Profile-Guided Optimizations

Recall
- Instruction scheduling
  - List scheduling
  - Register renaming
  - Loop unrolling
  - Software pipelining
- Alias analysis
  - how can we use alias analysis for instruction scheduling?
  - what causes conservative results?

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
  10% 90%
  \[ \text{if } c \]
Optimize the more frequent path
(perhaps at the expense of the less frequent path)
- Memory conflicts
  \[ \text{st } r1,0(r5) \]
  \[ \text{ld } r2,0(r4) \]
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

Profile-Guided Optimizations

Recall
- Instruction scheduling
  - List scheduling
  - Register renaming
  - Loop unrolling
  - Software pipelining
- Alias analysis
  - how can we use alias analysis for instruction scheduling?
  - what causes conservative results?

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
  10% 90%
  \[ \text{if } c \]
Optimize the more frequent path
(perhaps at the expense of the less frequent path)
- Memory conflicts
  \[ \text{st } r1,0(r5) \]
  \[ \text{ld } r2,0(r4) \]
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**

#### 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 \( s_1 \) to A, we must move \( s_1 \) to both A and B
- If we want to move \( s_1 \) to B, we must move \( s_1 \) to both B and C

#### Instrumentation
- Move code above a join
- Move code below a split

### 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 \( st_1 \) and \( ld_2 \) 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
Profile-Guided Optimizations

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)

```
trace:
  b[i] = "old"
  a[i] = ...
  if (a[i]>0) then
    b[i]="new";
    c[i] = ...
  else
    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 (cont)
- Convert traces into Superblocks
- Use tail duplication to eliminate side entrances

"trace" is a maximal sequence of mutual most-likely blocks that does not contain a back edge
- Each block belongs to exactly one trace

Tail duplication increases code size
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

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)

```c
b[i] = "old"
a[i] = ...
if (a[i]>0) then
b[i]="new"
else
stmt X
stmt Y
endif
c[i] = a[j]
trace:
trace: b[i] = "old"
c[i] = a[j]
a[i] = ...
if (i==j) then goto deprepair
if (a[i]<0) then goto repair
continue:
...
deprepair:
c[i] = a[i]
if (a[i]<0) then goto repair
goto continue
repair:
restore old b[i]
stmt X
stmt Y
goto continue
```

---

Enhancements to Profile-Guided Code Scheduling

Path profiling [Ball and Larus 96]
- Collect information about entire paths instead of about individual edges
- 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
- Trace scheduling
- Uses profile information
- Looks at scopes beyond basic blocks

Miscellany
- Path profiling