

Lecture 9

Dynamic Compilation

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

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

I. Goals of This Lecture

- Beyond static compilation
- Example of a complete system
- Use of data flow techniques in a new context
- Experimental approach

Static/Dynamic, HL/binary

- Compiler: high-level → binary, static
- Interpreter: high-level, emulate, dynamic
- Dynamic compilation: high-level → binary, dynamic
 - machine-independent, dynamic loading
 - cross-module optimization
 - Specialize program using runtime information (without profiling)
- Binary translator: mostly dynamic
 - Run “as-is”
 - Software migration (x86 → alpha, sun, transmeta;
68000 → powerPC → x86)
 - Virtualization (make hardware virtualizable)
 - Dynamic optimization (Dynamo Rio)
 - Security (execute out of code in a cache that is “protected”)

Closed-world vs. Open-world

- Closed-world assumption (most static compilers)
 - All code is available a priori for analysis and compilation.
- Open-world assumption (most dynamic compilers)
 - Code is not available; arbitrary code can be loaded at run time.
- Open-world assumption precludes many optimization opportunities.
 - Solution: Optimistically assume the best case, but provide a way out if necessary.

II. Overview in 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 is expensive on modern CPUs
 - Very common in object-oriented code
- Speculatively inline the most likely call target based on class hierarchy or profile information.
 - Many virtual call sites have only one target, so this technique is very effective in practice.

III. Compilation Policy

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

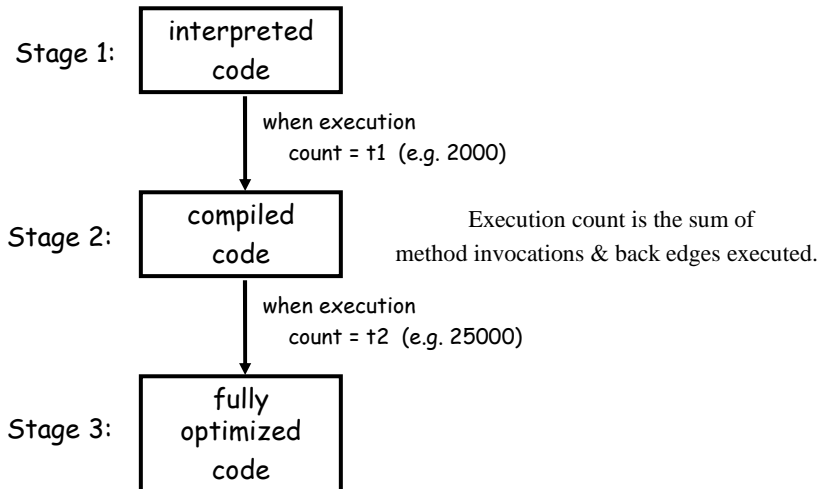
Interpreter + Compiler Approach

- Interpreter plus optimizing compiler
 - Interpret methods by default.
 - When execution count hits some threshold, dynamically compile the method.
- Problems:
 - Interpreter is 10x to 100x slower than compiled code.
 - How to transition long-running methods?

Compile-Only Approach

- Simple compiler plus optimizing compiler (Jalapeño, JUDO, Microsoft)
 - Initially compile each method using a simple compiler.
 - Recompile frequently-executed methods with optimizing compiler.
- Problems:
 - Code from simple compiler is still too slow. (30% to 100% slower than optimizing)
 - Memory footprint problems.

Multi-Stage Dynamic Compilation System



Granularity of Compilation

- Compilation takes time 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 or never executed.

