M[r[9]] = r[2];
    RS1 = r[9];
    RS2 = r[2];
    M[RS1] = RS2;
    # r[9] = r[9] + 4;
    SE = 4;
    RS1 = r[9];
    OPER2 = RS1 + SE;
    r[9] = OPER2;
    SE = offset(L2);
    TARG = PC + SE;
    RS1 = r[9];
    RS2 = r[5];
    PC = OPER2 != RS2, TARG;
```

### Copy Propagation

```c
r[6] = LV;
r[9] = OPER2;
L2:
    RS1 = r[9];
    LV = M[RS1];
    r[3] = LV;
    RS1 = r[3];
    RS2 = r[6];
    OPER2 = LV + RS2;
    r[2] = OPER2;
    RS1 = r[9];
    RS2 = r[2];
    M[RS1] = OPER2;
    # r[9] = r[9] + 4;
    SE = 4;
    RS1 = r[9];
    OPER2 = RS1 + SE;
    r[9] = OPER2;
    SE = offset(L2);
    TARG = PC + SE;
    RS1 = r[9];
    RS2 = r[5];
    PC = OPER2 != RS2, TARG;
```

### Motivation

Copy Propagation

```c
r[6] = LV;
r[9] = OPER2;
L2:
    RS1 = r[9];
    LV = M[RS1];
    r[3] = LV;
    RS1 = r[3];
    RS2 = r[6];
    OPER2 = LV + RS2;
    r[2] = OPER2;
    RS1 = r[9];
    RS2 = r[2];
    M[RS1] = OPER2;
    # r[9] = r[9] + 4;
    SE = 4;
    RS1 = r[9];
    OPER2 = RS1 + SE;
    r[9] = OPER2;
    SE = offset(L2);
    TARG = PC + SE;
    RS1 = r[9];
    RS2 = r[5];
    PC = OPER2 != RS2, TARG;
```

### Compilation

```c
r[6] = LV;
r[9] = OPER2;
L2:
    RS1 = r[9];
    LV = M[RS1];
    r[3] = LV;
    RS1 = r[3];
    RS2 = r[6];
    OPER2 = LV + RS2;
    r[2] = OPER2;
    RS1 = r[9];
    RS2 = r[2];
    M[RS1] = OPER2;
    # r[9] = r[9] + 4;
    SE = 4;
    RS1 = r[9];
    OPER2 = RS1 + SE;
    r[9] = OPER2;
    SE = offset(L2);
    TARG = PC + SE;
    RS1 = r[9];
    RS2 = r[5];
    PC = OPER2 != RS2, TARG;
```

### Results

Copy Propagation

```c
r[6] = LV;
r[9] = OPER2;
L2:
    RS1 = r[9];
    LV = M[RS1];
    r[3] = LV;
    RS1 = r[3];
    RS2 = r[6];
    OPER2 = LV + RS2;
    r[2] = OPER2;
    RS1 = r[9];
    RS2 = r[2];
    M[RS1] = OPER2;
    # r[9] = r[9] + 4;
    SE = 4;
    RS1 = r[9];
    OPER2 = RS1 + SE;
    r[9] = OPER2;
    SE = offset(L2);
    TARG = PC + SE;
    RS1 = r[9];
    RS2 = r[5];
    PC = OPER2 != RS2, TARG;
```

### Dead Assignment Elimination

