Register Reuse Scheduling

Gergö Barany
gergo@complang.tuwien.ac.at

Institute of Computer Languages
Vienna University of Technology

ODES-9, Chamonix, France
April 2, 2011
We present **Register Reuse Scheduling** for integrated instruction scheduling and register allocation.

- Suboptimal instruction scheduling can cause expensive register spills.
- We minimize spilling by letting the register allocator choose scheduling constraints to facilitate reuse of registers.
- Our approach reduces spills by 8.9% and static spill costs by 3.4% on SPEC CPU2000.
The order of instructions within a block greatly impacts the register allocator’s freedom.

\[
\begin{align*}
A & \leftarrow \ldots \\
B & \leftarrow \ldots \\
C & \leftarrow \ldots \\
D & \leftarrow A + B
\end{align*}
\]

3 overlapping ranges

vs.

\[
\begin{align*}
A & \leftarrow \ldots \\
B & \leftarrow \ldots \\
D & \leftarrow A + B \\
C & \leftarrow \ldots
\end{align*}
\]

2 overlapping ranges
The order of instructions within a block greatly impacts the register allocator’s freedom.

\[
\begin{align*}
A & \leftarrow \ldots \\
B & \leftarrow \ldots \\
C & \leftarrow \ldots \\
D & \leftarrow A + B
\end{align*}
\]

vs.

\[
\begin{align*}
A & \leftarrow \ldots \\
B & \leftarrow \ldots \\
D & \leftarrow A + B \\
C & \leftarrow \ldots
\end{align*}
\]

3 overlapping ranges

2 overlapping ranges

Local heuristic schedulers to reduce register usage within a block leave room for improvement.

→ We use the register allocator’s global spill cost model to compute schedules that minimize spilling.
Register Reuse Scheduling (RRS) finds a schedule that reduces spilling.

Basic idea: Adding scheduling constraints to data dependence graph (DDG) allows register reuse by avoiding live range overlaps.
Register Reuse Scheduling (RRS) finds a schedule that reduces spilling.

Basic idea: Adding scheduling constraints to data dependence graph (DDG) allows register reuse by avoiding live range overlaps.

Register Reuse Scheduling (RRS)

- Identify possible register reuses and associated scheduling constraints.
Register Reuse Scheduling (RRS) finds a schedule that reduces spilling.

Basic idea: Adding scheduling constraints to data dependence graph (DDG) allows register reuse by avoiding live range overlaps.

Register Reuse Scheduling (RRS)

- Identify possible register reuses and associated scheduling constraints.
- Select acyclic set of reuse candidates that minimize overlaps.
Register Reuse Scheduling (RRS) finds a schedule that reduces spilling.

Basic idea: Adding scheduling constraints to data dependence graph (DDG) allows register reuse by avoiding live range overlaps.

Register Reuse Scheduling (RRS)

- Identify possible register reuses and associated scheduling constraints.
- Select acyclic set of reuse candidates that minimize overlaps.
- Allocate registers on simplified problem.
Register Reuse Scheduling (RRS) finds a schedule that reduces spilling.

Basic idea: Adding scheduling constraints to data dependence graph (DDG) allows register reuse by avoiding live range overlaps.

Register Reuse Scheduling (RRS)

- Identify possible register reuses and associated scheduling constraints.
- Select acyclic set of reuse candidates that minimize overlaps.
- Allocate registers on simplified problem.
- Map register assignment to scheduling constraints.
Identify **avoidable** live range overlaps and scheduling constraints.

1. load V1 ← a
2. load V2 ← b
3. mul V3 ← V1 * V2
4. load V4 ← c
5. load V5 ← d
6. add V6 ← V4 + V5
7. load V7 ← e
8. add V8 ← V1 + V7
9. mul V9 ← V6 * V8
10. add V10 ← V3 + V9
11. store h ← V10
Identify **avoidable** live range overlaps and scheduling constraints.

