Disclaimer: Many slides of this lecture are based on the slides of authors of the textbook from Elsevier. All copyrights reserved.
Profiling Directed Optimization

• Identify frequently executed *hot* code regions
  – Basic blocks
  – Paths
    • Better because it indicates control flow

• Dynamic Profiling
  – Counts execution frequencies
  – Software implemented
  – Hardware implemented
  – Hybrids
Optimization Example

Basic Block A

... 
... 
R3 ← ... 
R7 ← ... 
R1 ← R2 + R3 
Br L1 if R3==0 

Basic Block B

... 
... 
R6 ← R1 + R6 
... 
...

Basic Block C

L1: R1 ← 0 
... 
...

Basic Block A

... 
... 
R3 ← ... 
R7 ← ... 
Br L1 if R3==0 

Compensation code

R1 ← R2 + R3

Basic Block B

... 
... 
R6 ← R1 + R6 
... 
...

Basic Block C

L1: R1 ← 0 
... 
...
Another Optimization Example

Basic Block A

\[ \ldots \]
\[ \ldots \]
\[ R3 \leftarrow \ldots \]
\[ R7 \leftarrow \ldots \]
\[ R1 \leftarrow R2 + R3 \]
\[ \text{BEQ L1 if } R3 == 0 \]

Basic Block B

\[ \ldots \]
\[ R6 \leftarrow R1 + R6 \]
\[ \ldots \]

Basic Block C

\[ \text{L1: } R1 \leftarrow 0 \]
\[ \ldots \]

Superblock

\[ \ldots \]
\[ \ldots \]
\[ R3 \leftarrow \ldots \]
\[ R7 \leftarrow \ldots \]
\[ \text{BNE L2 if } R3 != 0 \]
\[ R1 \leftarrow 0 \]
\[ \ldots \]

Compensation code

\[ R1 \leftarrow R2 + R3 \]

Basic Block B

\[ \text{L2: } \ldots \]
\[ R6 \leftarrow R1 + R6 \]
\[ \ldots \]
Program Behavior

- Many aspects of program behavior are predictable
  - Based on history

R3 ← 100
loop:  R1 ← mem(R2) ; load from memory
       Br found if R1 == -1 ; look for -1
       R2 ← R2 + 4
       R3 ← R3 -1
       Br loop if R3 != 0 ; loop closing branch

found:

- Test for -1 primarily not taken
- Loop closing branch primarily taken
Branch Behavior

- A Conditional Branch is predominantly decided one way
  - Either taken or not taken

For SPEC benchmark suite
Branch Behavior

- Most branches are decided the same way as on previous execution
- Backward conditional branches are mostly taken
  - Forward conditional branches taken less often

For SPEC benchmark suite
Program Behavior

- Some indirect jumps (i.e. target is stored in register) have a single target
  - Others have several targets (e.g. returns)
Program Behavior

- **Predictability extends to data values**
  - Many instructions always produce the same result
Profiling

- Collect statistics about a program as it runs
  - Branches (taken, not taken)
  - Jump targets
  - Data values
- Predictability allows these statistics to be used for optimizations to be used in the future
- Profiling in a VM differs from traditional profiling used for compiler feedback
Conventional Profiling

- Multiple passes through compiler
- Done at program development time
  - Profile overhead is a small issue
- Can be based on global analysis
VM-Based Profiling

- Profile overhead is very important
  - Profile time comes out of execution time
- Limited view of program (no a priori global view)
  - Profile probes cannot be carefully placed
- Program characteristics must be determined as early as possible.

![Diagram of VM-Based Profiling process]

1. Program Binary
2. Interpreter
3. Partial Program Statistics
4. Translator/Optimizer
5. Program Data

Partially "discovered" code
Types of Profiles

- **Block or node profiles**
  - Identify "hot" code blocks

- **Edge profiles**
  - Give a more precise idea of program flow
  - Block profile can be derived from edge profile (not vice versa)
Collecting Profiles

• Instrumentation-based
  – Software probes
    • Slows down program more
    • Requires less total time
  – Hardware probes
    • Less overhead than software
    • Less well-supported in processors
    • Typically event counters

• Sampling based
  – Interrupt at random intervals and take sample
    • Slows down program less
    • Requires longer time to get same amount of data
  – Not useful during interpretation
Sampling

- Set interval counter
- Interrupt when counter hits zero
- Sample PC at that point
- Gives block profile
- Could be modified to give edge profile
Profiling During Interpretation

• Source instructions are accessed as data.

• Interpreter routines are the code that is being executed.

• So: profiling code must be added to the interpreter routines.
### Profiling During Interpretation

**Instruction function list**

```
branch_conditional(inst) {
    BO = extract(inst,25,5);
    BI = extract(inst,20,5);
    displacement = extract(inst,15,14) * 4;
    // code to compute whether branch should be taken
    profile_addr = lookup(PC);
    if (branch_taken) {
        profile_cnt(profile_addr, taken)++;
        PC = PC + displacement;
    } Else {
        profile_cnt(profile_addr, nottaken)++;
        PC = PC + 4;
    }
}
```
Profiling Translated Code

- Software Instrumentation in Stub Code

```
increment edge counter (j)
if (counter (j) > trigger) then invoke optimizer
else branch to target basic block
```

```
increment edge counter (i)
if (counter (i) > trigger) then invoke optimizer
else branch to fallthrough basic block
```
Now that we have profiling data, what can we do with it?
Strategies

• Use our knowledge of control flow to put frequently followed sequences of basic blocks in contiguous memory locations to increase locality.