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];
Hot   while ((b = sif.read(buffer, act, n-act))>0) {
loop  →   act = act + b;
        }
        sif.close();
        if (act != n) {
            /* lots of error handling code, rare */
        }
    } catch (IOException ioe) {
        /* lots of error handling code, rare */
    }
}
```

Advanced Compilers L9. Dynamic Compilation

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!

Advanced Compilers L9. Dynamic Compilation

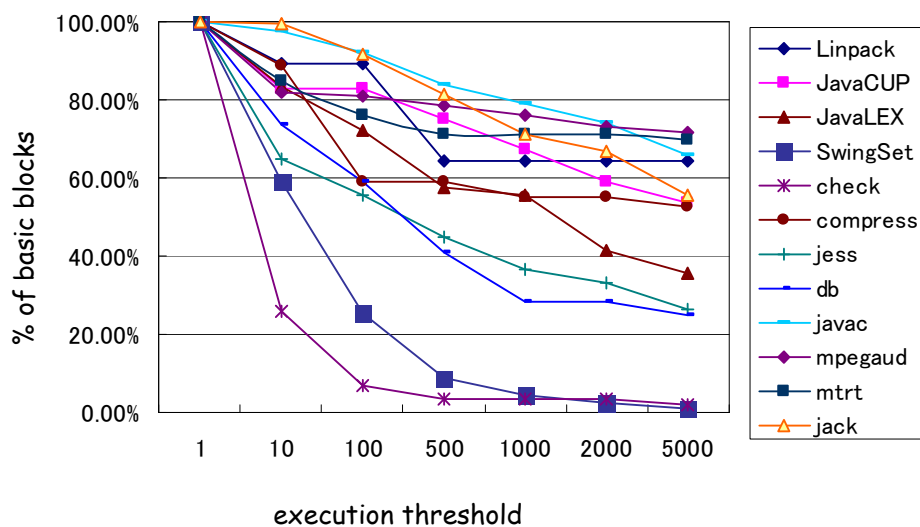
Hot “regions”, not methods

- Compile only the most frequently executed segments within a method.
 1. Collect basic block level profile data for hot methods.
 2. Recompile using profile data, replacing rare code entry points with branches into the interpreter.
- Beneficial secondary effect of improving optimization opportunities on the common paths.

Advanced Compilers

L9. Dynamic Compilation

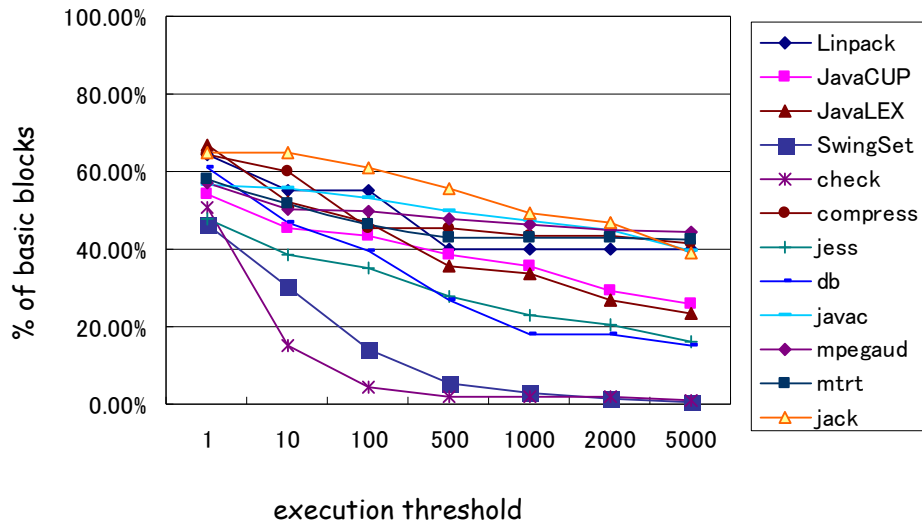
Method-at-a-time Strategy



Advanced Compilers

L9. Dynamic Compilation

Actual Basic Blocks Executed



Advanced Compilers

L9. Dynamic Compilation

Identifying Rare Code

- Simple technique: any basic block executed during Stage 2 is said to be hot.
- Effectively ignores initialization.
- Add instrumentation to the targets of conditional forward branches.
- Enable/disable profiling is implicitly handled by stage transitions.

Advanced Compilers

L9. Dynamic Compilation

Dynamic Code Transformations

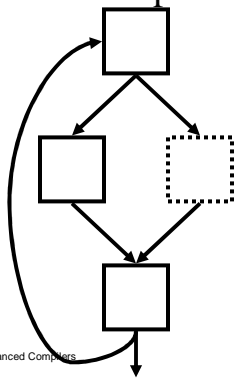
- Compiling partial methods
- Partial dead code elimination
- Escape analysis

Advanced Compilers

L9. Dynamic Compilation

IV. Partial Method Compilation

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



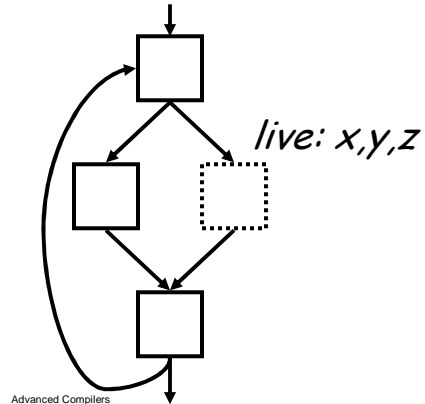
Advanced Compilers

L9. Dynamic Compilation

Partial Method Compilation

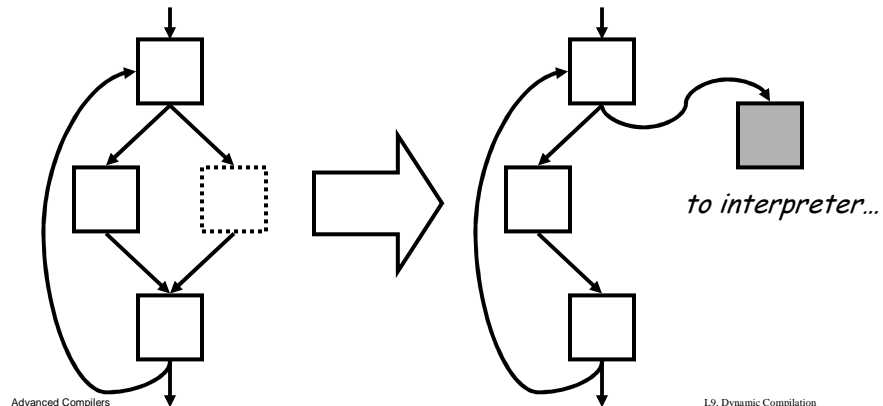
2. Perform live variable analysis.

- Determine the set of live variables at rare block entry points.



Partial Method Compilation

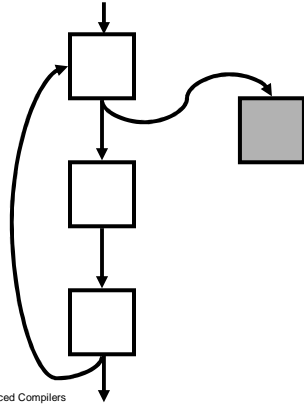
- ### 3. Redirect the control flow edges that targeted rare blocks, and remove the rare blocks.



Partial Method Compilation

4. Perform compilation normally.

- Analyses treat the interpreter transfer point as an unanalyzable method call.

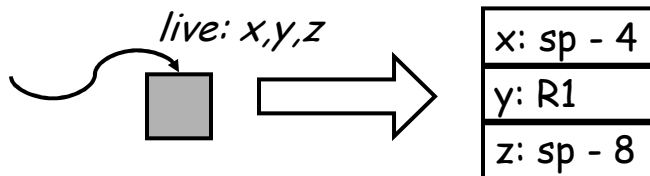


L9. Dynamic Compilation

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



Advanced Compilers

L9. Dynamic Compilation

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.

Advanced Compilers

L9: Dynamic Compilation

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;
    ...
}
```