1. load $V_1 \leftarrow a$
2. load $V_2 \leftarrow b$
3. mul $V_3 \leftarrow V_1 \times V_2$
4. load $V_4 \leftarrow c$
5. load $V_5 \leftarrow d$
6. add $V_6 \leftarrow V_4 + V_5$
7. load $V_7 \leftarrow e$
8. add $V_8 \leftarrow V_1 + V_7$
9. mul $V_9 \leftarrow V_6 \times V_8$
10. add $V_{10} \leftarrow V_3 + V_9$
11. store $h \leftarrow V_{10}$

Classical analysis: Live range of $V_5$ ends before live range of $V_9$, can reuse register.
Identify **avoidable** live range overlaps and scheduling constraints.

- Cannot allocate $V_1$ and $V_4$ to the same register in the given schedule.
  - But: Can reuse the same register for $V_1$ and $V_4$ if instruction 8 comes before instruction 4, or 6 before 1.
Identify register reuses

Identify **avoidable** live range overlaps and scheduling constraints.

1. load V1 ← a
2. load V2 ← b
3. mul V3 ← V1 * V2
4. load V4 ← c
5. load V5 ← d
6. add V6 ← V4 + V5
7. load V7 ← e
8. add V8 ← V1 + V7
9. mul V9 ← V6 * V8
10. add V10 ← V3 + V9
11. store h ← V10

- Can reuse register for **V3** and **V6** *if* instruction 9 comes before 3.
- Reuse legal in **any** schedule that fulfills the additional constraint.
Possible reuse candidates:

<table>
<thead>
<tr>
<th>Reuse</th>
<th>DDG arc</th>
<th>Reuse</th>
<th>DDG arc</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1 → V3</td>
<td>8 → 3</td>
<td>V5 → V2</td>
<td>6 → 2</td>
</tr>
<tr>
<td>V1 → V8</td>
<td>3 → 8</td>
<td>V5 → V3</td>
<td>6 → 3</td>
</tr>
<tr>
<td>V1 → V9</td>
<td>3 → 9</td>
<td>V5 → V7</td>
<td>6 → 7</td>
</tr>
<tr>
<td>V2 → V9</td>
<td>3 → 9</td>
<td>V5 → V8</td>
<td>6 → 8</td>
</tr>
<tr>
<td>V4 → V1</td>
<td>6 → 1</td>
<td>V6 → V2</td>
<td>9 → 2</td>
</tr>
<tr>
<td>V4 → V2</td>
<td>6 → 2</td>
<td>V6 → V3</td>
<td>9 → 3</td>
</tr>
<tr>
<td>V4 → V3</td>
<td>6 → 3</td>
<td>V7 → V2</td>
<td>8 → 2</td>
</tr>
<tr>
<td>V4 → V7</td>
<td>6 → 7</td>
<td>V7 → V3</td>
<td>8 → 3</td>
</tr>
<tr>
<td>V4 → V8</td>
<td>6 → 8</td>
<td>V8 → V2</td>
<td>9 → 2</td>
</tr>
<tr>
<td>V5 → V1</td>
<td>6 → 1</td>
<td>V8 → V3</td>
<td>9 → 3</td>
</tr>
</tbody>
</table>
Identify Acyclic Reuse Candidate Set

Possible reuse candidates:

<table>
<thead>
<tr>
<th>Reuse</th>
<th>DDG arc</th>
<th>Reuse</th>
<th>DDG arc</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1 → V3</td>
<td>8 → 3</td>
<td>V5 → V2</td>
<td>6 → 2</td>
</tr>
<tr>
<td>V1 → V8</td>
<td>3 → 8</td>
<td>V5 → V3</td>
<td>6 → 3</td>
</tr>
<tr>
<td>V1 → V9</td>
<td>3 → 9</td>
<td>V5 → V7</td>
<td>6 → 7</td>
</tr>
<tr>
<td>V2 → V9</td>
<td>3 → 9</td>
<td>V5 → V8</td>
<td>6 → 8</td>
</tr>
<tr>
<td>V4 → V1</td>
<td>6 → 1</td>
<td>V6 → V2</td>
<td>9 → 2</td>
</tr>
<tr>
<td>V4 → V2</td>
<td>6 → 2</td>
<td>V6 → V3</td>
<td>9 → 3</td>
</tr>
<tr>
<td>V4 → V3</td>
<td>6 → 3</td>
<td>V7 → V2</td>
<td>8 → 2</td>
</tr>
<tr>
<td>V4 → V7</td>
<td>6 → 7</td>
<td>V7 → V3</td>
<td>8 → 3</td>
</tr>
<tr>
<td>V4 → V8</td>
<td>6 → 8</td>
<td>V8 → V2</td>
<td>9 → 2</td>
</tr>
<tr>
<td>V5 → V1</td>
<td>6 → 1</td>
<td>V8 → V3</td>
<td>9 → 3</td>
</tr>
</tbody>
</table>

Adding all reuses would cause cycles in the DDG!

