Dynamic Binary Optimization

- Introduction
- Application profiling
- Optimizing translation blocks
- Compatibility
- Code reordering
- Other code optimizations
Optimization Overview

- Identify frequently executed *hot* code regions
  - basic blocks
  - paths – indicate control flow
  - edges – approximation to paths
- Dynamic profiling
  - count execution frequencies
  - software or hardware implemented
- Form large translation blocks
  - traces and superbloks
- Schedule and optimize large blocks
Optimization Based On Profiling

**Basic Block A**

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

**Basic Block B**

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

**Basic Block C**

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

**Basic Block A**

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

**Compensation code**

\[ R1 \leftarrow R2 + R3 \]

**Basic Block B**

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

**Basic Block C**

\[ \text{L1: } R1 \leftarrow 0 \]
\[ \ldots \]
\[ \ldots \]
Optimization Based On Profiling (2)

Basic Block A
... ...
R3 ← ...
R7 ← ...
R1 ← R2 + R3
BEQ L1 if R3 == 0

Basic Block B
... 
R6 ← R1 + R6
... 

Basic Block C
L1: R1 ← 0
... 

Superblock
... ...
R3 ← ...
R7 ← ...
BNE L2 if R3 != 0
R1 ← 0
... 

Compensation code
R1 ← R2 + R3

Basic Block B
L2: ...
R6 ← R1 + R6
... 
... 

EECS 768 Virtual Machines
Program Behavior

- Many aspects of a program's behavior are predictable
  - branches, data values

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:

- Backward branch primarily taken
- Forward branch mostly not taken
Branch Behavior

- Conditional branch predominantly decided one way
  - either taken or not taken

[Bar chart showing distribution of percent taken for conditional branches]
Branch Behavior (2)

- Most branches decided the same way as on previous execution
  - backward conditional branches are mostly taken
  - forward conditional branches taken less often
Other Program Behavior

- Some indirect jumps have a single target
  - others have several targets (e.g. returns)
- Predictability extends to data values
  - many instructions always produce the same result

![Graph showing the fraction of constant values for different instruction types.](image)
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 in the future
- Profiling in a VM differs from traditional profiling used for compiler feedback
Conventional (Offline) Profiling

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

- Profile overhead is very important
  - profile time part of total execution time
- Limited view of program (no a priori global view)
  - profile probes cannot be carefully placed
Types of Profiles

- **Block or node profiles**
  - identify *hot* code blocks; fewer nodes than edges
- **Edge profiles**
  - more precise idea of program flow
  - block profile can be derived from edge profile
Collecting Profiles

- Instrumentation-based
  - software probes
    - slows down program more
    - requires less total time than sampling
  - 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
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

<table>
<thead>
<tr>
<th>Translated Basic Block</th>
<th>Fall-thru stub</th>
<th>Branch target stub</th>
</tr>
</thead>
</table>

Increment edge counter (i)
If (counter (i) > trigger) then invoke optimizer
Else branch to fall-thru basic block

Increment edge counter (j)
If (counter (j) > trigger) then invoke optimizer
Else branch to target basic block
Sampling

- Set interval counter
- Interrupt when counter hits zero
- Sample PC at that point
- Gives block profile
- Could be modified to give edge profile
Improving Code Locality

- Provide more optimization opportunities.
- *Spatial* locality
  - consecutive memory accesses are adjacent
- *Temporal* locality
  - same memory access is repeated in near future
- Reasons for spatial and temporal locality
  - loops and sequential program flow
Improving Locality: Example

```
A
Br cond1 == true
B
Br cond2 == false
C
Br uncond
D
Br cond3 == true
E
Br uncond
F
G
Br cond4 == true
```

### Diagram

- **A**
  - **B**
    - **C**
      - **D**
      - **E**
    - **F**
  - **G**
    - **B**
      - **C**
      - **D**
    - **E**
  - **30**
  - **70**

- **A**
  - **B**
    - **C**
      - **D**
      - **E**
    - **F**
  - **G**
    - **B**
      - **C**
      - **D**
    - **E**
  - **3**
  - **2**
Improving Locality: Example (2)

