Lecture 25

Dynamic Code Optimization

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

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

(Slide content courtesy of John Whaley & Monica Lam.)

Scenario #1: Better Information for Offline Optimization

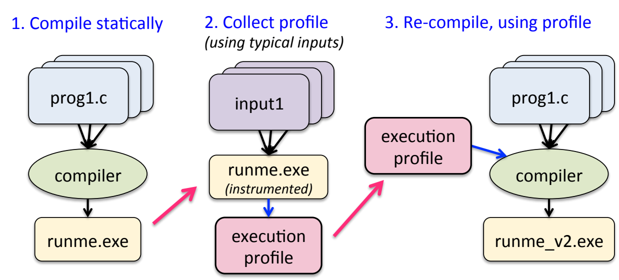
- Understanding common dynamic behaviors may help guide optimizations
  - e.g., control flow, data dependences, input values

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

What are typical values of A, B?
How often is this condition true?
How often does `*p == val[i]`?
Is this loop invariant?

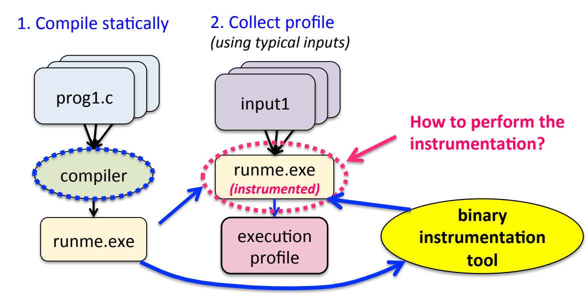
- Useful for speculative scheduling, cache optimizations, code specialization, etc.

Profile-Based Optimization



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

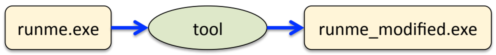
Instrumenting Executable Binaries



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



- 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 and 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

Another Extreme: The Interpreter Approach

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

```
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?

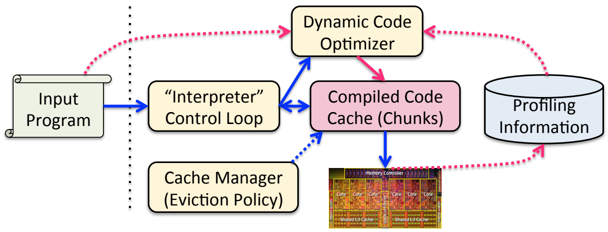
- Is there a way that we can combine:
  - the flexibility of an interpreter (analyzing and changing code dynamically); and
  - the performance of direct hardware execution?
- Key insights:
  - increase the granularity of interpretation
    - ~~instructions~~ → chunks of code (e.g., procedures, basic 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

Main Loop of Chunk-Based Dynamic Optimizer

```
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

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

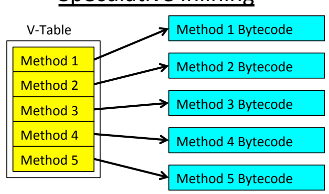


- Cached chunks of compiled code run at hardware speed
  - returns control to "interpreter" loop when chunk is finished
- Dynamic optimizer uses profiling information to guide code optimization
  - as code becomes hotter, more aggressive optimization is justified
    - → replace the old compiled code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

II. Overview of Dynamic Compilation

- Interpretation/Compilation policy decisions
  - Choosing what and how to compile
- Collecting runtime information
  - Instrumentation
  - Sampling
- Exploiting runtime information
  - frequently-executed code paths

Speculative Inlining



- Virtual call sites are deadly
  - kill optimization opportunities
  - virtual dispatch (via indirect jumps) is expensive on modern CPUs
  - very common in object-oriented code
- Speculatively inline the most likely call target (based on profile, class hierarchy)
  - works well since many virtual call sites have only one actual target

III. Compilation Policy

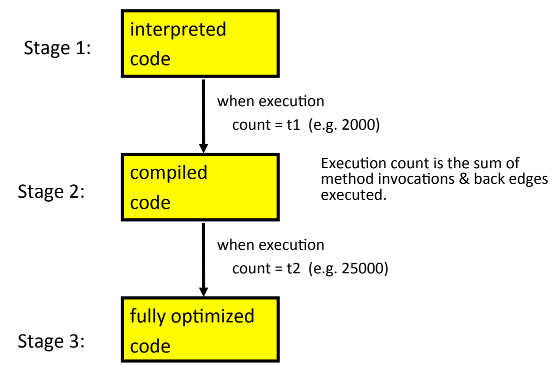
- $\Delta T_{total} = T_{compile} - (n_{executions} * T_{improvement})$
  - If ΔT_{total} is negative, our compilation policy decision was effective.
- We can try to:
  - Reduce $T_{compile}$ (faster compile times)
  - Increase $T_{improvement}$ (generate better code)
  - Focus on large $n_{executions}$ (compile 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

	Startup speed	Execution performance
Interpreter	Best	Poor
‘Quick’ compiler	Fair	Fair
Optimizing compiler	Poor	Best