→ Select an acyclic set.
We must select an acyclic set of reuse candidates.

**Ideally:** Let the register allocator choose an acyclic set of reuses during allocation. Future work...

**Currently:**
- Use a greedy heuristic to select a “most profitable” acyclic set.
- Profitability is judged by register allocator’s spill costs for the live ranges.
Selected scheduling constraints remove live range interferences.

Original interference graph

Relaxed interference graph

Relaxed interference graph is 3-colorable! Any valid schedule (with additional constraints) can be allocated using 3 registers.
Selected scheduling constraints remove live range interferences.

Original interference graph

Relaxed interference graph

Relaxed interference graph is 3-colorable!

⇒ Any valid schedule (with additional constraints) can be allocated using 3 registers.
Register allocation implicitly selects live range reuses.

Corresponding scheduling constraints must be added to the DDG.

Any resulting schedule is valid for the allocation by construction.
We implemented Register Reuse Scheduling in LLVM, using its PBQP register allocator.

**SPEC CPU2000 on ARM, vs. PBQP allocator:**
- Number of spills: average 8.9% reduction (up to 46.4%)
- Static spill costs: average 3.4% reduction (up to 15.4%)

Compile time: about $5 \times$ the time of PBQP register allocator (PBQP: 395 s, RRS: 2083 s on about 370 kLOC)
Summary

- Suboptimal instruction scheduling can cause expensive register spills.

- We minimize spilling by letting the register allocator choose scheduling constraints to facilitate reuse of registers.

- Our approach reduces spills by 8.9% and static spill costs by 3.4% on SPEC CPU2000.
Summary

- Suboptimal instruction scheduling can cause expensive register spills.

- We minimize spilling by letting the register allocator choose scheduling constraints to facilitate reuse of registers.

- Our approach reduces spills by 8.9% and static spill costs by 3.4% on SPEC CPU2000.

Thank you for your attention!

Acknowledgements: This work was supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) under contract P21842. *Optimal Code Generation for Explicitly Parallel Processors*, http://www.complang.tuwien.ac.at/epicomp/.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>PBQP</th>
<th>RRS</th>
<th>(\frac{RRS}{PBQP})</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Spills</td>
<td>Spill Costs</td>
<td>Spills</td>
</tr>
<tr>
<td>164.gzip</td>
<td>168</td>
<td>3.83e+03</td>
<td>168</td>
</tr>
<tr>
<td>175.vpr</td>
<td>1428</td>
<td>1.12e+04</td>
<td>1393</td>
</tr>
<tr>
<td>181.mcf</td>
<td>42</td>
<td>3.99e+01</td>
<td>32</td>
</tr>
<tr>
<td>186.crafty</td>
<td>2002</td>
<td>3.33e+02</td>
<td>1074</td>
</tr>
<tr>
<td>197.parser</td>
<td>441</td>
<td>6.12e+03</td>
<td>437</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>1445</td>
<td>1.23e+04</td>
<td>1539</td>
</tr>
<tr>
<td>254.gap</td>
<td>2270</td>
<td>6.10e+03</td>
<td>2263</td>
</tr>
<tr>
<td>255.vortex</td>
<td>1908</td>
<td>6.34e+03</td>
<td>1749</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>177</td>
<td>2.33e+03</td>
<td>176</td>
</tr>
<tr>
<td>177.mesa</td>
<td>8821</td>
<td>1.94e+04</td>
<td>8779</td>
</tr>
<tr>
<td>179.art</td>
<td>295</td>
<td>1.44e+04</td>
<td>257</td>
</tr>
<tr>
<td>183.equake</td>
<td>1092</td>
<td>6.48e+02</td>
<td>1015</td>
</tr>
<tr>
<td>188.ammp</td>
<td>7026</td>
<td>1.06e+04</td>
<td>6777</td>
</tr>
<tr>
<td>geometric mean</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>