Lecture 17
Dynamic Code Optimization

I. Motivation & Background
II. Overview
III. Partial Method Compilation
IV. Partial Dead Code Elimination
V. Partial Escape Analysis
VI. Results

“Partial Method Compilation Using Dynamic Profile Information”,
John Whaley, OOPSLA 01

1) Beyond Static Compilation
   1) Profile-based Compiler: high-level \(\rightarrow\) binary, static
      - Uses (dynamic=runtime) information collected in profiling passes
   2) Interpreter: high-level, emulate, dynamic
   3) Dynamic compilation / code optimization: high-level \(\rightarrow\) binary, dynamic
      - interpreter/compiler hybrid
      - supports cross-module optimization
      - can specialize program using runtime information
        - without separate profiling passes

2) Profile-based Compile-time Optimization
   1. Compile statically
   2. Collect profile (using typical inputs)
   3. Re-compile, using profile

   - Collecting control-flow profiles is relatively inexpensive
   - profiling data dependences, data values, etc., is more costly
   - Limitations of this approach?
### Instrumenting Executable Binaries

1. Compile statically
   - prog1.c

2. Collect profile
   - (using typical inputs)
   - input1

3. Instrumentation
   - How to perform the instrumentation?
   - runme.exe
   - (instrumented)

4. Execution profile
   - execution

- The compiler could insert it directly
- A binary instrumentation tool could modify the executable directly
  - that way, we don’t need to modify the compiler
  - compilers that target the same architecture (e.g., x86) can use the same tool

#### Binary Instrumentation/Optimization Tools

- Unlike typical compilation, the input is a binary (not source code)
- One option: static binary-to-binary rewriting

- Challenges (with the static approach):
  - what about dynamically-linked shared libraries?
  - if our goal is optimization, are we likely to make the code faster?
  - a compiler already tried its best, and it had source code (we don’t)
  - if we are adding instrumentation code, what about time/space overheads?
    - instrumented code might be slow & bloated if we aren’t careful
    - optimization may be needed just to keep these overheads under control

- Bottom line: the purely static approach to binary rewriting is rarely used

---

### 2) (Pure) Interpreter

- One approach to dynamic code execution/analysis is an interpreter
  - basic idea: a software loop that grabs, decodes, and emulates each instruction

```plaintext
while (stillExecuting) {
    inst = readInst(PC);
    instInfo = decodeInst(inst);
    switch (instInfo.opType) {
        case binaryArithmetic: …
        case memoryLoad: …
        …
    }
    PC = nextPC(PC, instInfo);
}
```

- Advantages:
  - also works for dynamic programming languages (e.g., Java)
  - easy to change the way we execute code on-the-fly (SW controls everything)

- Disadvantages:
  - runtime overhead!
    - each dynamic instruction is emulated individually by software

---

### A Sweet Spot?

- Is there a way that we can combine:
  - the flexibility of an interpreter (analyzing and changing code dynamically); and
  - the performance of direct hardware execution?

- Key insights:
  - increase the granularity of interpretation
  - instructions → chunks of code (e.g., procedures, basic blocks)
  - dynamically compile these chunks into directly-executed optimized code
  - store these compiled chunks in a software code cache
  - jump in and out of these cached chunks when appropriate
  - these cached code chunks can be updated!
  - invest more time optimizing code chunks that are clearly hot/important
    - easy to instrument the code, since already rewriting it
    - must balance (dynamic) compilation time with likely benefits
3) Dynamic Compiler

```java
while (stillExecuting) {
    if (codeCompiledAlready(PC)) {
        compileChunkAndInsertInCache(PC);
    }
    jumpIntoCodeCache(PC);
    // compiled chunk returns here when finished
    PC = getNextPC(...);
}
```

- This general approach is widely used:
  - Java virtual machines
  - dynamic binary instrumentation tools (Valgrind, Pin, Dynamo Rio)
  - hardware virtualization

II. Overview of Dynamic Compilation / Code Optimization

- Interpretation/Compilation/Optimization policy decisions
  - Choosing what and how to compile, and how much to optimize
- Collecting runtime information
  - Instrumentation
  - Sampling
- Optimizations exploiting runtime information
  - Focus on frequently-executed code paths

Dynamic Compilation Policy

- \( \Delta T_{\text{total}} = T_{\text{compile}} - (n_{\text{executions}} \times T_{\text{improvement}}) \)
  - If \( \Delta T_{\text{total}} \) is negative, our compilation policy decision was effective.
- We can try to:
  - Reduce \( T_{\text{compile}} \) (faster compile times)
  - Increase \( T_{\text{improvement}} \) (generate better code, but at cost of increasing \( T_{\text{compile}} \))
  - Focus on large \( n_{\text{executions}} \) (compile/optimize hot spots)
- 80/20 rule: Pareto Principle
  - 20% of the work for 80% of the advantage

Components in a Typical Just-In-Time (JIT) Compiler

- Cached chunks of compiled code run at hardware speed
  - returns control to “interpreter” loop when chunk is finished
- Dynamic optimizer uses profiling information to guide code optimization
  - as code becomes hotter, more aggressive optimization is justified
    - replace the old compiled code chunk with a faster version
Latency vs. Throughput

- Tradeoff: startup speed vs. execution performance

<table>
<thead>
<tr>
<th></th>
<th>Startup speed</th>
<th>Execution performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreter</td>
<td>Best</td>
<td>Poor</td>
</tr>
<tr>
<td>‘Quick’ compiler</td>
<td>Fair</td>
<td>Fair</td>
</tr>
<tr>
<td>Optimizing compiler</td>
<td>Poor</td>
<td>Best</td>
</tr>
</tbody>
</table>

Multi-Stage Dynamic Compilation System

Stage 1: interpreted code
- Execution count is the sum of method invocations & back edges executed
- when execution count = t1 (e.g. 2000)

Stage 2: compiled code
- when execution count = t2 (e.g. 25,000)

Stage 3: fully optimized code

Granularity of Compilation: Per Method?