Advanced Compilers

L9: Dynamic Compilation

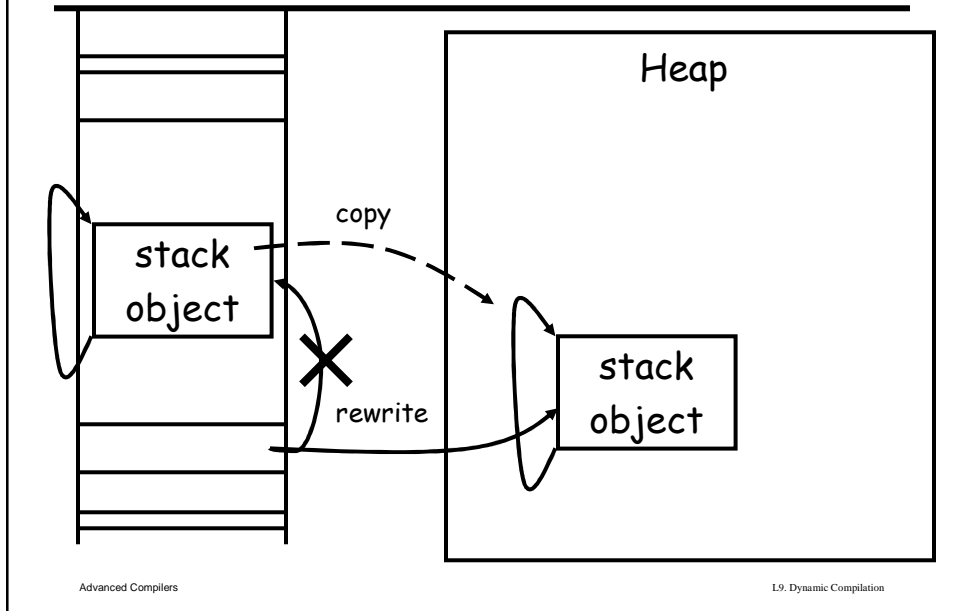
VI. Escape Analysis

- 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.

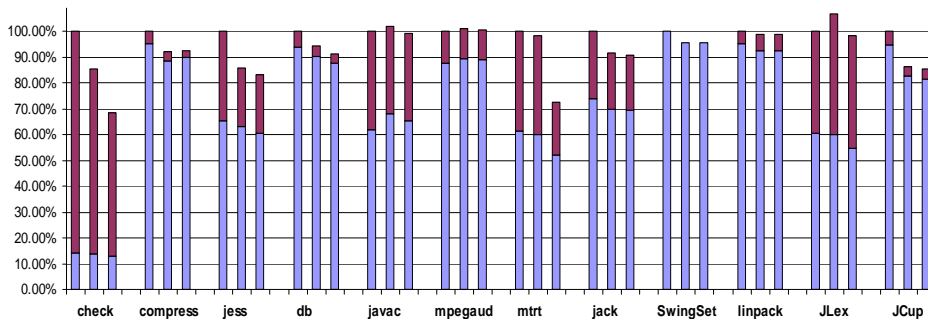
Escape Analysis

- 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.

Copying from Stack to Heap



VII. Run Time Improvement



First bar: original (Whole method opt)
 Second bar: Partial Method Comp (PMC)
 Third bar: PMC + opts
 Bottom bar: fully optimized

Summary

- **Dynamic compilation**
 - Runs as is: No accessibility to source program statically
 - Compilation time counted in the program performance
 - Use run-time information
- **Multi-stage Compilation**
 - Pareto Principle: compile where it makes a difference.
- **Partial method compilation: reduce compile time by reducing granularity of compilation.**
 - Partial dead code elimination
 - Escape analysis