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

| E  | Br uncond | F | F | F |
Improving Locality: Rearrange Code

![Diagram of code rearrangement]

- Br cond1 == true
- Br cond2 == false
- Br uncond
- Br cond3 == true
- Br cond4 == true
- Br cond1 == false
- Br cond2 == false
- Br uncond
- Br cond3 == true
- Br cond4 == true
- Br uncond
- Br uncond
Improving Locality: Procedure Inlining

- **Inlining** – duplicate procedure body at call-site

- Partial inlining
  - follow dominant flow of control
  - not practical to find full procedure during dynamic incremental code discovery

- Disadvantages
  - increases code size
  - increases register pressure
Improving Locality: Traces

- Divide program into chunks
  - may contain multiple blocks
- Greedy Method
  - suitable for on-the-fly translation
  - start at hottest block not in trace
  - follow hottest edges
  - stop when trace reaches a certain size
  - stop when a block already in a trace is reached
Improving Locality: Traces (2)

- No redundancy
  - may reduce I-cache pressure
  - good for spatial locality
- Join points sometimes inhibit optimizations.
- Typically not used in optimizing VMs.
Improving Locality: Superblocks

- **Superblock** – One entry, multiple exits
- May contain redundant blocks (tail duplication)
- More commonly used by dynamic optimizers
  - better branch prediction
  - less constraints on optimizations
Superblocks: Example
Optimization Strategy

Collect basic blocks using profile information

Convert to intermediate form; place in buffer

Schedule and optimize

Generate target code

Add compensation code; place in code cache

Original source code

Intermediate form

Optimized target code
Optimization and Compatibility

• Requirements for compatibility
  • isomorphism of user/privilege mode control transfer points
  • isomorphism of guest state at the control transfer points

• Optimizations can affect the visibility of traps
  • reordering instructions may affect where traps occur
  • adding/eliminating instructions may affect if traps occur

• Trap compatibility
  • trap during native execution of source instruction also occurs during emulation of corresponding target instruction
  • trap observed during emulation should also occur in the corresponding source instruction
Optimization and Compatibility (2)

- **Trap compatibility**
  
  **Source**
  
  ```
  ...  
  r4 ← r6 + 1
  r1 ← r2 + r3  → trap?
  r1 ← r4 + r5
  r6 ← r1 * r7
  ```

  **Target**
  
  ```
  ...  
  R4 ← R6 + 1
  R1 ← R4 + R5
  R6 ← R1 * R7
  ```

  - **Memory and register state compatibility**
  
  - consistent program state on guest and native platform at each control transfer point

  **Source**
  
  ```
  ...  
  r1 ← r2 + r3
  r9 ← r1 + r5  ⬤ reschedule
  r6 ← r1 * r7
  r3 ← r6 + 1  
  ...  
  ```

  **Target**
  
  ```
  ...  
  R1 ← R2 + R3
  R6 ← R1 * R7
  R9 ← R1 + R5  → trap?
  R3 ← R6 + 1  
  ```

  **Target with saved reg. state**
  
  ```
  ...  
  R1 ← R2 + R3
  S1 ← R1 * R7
  R9 ← R1 + R5
  R6 ← S1
  R3 ← S1 + 1  
  ```
Code Reordering

• Important aspect of several optimizations
  • especially for pipelined RICS, and VLIW processors
  • reduce pipeline stalls and functional unit latencies
• Primitive instruction reordering issues
  • consider reordering pairs of instructions
  • divide instructions into basic categories
Instruction Categories

- *reg* updates – instructions updating registers
- *memory* updates – instructions updating memory
- *branch* instructions – transfer of control instructions
- *join* point – points where jump/branch enter code sequence (only for traces)

```
...  
R1  →  mem(R6)  reg
R2  →  mem(R6 +4)  reg
R3  →  R1 + 1  reg
R4  →  R1 << 2  reg
Br exit; if R7 == 0  br
R7  →  R7 + 1  reg
mem (R6)  →  R3  mem
```
Moving Instructions Below Branches

