Profile-Guided Optimizations

Last time
- Instruction scheduling
- Register renaming
- Balanced Load Scheduling
- Loop unrolling
- Software pipelining

Today
- More instruction scheduling
- Profiling
- Trace scheduling

Motivation for Profiling

Limitations of static analysis
- Compilers can analyze possible paths but must behave conservatively
- Frequency information cannot be obtained through static analysis

How runtime information helps
- Control flow information

```
if c
```

Optimize the more frequent path
(perhaps at the expense of the less frequent path)

- Memory conflicts

```plaintext
st r1,0(r5)
lod r2,0(r4)
```

If r5 and r4 always have different values,
we can move the load above the store
Profile-Guided Optimizations

Basic idea
- Instrument and run program on sample inputs to get likely runtime behavior
- Can use this information to improve instruction scheduling
- Many other uses
  - Code placement
  - Inlining
  - Value speculation
  - Branch prediction
  - Class-based optimization (static method lookup)

Profiling Issues

Profile data
- Collected over whole program run
- May not be useful (unbiased branches)
- May not reflect all runs
- May be expensive and inconvenient to gather
  - Continuous profiling [Anderson 97]
- May interfere with program
**Control-Flow Profiles**

**Commonly gather two types of information**
- Execution frequencies of basic blocks
- Branch frequencies of conditional branches
- Represent information in a weighted flow graph

<table>
<thead>
<tr>
<th></th>
<th>100</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>execution frequencies</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>70</td>
<td>branch frequencies</td>
</tr>
<tr>
<td>3</td>
<td>70</td>
<td>30</td>
</tr>
</tbody>
</table>

**Instrumentation**
- Insert instrumentation code at basic block entrances and before each branch
- Take average of values from multiple training runs
- Fairly inexpensive

---

**Code Motion Using Control Flow Profiles**

**Code motion across basic blocks**
- Increased scheduling freedom

- If we want to move s1 to A, we must move s1 to both A and B

move code above a join

- If we want to move s1 to B, we must move s1 to both B and C

move code below a split
**Code Motion Using Control Flow Profiles (cont)**

**Code motion across basic blocks**
- Increased scheduling freedom

- If we want to move s1 from B to A and if s1 would destroy a value along the A→C path, do renaming (in hardware or software)

- What if s1 might cause an exception?

- If we want to move s1 from B to A and if s1 would destroy a value along the A→C path, do renaming (in hardware or software)

- What if s1 might cause an exception?

**Memory-Dependence Profiles**

**Gather information about memory conflicts**
- Frequencies of address matches between pairs of loads and stores
- Attempts to answer the question: Are two references independent of one another?
- Concentrate on ambiguous reference pairs (those that the compiler cannot figure out)

- If this number is low, we can speculatively assume that st1 and ld2 do not conflict

**Instrumentation**
- Much more expensive (in both space and time) to gather than control flow information
- First perform control flow profiling
- Apply only to most frequently executed blocks
Trace Scheduling [Fisher 81] and [Ellis 85]

Basic idea
- We want large blocks to create large scheduling windows, but basic blocks are small because branches are frequent
- Create superblocks to increase scheduling window
- Use profile information to create good superblocks
- Optimize each superblock independently

Superblocks
- A sequence of basic blocks with a single entrance and multiple exits

Goals
- Want large superblocks
- Want to avoid early exits
- Want blocks that match actual execution paths

Trace Scheduling (example)

```plaintext
b[i] = “old”
a[i] = ...
if (a[i]>0) then
  b[i]=“new”;
else
  stmt X
  stmt Y
endif
 c[i] = ...

trace:
  b[i] = “old”
a[i] = ...
  b[i]=“new”;
  c[i] = ...
if (a[i]<=0) then goto repair
  continue:
  ...

repair:
  restore old b[i]
  stmt X
  stmt Y
  recalculate c[i]?
  goto continue
```
Trace Scheduling (cont)

**Three steps**
1. Create superblocks
2. Enlarge superblocks
3. Compact (optimize) superblocks

**1. Superblock formation**

– Create **traces** using **mutual-most-likely** heuristic
  (two blocks $A$ and $B$ are mutual-most-likely if $B$ is the most likely successor of $A$, and $A$ is the most likely predecessor of $B$)

– A **trace** is a maximal sequence of mutual-most-likely blocks that does not contain a back edge

– Each block belongs to exactly one trace

Trace Scheduling (cont)

**1. Superblock formation** (cont)

– Convert traces into **Superblocks**

– Use tail duplication to eliminate side entrances

– Tail duplication increases code size
Trace Scheduling (cont)

2. Superblock enlargement
   - Enlarge superblocks that are too small
   - Code expansion can hurt i-cache performance

Three techniques for enlargement
   - Branch target expansion
     - If the last branch in a superblock is likely to jump to the start of another superblock, append the contents of the target superblock to the first superblock
   - Loop peeling
   - Loop unrolling
     - These last two techniques apply to superblock loops, which are superblocks whose last blocks are likely to jump to their first blocks
     - Assume that each loop body has a single dominant path

Trace Scheduling (cont)

3. Optimizations
   - Perform list scheduling for each superblock
   - Memory-dependence profiles can be used to speculatively assume that load/store pairs do not conflict
     - Insert repair code in case the assumption is incorrect
   - Software pipelining
Speculation based on memory-dependence profiles (example)

\[
\begin{align*}
\text{b[i]} &= \text{"old"} \\
\text{a[i]} &= ... \\
\text{if (a[i]}>0) \text{ then} \\
   \quad \text{b[i]}&=\text{"new"}; \\
\text{else} \\
   \quad \text{stmt X} \\
   \quad \text{stmt Y} \\
\text{endif} \\
\text{c[i]} &= \text{a[j]}
\end{align*}
\]

\[
\begin{align*}
\text{trace:} \\
\quad \text{b[i]} &= \text{"old"} \\
\quad \text{c[i]} &= \text{a[j]} \\
\quad \text{a[i]} &= ... \\
\quad \text{b[i]}&=\text{"new"}; \\
\quad \text{if (i==j)} \text{ then goto deprepair} \\
\quad \text{if (a[i]<=0) then goto repair} \\
\quad \text{continue:} \\
\quad \quad \quad \ldots
\end{align*}
\]

\[
\begin{align*}
\quad \text{deprepair:} \\
\quad \quad \text{c[i]} &= \text{a[i]} \\
\quad \quad \text{if (a[i]<=0) then goto repair} \\
\quad \quad \text{goto continue}
\end{align*}
\]

\[
\begin{align*}
\text{repair:} \\
\quad \text{restore old b[i]} \\
\quad \text{stmt X} \\
\quad \text{stmt Y} \\
\quad \text{goto continue}
\end{align*}
\]

Enhancements to Profile-Guided Code Scheduling

Path profiling [Ball and Larus 96]
- Collect information about entire paths instead of about individual edges

```
\begin{align*}
\quad \text{Edge profiles} & \quad \text{Path profiles} & \quad \text{Path profiles}
\end{align*}
```

- Limit paths to some specified length (can thus handle loops)
- Can also stop paths at back edges
- Disadvantages of path profiling?
Lessons

Larger scope helps
- How can we increase scope? How do we schedule across control dependences?

Static information is limited
- Use profiles
- How else can profiles be used in optimization?
- Can we do these kinds of optimizations at runtime?

Concepts

Instruction scheduling
- Software pipelining
- Trace scheduling
- Both use profile information
- Both look at scopes beyond basic blocks

Miscellany
- Path profiling
- Speculative hedging
Next Time

Reading
– Mahlke’92

Lecture
– Speculation and predication