Multi-Stage Dynamic Compilation System



Granularity of Compilation

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

Example: SpecJVM db

Hot loop

```
void read_db(String fn) {
  int n = 0, act = 0; byte buffer[] = null;
  try {
    FileInputStream sif = new FileInputStream(fn);
    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: SpecJVM db

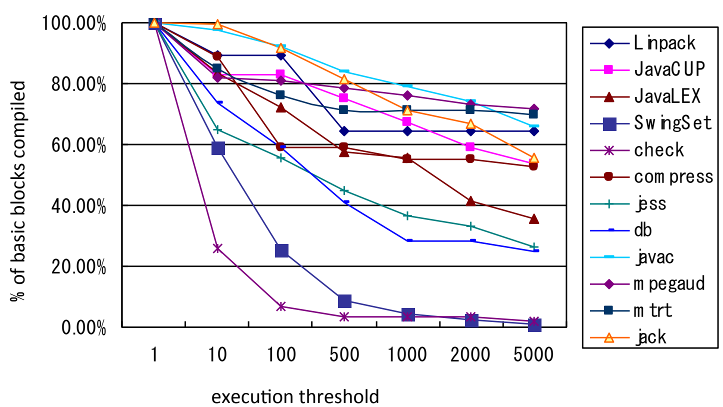
```
void read_db(String fn) {
  int n = 0, act = 0; byte buffer[] = null;
  try {
    FileInputStream sif = new FileInputStream(fn);
    buffer = new byte[n];
    while ((b = sif.read(buffer, act, n-act))>0) {
      act = act + b;
    }
    sif.close();
    if (act != n) {
      /* lots of error handling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

Lots of rare code!

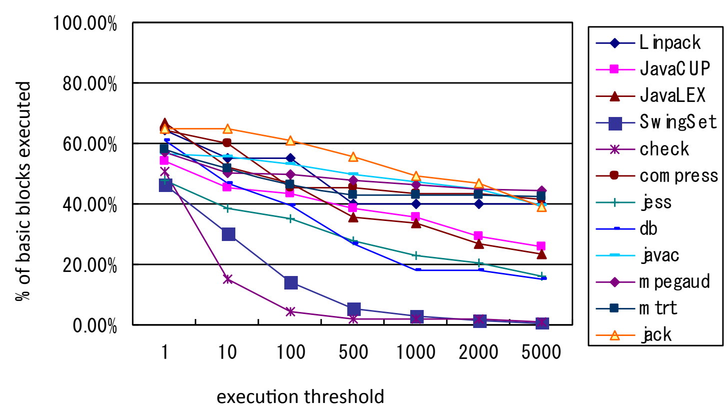
Optimize hot “regions”, not methods

- Optimize only the most frequently executed segments within a method.
  - Simple technique:
    - any basic block executed during Stage 2 is considered to be hot.
- Beneficial secondary effect of improving optimization opportunities on the common paths.

Method-at-a-Time Strategy



Actual Basic Blocks Executed

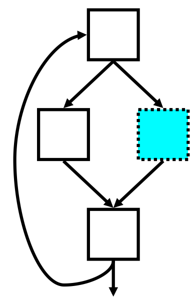


Dynamic Code Transformations

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

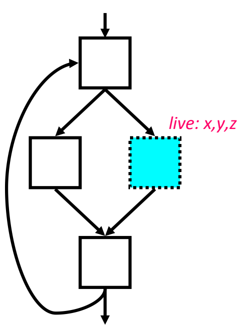
IV. Partial Method Compilation

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



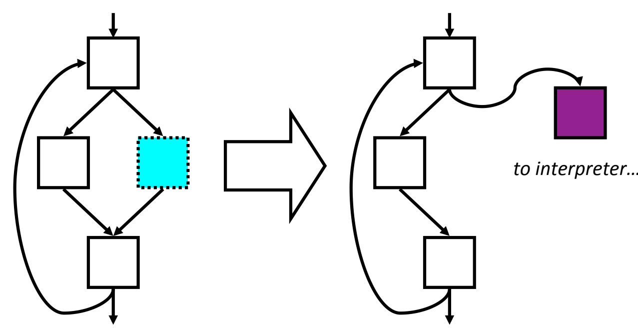
Partial Method Compilation

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



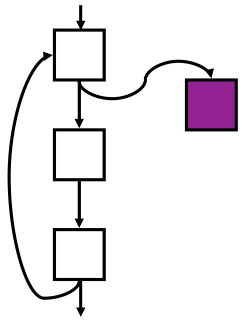
Partial Method Compilation

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



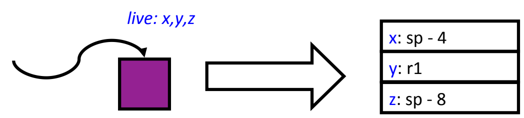
Partial Method Compilation

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



V. 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;
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;
    ...
}
```

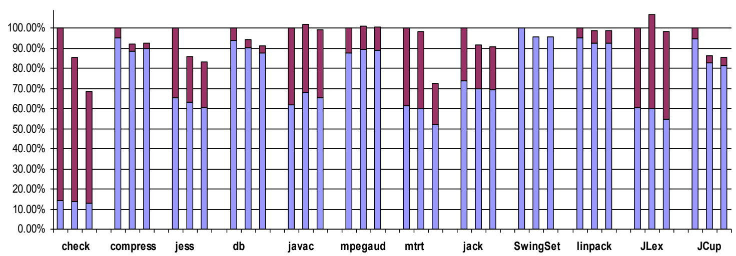
IV. Escape Analysis

- 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.

Escape Analysis: Optimizations

- 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.

VII. Run Time Improvement



First bar: original (Whole method opt)
Second bar: Partial Method Comp (PMC)
Third bar: PMC + opts
Bottom bar: Execution time if code was compiled/opt. from the beginning