- Duplicate compensation code at the exit point.
- Pretty straightforward.
- Works for registers as well as memory state.

```
... R1 ← mem(R6)             R1 ← mem(R6)
R2 ← mem(R6+4)             R2 ← mem(R6+4)
R3 ← R1 + 1               R3 ← R1 + 1
R4 ← R1 << 2              Br exit if R7 == 0
Br exit if R7 == 0
R7 ← R7 + 1               R4 ← R1 << 2
mem(R6) ← R3             R7 ← R7 + 1
mem(R6) ← R3             mem(R6) ← R3
...                        R4 ← R1 << 2
```
Moving Instructions Above Branches

- Use **checkpoint** for moving reg instructions
  - calculate *reg* update in a temporary register
  - if branch taken, real register is unmodified
  - if instruction traps, all register state unmodified

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

\[
\begin{align*}
T1 & \leftarrow R7 \times R2 \\
Br & \text{ exit if } R8 \equiv 0 \\
R6 & \leftarrow T1 \\
\text{mem}(T1) & \leftarrow R3 \\
R6 & \leftarrow R2 + 2
\end{align*}
\]
Moving Instructions Above Branches

- Moving stores above branches breaks memory state compatibility
  - what if exit branch is taken?
  - difficult to replicate memory state!

```plaintext
... 
R2 ← R1 << 2
T1 ← R7 * R2
Br exit if R8 == 0
mem(T1) ← R3
R6 ← R2 + 2
```
Moving Code Above Join Points

- Similar to previous case of branches
- Straightforward, compensation is via duplication

```
R1 ← R1 + 1
R7 ← mem(R6)
R7 ← R7 + 1
...
```

```
R1 ← R1 + 1
R7 ← mem(R6)
R7 ← R7 + 1
...
```
Moving Code Below Join Point

- Should not be done in most cases.
- No way to compensate if the join is taken.
Movement in Straight Line Code

- Can be done via *checkpointing* registers

R1 ← R1 * 3
mem(R6) ← R1
R7 ← R7 << 3
R9 ← R7 + R2

...
Movement in Straight Line Code

- Hoisting stores breaks memory state compatibility
  - unless there is a way to back up store instructions
  - expensive

\[
\begin{align*}
\text{reg} & \quad \text{mem} \\
\text{mem} & \quad \text{mem}
\end{align*}
\]
## Instruction Reordering – Summary

<table>
<thead>
<tr>
<th>first</th>
<th>second</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>reg</strong></td>
<td><strong>mem</strong></td>
<td><strong>br</strong></td>
<td><strong>join</strong></td>
</tr>
<tr>
<td><strong>reg</strong></td>
<td>extend live range of reg instruction</td>
<td>extend live range of reg instruction</td>
<td>extend live range of reg instruction</td>
<td>add compensation code at entrance</td>
</tr>
<tr>
<td><strong>mem</strong></td>
<td>not allowed</td>
<td>not allowed</td>
<td>not allowed</td>
<td>add compensation code at entrance</td>
</tr>
<tr>
<td><strong>br</strong></td>
<td>add compensation code at branch exit</td>
<td>add compensation code at branch exit</td>
<td>Not allowed (changes control flow)</td>
<td>Not allowed (changes control flow)</td>
</tr>
<tr>
<td><strong>join</strong></td>
<td>Not allowed (can only be done in rare cases)</td>
<td>Not allowed (can only be done in rare cases)</td>
<td>Not allowed (changes control flow)</td>
<td>no effect</td>
</tr>
</tbody>
</table>
Optimizations

• Basic local optimizations
  • applied within translation blocks
  • can even optimize statically optimized code further
  • constant propagation, constant folding, strength reduction, dead-assignment elimination, cse, register assignment, etc.
  • compatibility issues verified on a case-by-case basis
• Inter-superblock optimizations
  • go across basic blocks
• ISA-specific optimizations
  • if conversion, instruction alignment
Static Vs Dynamic Optimizations

- Advantages of dynamic optimizations
  - availability of runtime profile information (specialization)
  - ability to see the whole program post-link-time
  - ability to detect and optimize program phases

- Disadvantages
  - compilation time adds to total execution time
    - apply low-overhead conservative optimizations
    - only apply local optimizations
  - high level semantic information may not be available
    - exception, HLL (Java) Vms