• Aggregate basic blocks into superblocks/traces/tree groups and optimize them.
Optimization: Improving Locality
Improving Locality: Example

A
Br cond1 == true
B
Br cond2 == false
C
Br cond3 == true
D
Br cond4 == true
E
Br cond5 == true
F
Br cond6 == true
G
Br cond7 == true
H
Improving Locality: Example

- Little locality (spatial or temporal) in cache line that spans blocks E and F
- F seldom used
  - Wasted I-cache space
  - Wasted I-fetch bandwidth
- Heavily used discontiguous code blocks
  - E.g., C and D
  - Still wastes I-fetch bandwidth

|   | Br uncond | F___________ | F___________ | F___________ |
Improving Locality: Rearrange Code

1. Decide on blocks arrangement
2. Update branches accordingly
Improving Locality: Procedure Inlining

- **Partial inlining**
  - Unlike static full inlining
  - Follow dominant flow of control
Improving Locality: Traces

- Proposed by Fisher (Multiflow)
  - Used overall profile/analysis
  - Can have side entrance and side exits.
- Greedy Method
  - Start at hottest block not yet in a trace
  - Follow hottest edges
  - Stop when trace reaches a certain size
  - Stop when a block already in a trace is reached
Traces, contd.

- No redundancy
  - Good for spatial locality
  - Not good for temporal locality

- Typically not used in optimizing VMs
  - Because cannot be easily built on the fly during execution → needs profiling on test data then static building.
Improving Locality: Superblocks

- One entry (no side entrance) multiple exits
- May contain redundant blocks *(tail duplication)*
- More commonly used by dynamic optimizers than traces

No Tail Duplication

With Tail Duplication
Superblock Formation

• **Start Points**
  – When block use reaches a threshold
  – Profile all blocks
  – Profile selected blocks
    • Profile only targets of backward branches (close loops)
    • Profile exits from existing superblocks

• **Continuation**
  – Use hottest edges above a (second) threshold
  – Follow current control path (most recent edge)

• **End Points**
  – Start point of this superblock
  – Start point of some other superblock
  – When a maximum size is reached
  – When no edge above threshold can be found
  – When an indirect jump is reached (depends on whether inlining is enabled)
Tree Groups (Tree Regions)

- Generalization of Superblocks
  - One entrance
  - Several exits
  - Several flows of control
- Good when one branch direction is not dominant
- Larger scope for optimization
Now that we have superbblocks, tree groups, etc. What do we do with them?
Static versus Dynamic Optimization

• With Static Optimization
  – More time for analysis (done offline)
    • Profiling/Opt. overhead does not add to total execution time
    • Can place profile probes more carefully
    • Can analyze results more carefully

• With Dynamic Optimization
  – Often use simpler, less optimal methods
Dynamic Optimization Overview

Original source code

Collect basic blocks using profile information

A

B

C

A

B

C

Intermediate form

Convert to intermediate form; place in buffer

Schedule and optimize

Generate target code

Optimized target code

Add compensation code; place in code cache
Code Scheduling

• Order code for better performance

• An important optimization in many VMs
  – Especially if host platform is in-order issue or VLIW

• We first will consider code movement of specific instruction types
  – Instruction Types:
    • REG: Register updates
      – includes loads
    • MEM: Memory updates
    • BR: Branches and Jumps
    • JOIN: Join points
“Micro” Code Scheduling

• Example Code Sequence

\[
\begin{align*}
R1 & \leftarrow \text{mem}(R6) \quad \text{reg} \\
R2 & \leftarrow \text{mem}(R6 + 4) \quad \text{reg} \\
R3 & \leftarrow R1 + 1 \quad \text{reg} \\
R4 & \leftarrow R1 \ll 2 \quad \text{reg} \\
\text{Br exit; if } R7 &= 0 & \text{br} \\
R7 & \leftarrow R7 + 1 \quad \text{reg} \\
\text{mem (R6)} & \leftarrow R3 \quad \text{mem}
\end{align*}
\]
Moving Code Below Branches

- Generally straightforward
- Compensation is via duplication
Moving Code Above Branches

\[ \begin{align*}
R2 & \leftarrow R1 \ll 2 \\
\text{Br exit; if } R8 == 0 \\
R6 & \leftarrow R7 \times R2 \\
\text{mem (R6)} & \leftarrow R3 \\
R6 & \leftarrow R2 + 2
\end{align*} \]
Moving Code Above Branches

• For reg instructions, “checkpoint”
  – Keep old value live in a temporary register
  – If exit branch is taken, mapped register does not get modified
  – If instruction traps, backup and interpret forward

• Moving store breaks memory state compatibility
  – E.g. what if exit branch is taken?
Conclusions

• Profiling is crucial to ensure acceptable performance for VMs
• The profiling data we gather depends on the type of optimizing we want to do.
• Usually we follow the following steps:
  1. Gather profiling data
  2. Form superbloks
  3. Optimize
• One of the most commonly used form of optimization is code reordering.
• Remember that till now we were working at the ISA level