```c
r[6] = LV;
r[9] = OPER2;
L2:
    RS1 = r[9];
    LV = M[RS1];
    r[3] = LV;
    RS1 = r[3];
    RS2 = r[6];
    OPER2 = LV + RS2;
    r[2] = OPER2;
    RS1 = r[9];
    RS2 = r[2];
    M[RS1] = OPER2;
    # r[9] = r[9] + 4;
    SE = 4;
    RS1 = r[9];
    OPER2 = RS1 + SE;
    r[9] = OPER2;
    SE = offset(L2);
    TARG = PC + SE;
    RS1 = r[9];
    RS2 = r[5];
    PC = OPER2 != RS2, TARG;
```
Redundant Assignment Elimination

\[ r[6] = LV; \]
\[ r[9] = OPER2; \]
\[
L2:
\]
\[ RS1 = r[9]; \]
\[ LV = M[RS1]; \]
\[ RS2 = r[6]; \]
\[ OPER2 = LV + RS2; \]
\[ RS1 = r[9]; \]
\[ M[RS1] = OPER2; \]
\[ SE = 4; \]
\[ RS1 = r[9]; \]
\[ OPER2 = RS1 + SE; \]
\[ r[9] = OPER2; \]
\[ SE = \text{offset}(L2); \]
\[ TARG = PC + SE; \]
\[ RS2 = r[5]; \]
\[ PC = OPER2 != RS2, TARG; \]
Loop Invariant Code Motion

\[
\begin{align*}
\text{r}[6] &= \text{LV}; \\
\text{r}[9] &= \text{OPER2}; \\
\text{L2:} & & \text{SE} = \text{offset}(\text{L2}); \\
\text{RS1} &= \text{r}[9]; & \text{TARG} &= \text{PC} + \text{SE}; \\
\text{LV} &= \text{M}[	ext{RS1}]; & \text{SE} &= 4; \\
\text{RS2} &= \text{r}[6]; & \text{L2:} & & \text{RS1} &= \text{r}[9]; \\
\text{OPER2} &= \text{LV} + \text{RS2}; & \text{LV} &= \text{M}[	ext{RS1}]; \\
\text{M}[	ext{RS1}] &= \text{OPER2}; & \text{RS2} &= \text{r}[6]; \\
\text{SE} &= 4; & \text{OPER2} &= \text{RS1} + \text{SE}; \\
\text{TARG} &= \text{PC} + \text{SE}; & \text{r}[9] &= \text{OPER2}; \\
\text{RS2} &= \text{r}[5]; & \text{RS2} &= \text{r}[5]; \\
\text{PC} &= \text{OPER2} \neq \text{RS2}, \text{TARG}; & \text{PC} &= \text{OPER2} \neq \text{RS2}, \text{TARG};
\end{align*}
\]

CP Register Allocation

\[
\begin{align*}
\text{r}[6] &= \text{LV}; & \text{CP1} &= \text{LV}; \\
\text{r}[9] &= \text{OPER2}; & \text{CP2} &= \text{OPER2}; \\
\text{SE} &= \text{offset}(\text{L2}); & \text{SE} &= \text{offset}(\text{L2}); \\
\text{TARG} &= \text{PC} + \text{SE}; & \text{TARG} &= \text{PC} + \text{SE}; \\
\text{SE} &= 4; & \text{SE} &= 4; \\
\text{L2:} & & \text{L2:} & & \text{RS1} &= \text{r}[9]; \\
\text{RS1} &= \text{r}[9]; & \text{LV} &= \text{M}[	ext{CP2}]; \\
\text{LV} &= \text{M}[	ext{CP2}]; & \text{RS2} &= \text{r}[6]; \\
\text{RS2} &= \text{r}[6]; & \text{OPER2} &= \text{LV} + \text{CP1}; \\
\text{OPER2} &= \text{LV} + \text{RS2}; & \text{M}[	ext{RS1}] &= \text{OPER2}; \\
\text{M}[	ext{RS1}] &= \text{OPER2}; & \text{M}[	ext{CP2}] &= \text{OPER2}; \\
\text{OPER2} &= \text{CP2} + \text{SE}; & \text{OPER2} &= \text{RS1} + \text{SE}; \\
\text{r}[9] &= \text{OPER2}; & \text{r}[9] &= \text{OPER2}; \\
\text{RS2} &= \text{r}[5]; & \text{RS2} &= \text{r}[5]; \\
\text{PC} &= \text{OPER2} \neq \text{RS2}, \text{TARG}; & \text{PC} &= \text{OPER2} \neq \text{RS2}, \text{TARG};
\end{align*}
\]
Final Code after Other Optimizations and Scheduling

```
CP1=LV; SE=4; RS2=r[5];
CP2=OPER2; SEQ=PC+1;
L2:
  LV=M[CP2]; OPER2=CP2+SE;
  OPER1=LV+CP1; PTB=b:SEQ;
  M[CP2]=OPER1; CP2=OPER2; PC=OPER2!=RS2,SEQ;
```

Experimental Evaluation

- Implemented a simulator based on SimpleScalar in-order MIPS.
- Baseline from VPO MIPS port with all optimizations enabled.
- Compiled and simulated 17 MiBench benchmarks.

Results

- Performance is improved by 9%.
- Code size is decreased by 7%.
- Processor energy usage is reduced by 27%.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Average SP to MIPS Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register File Reads</td>
<td>0.26</td>
</tr>
<tr>
<td>Register File Writes</td>
<td>0.33</td>
</tr>
<tr>
<td>Internal Register Writes</td>
<td>0.39</td>
</tr>
<tr>
<td>Branch Predictions</td>
<td>0.13</td>
</tr>
<tr>
<td>Target Address Calculations</td>
<td>0.61</td>
</tr>
<tr>
<td>BTB Accesses</td>
<td>0.00</td>
</tr>
</tbody>
</table>
SP allows for a new level of compiler optimizations to reduce energy usage by avoiding redundant or unnecessary operations in conventional pipelines.

- A low-level SP representation can be used and still achieve performance and code size improvements.
- SP reduces energy usage by significantly decreasing register file accesses, internal register writes, branch predictions, branch target address calculations, and completely eliminating the need for a BTB.

Questions???