Lecture 17
Dynamic Code Optimization

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

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

Stadler et al., “Partial Escape Analysis and Scalar Replacement for Java,” CGO’14
I. 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
1) Dynamic Profiling Can Improve Compile-time Optimizations

- Understanding common dynamic behaviors may help guide optimizations
  - e.g., control flow, data dependences, input values

```c
void foo(int A, int B) {
    ...
    while (...) {
        if (A > B)
            *p = 0;
        C = val[i] + D;
        E += C - B;
        ...
    }
}
```

- Profile-based compile-time optimizations
  - e.g., speculative scheduling, cache optimizations, code specialization

How often is this condition true?
How often does *p == val[i]?
Is this loop invariant?

What are typical values of A, B?
Profile-Based Compile-time Optimization

1. Compile statically
   - prog1.c
   - compiler
   - runme.exe

2. Collect profile (using typical inputs)
   - input1
   - runme.exe (instrumented)
   - execution profile

3. Re-compile, using profile
   - prog1.c
   - compiler
   - runme_v2.exe

- Collecting control-flow profiles is relatively inexpensive
  - profiling data dependences, data values, etc., is more costly
- Limitations of this approach?
  - e.g., need to get typical inputs
Instrumenting Executable Binaries

1. Compile statically

   prog1.c

   compiler

   runme.exe

2. Collect profile
   (using typical inputs)

   input1

   runme.exe
   (instrumented)

   execution profile

   binary instrumentation tool

   How to perform the instrumentation?

1. The compiler could insert it directly
2. 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

\[
\text{runme.exe} \rightarrow \text{tool} \rightarrow \text{runme_modified.exe}
\]

• 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

```java
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 $\rightarrow$ 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

- In the simple dynamic compiler shown above, all code is compiled
  - In practice, can choose to compile only when expected benefits exceed costs
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
- Cache manager typically discards cold chunks (but could store in secondary structure)
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
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

when execution count = t1 (e.g. 2000)

Stage 2:

compiled code

when execution count = t2 (e.g. 25,000)

Stage 3:

fully optimized code

Execution count is the sum of method invocations & back edges executed
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;
        }
        sif.close();
        if (act != n) {
            /* lots of error handling code, rare */
        }
    } catch (IOException ioe) {
        /* lots of error handling code, rare */
    }
}
```
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 */
    }
}
```

Lots of rare code!
Optimize hot “code paths”, not entire methods

- Optimize only the most frequently executed code paths 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)

 execution threshold

% of basic blocks compiled

Linpack
JavaCUP
JavaLEX
SwingSet
check
compress
db
javac
mpegaud
mtrt
jack
% of Basic Blocks that are Executed > Threshold Times (hence get compiled under per-basic-block strategy)

![Graph showing the percentage of basic blocks executed versus execution threshold for various programs.](image-url)
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

![Diagram showing live variables at entry points]

live: x,y,z
3. Redirect the control flow edges that targeted rare blocks, and remove the rare blocks.
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
   – Used to reconstruct the interpreter state

Deoptimization

\[ \text{live: } x, y, z \]

\[
\begin{array}{l}
  x: \text{sp - 4} \\
  y: \text{r1} \\
  z: \text{sp - 8}
\end{array}
\]
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

\[
x = 0; \\
\text{if (rare branch 1)} \{ \\
\quad \ldots \\
\quad z = x + y; \\
\quad \ldots \\
\} \\
\text{if (rare branch 2)} \{ \\
\quad \ldots \\
\quad a = x + z; \\
\quad \ldots \\
\}
\]

May in fact **undo** an optimization done by the compiler (that did not know branch was rare)
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, avoiding heap allocation
    • scalar replacement: replace the object’s fields with local variables
  – “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
Oracle HotSpot JVM and Graal Dynamic Compiler

Figure 1: Overview of HotSpot and Graal.
Dynamic Optimizations in HotSpot JVM

- compiler tactics
  - delayed compilation
  - tiered compilation
  - on-stack replacement
  - delayed reoptimization
  - program dependence graph rep.
  - static single assignment rep.
- proof-based techniques
  - exact type inference
  - memory value inference
  - memory value tracking
  - constant folding
  - reassociation
  - operator strength reduction
  - null check elimination
  - type test strength reduction
  - type test elimination
  - algebraic simplification
  - common subexpression elimination
  - integer range typing
- flow-sensitive rewrites
  - conditional constant propagation
  - dominating test detection
  - flow-carried type narrowing
  - dead code elimination
- language-specific techniques
  - class hierarchy analysis
  - devirtualization
  - symbolic constant propagation
  - autobox elimination
  - escape analysis
  - lock elision
  - lock fusion
  - de-reflection
- speculative (profile-based) techniques
  - optimistic nullness assertions
  - optimistic type assertions
  - optimistic type strengthening
  - optimistic array length strengthening
  - undertaken branch pruning
  - optimistic N-morphic inlining
  - branch frequency prediction
  - call frequency prediction
- memory and placement transformation
  - expression hoisting
  - expression sinking
  - redundant store elimination
  - adjacent store fusion
  - card-mark elimination
  - merge-point splitting
- loop transformations
  - loop unrolling
  - loop peeling
  - safepoint elimination
  - iteration range splitting
  - range check elimination
  - loop vectorization
- global code shaping
  - inlining (graph integration)
  - global code motion
  - heat-based code layout
  - switch balancing
  - throw inlining
- control flow graph transformation
  - local code scheduling
  - local code bundling
  - delay slot filling
  - graph-coloring register allocation
  - linear scan register allocation
  - live range splitting
  - copy coalescing
  - constant splitting
  - copy removal
  - address mode matching
  - instruction peepholing
  - DFA-based code generator
HotSpot JVM and Graal Dynamic Compiler

- VM init: JVM invoked
- App init: Java main() invoked
- App active (warmup): First conscious action from application
- App active (steady): Reached semi-optimal performance
- Shutdown: Shutdown requested

Application (compiled)

Author: Aleksey Shipilev

Carnegie Mellon
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
Today’s Class: Dynamic Code Optimization

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

Friday’s Class: Student Presentations 1

• Portability
• Dynamic Optimization