

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

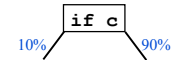
Motivation for Profiling

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



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

- Memory conflicts

```
st r1, 0(r5) } If r5 and r4 always have different values,  
ld r2, 0(r4) } 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

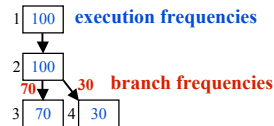
Profile data

- 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

CS553 Lecture

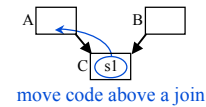
Profile-Guided Optimizations

6

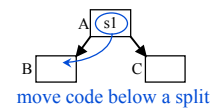
Code Motion Using Control Flow Profiles

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



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

CS553 Lecture

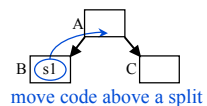
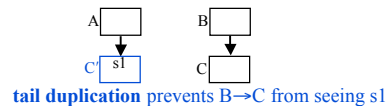
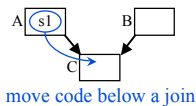
Profile-Guided Optimizations

7

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
- What if s1 might cause an exception?

CS553 Lecture

Profile-Guided Optimizations

8

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)

st1: store r5
 ld2: load r4
 (st1, ld2, 7) 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

CS553 Lecture

Profile-Guided Optimizations

9

Trace Scheduling [Fisher 81] and [Ellis 85]

Basic idea

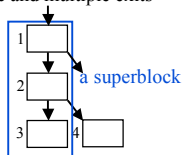
- We want large blocks to create large scheduling windows, but basic blocks are small because branches are frequent
- Create **superblocks** to increase scheduling window
- Use profile information to create good superblocks
- Optimize each superblock independently

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)

```

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
    
```

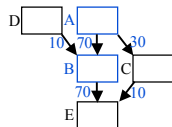
Trace Scheduling (cont)

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**)

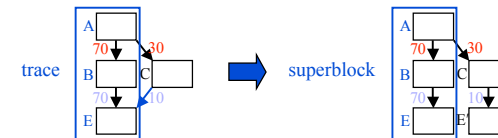


- A **trace** is a maximal sequence of mutual-most-likely blocks that does not contain a back edge
- Each block belongs to exactly one trace

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

CS553 Lecture

Profile-Guided Optimizations

14

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

CS553 Lecture

Profile-Guided Optimizations

15

Speculation based on memory-dependence profiles (example)

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

      trace:
b[i] = "old"
c[i] = a[j]
a[i] = ...
b[i]="new";
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
```

CS553 Lecture

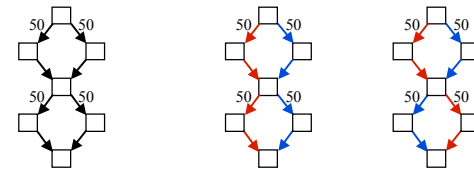
Profile-Guided Optimizations

16

Enhancements to Profile-Guided Code Scheduling

Path profiling [Ball and Larus 96]

- Collect information about entire paths instead of about individual edges



Edge profiles

Path profiles

Path profiles

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

CS553 Lecture

Profile-Guided Optimizations

17

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