- Methods can be large, especially after inlining
  - Cutting/avoiding inlining too much hurts performance considerably
- Compilation time is proportional to the amount of code being compiled
  - Moreover, many optimizations are not linear
- Even “hot” methods typically contain some code that is rarely/never executed

Example: SpecJVM98 db

```java
void read_db(String fn) {
    int n = 0, act = 0; int b; byte buffer[] = null;
    try {
        FileInputStream sif = new FileInputStream(fn);
        n = sif.getContentLength();
        buffer = new byte[n];
        while ((b = sif.read(buffer, act, n-act))>0) {
            act = act + b;
            if (act != n) {
                /* lots of error handling code, rare */
            }
        } catch (IOException ioe) {
            /* lots of error handling code, rare */
        }
    } finally {
        sif.close();
    }
}
```
Example: SpecJVM98 db

```java
void read_db(String fn) {
    int n = 0, act = 0; int b; byte buffer[] = null;
    try {
        FileInputStream sif = new FileInputStream(fn);
        n = sif.getContentLength();
        buffer = new byte[n];
        while ((b = sif.read(buffer, act, n-act))>0) {
            act = act + b;
        }
        sif.close();
        if (act != n) {
            /* lots of error handling code, rare */
        }
    } catch (IOException ioe) {
        /* lots of error handling code, rare */
    }
}
```

Optimize hot “regions”, not methods

- Optimize only the most frequently executed segments within a method
  - Simple technique:
    - Track execution counts of basic blocks in Stages 1 & 2
    - Any basic block executing in Stage 2 is considered to be not rare
  - Beneficial secondary effect of improving optimization opportunities on the common paths
- No need to profile any basic block executing in Stage 3
  - Already fully optimized

% of Basic Blocks in Methods that are Executed > Threshold Times (hence would get compiled under per-method strategy)

% of Basic Blocks that are Executed > Threshold Times (hence get compiled under per-basic-block strategy)
Dynamic Code Transformations

- Compiling partial methods
- Partial dead code elimination
- Partial escape analysis

III. Partial Method Compilation

1. Based on profile data, determine the set of rare blocks
   - Use code coverage information from the first compiled version

   Goal: Program runs correctly with white blocks compiled and blue blocks interpreted

What are the challenges?
- How to transition from white to blue
- How to transition from blue to white
- How to compile/optimize ignoring blue

Partial Method Compilation

2. Perform live variable analysis
   - Determine the set of live variables at rare block entry points

Partial Method Compilation

3. Redirect the control flow edges that targeted rare blocks, and remove the rare blocks

Once branch to interpreter, never come back to compiled (no blue-to-white transitions)
Partial Method Compilation

4. Perform compilation normally
   - Analyses treat the interpreter transfer point as an unanalyzable method call

Partial Method Compilation

5. Record a map for each interpreter transfer point
   - In code generation, generate a map that specifies the location, in registers or memory, of each of the live variables
   - Maps are typically < 100 bytes

IV. Partial Dead Code Elimination

• Move computation that is only live on a rare path into the rare block, saving computation in the common case

Partial Dead Code Example

```java
x = 0;
if (rare branch 1) {
    ...
    z = x + y;
    ...
}
if (rare branch 2) {
    ...
    a = x + z;
    ...
}
if (rare branch 1) {
    x = 0;
    ...
    z = x + y;
    ...
}
if (rare branch 2) {
    x = 0;
    ...
    a = x + z;
    ...
}
```
V. Escape Analysis

- Escape analysis finds objects that do not escape a method or a thread
  - "Captured" by method:
    - can be allocated on the stack or in registers
  - "Captured" by thread:
    - can avoid synchronization operations
- All Java objects are normally heap allocated, so this is a big win

Partial Escape Analysis

- Stack allocate objects that don’t escape in the common blocks
- Eliminate synchronization on objects that don’t escape the common blocks
- If a branch to a rare block is taken:
  - Copy stack-allocated objects to the heap and update pointers
  - Reapply eliminated synchronizations

Summary: Beyond Static Compilation

1) Profile-based Compiler: high-level $\rightarrow$ binary, static
   - Uses (dynamic=runtime) information collected in profiling passes
2) Interpreter: high-level, emulate, dynamic
3) Dynamic compilation / code optimization: high-level $\rightarrow$ binary, dynamic
   - interpreter/compiler hybrid
   - supports cross-module optimization
   - can specialize program using runtime information
     - without separate profiling passes
     - for what’s hot on this particular run
Looking Ahead

- Friday: No class

- Monday & Wednesday: “Recent Research on Optimization”
  - Student-led discussions, in groups of 2, with 20 minutes/group
  - Read 3 papers on a topic, and lead a discussion in class
  - See “Discussion Leads” tab of course web page for topics, sign-up sheet, instructions

- Spring Break

- Monday March 14
  - Homework #3 due
  - Meetings to discuss project proposal ideas