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

M. Lam & J. Whaley

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

- 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)

High-Level/Binary

- Binary translator:
  Binary-binary; 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

- $\Delta T_{total} = T_{compile} - (n_{executions} * T_{improvement})$
  - 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

Stage 1: **interpreted code**

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

Stage 2: **compiled code**

↓ when execution count = t2 (e.g. 25000)

Stage 3: **fully optimized code**

Execution count is the sum of method invocations & back edges executed.

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

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

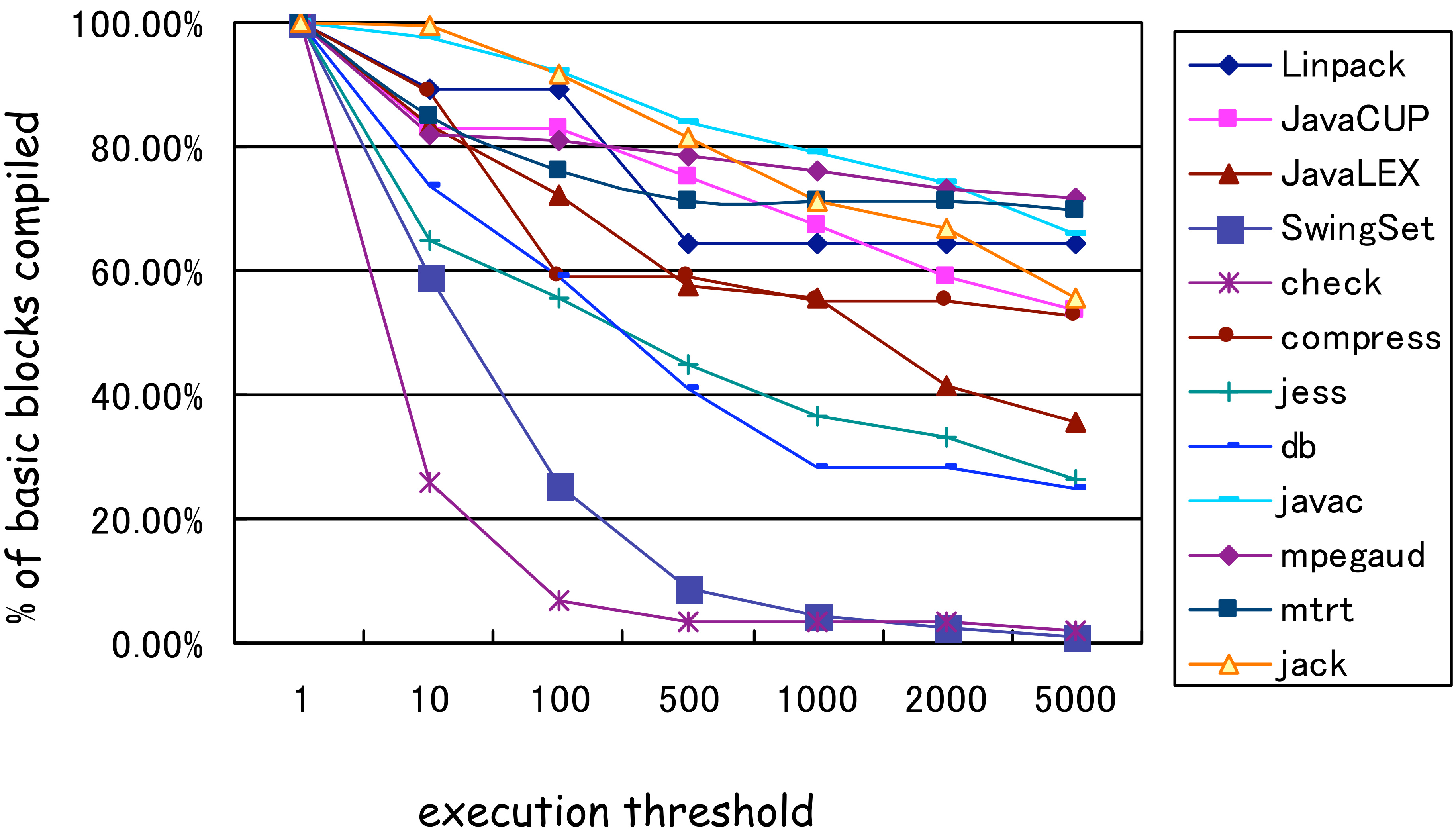
```
void read_db(String fn) {
  int n = 0, act = 0; byte buffer[] = null;
  try {
    FileInputStream sif = new FileInputStream(fn);
    buffer = new byte[n];
    while ((b = sif.read(buffer, act, n-act))>0) {
      act = act + b;
    }
    sif.close();
    if (act != n) {
      /* lots of error handling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

Lots of rare code!

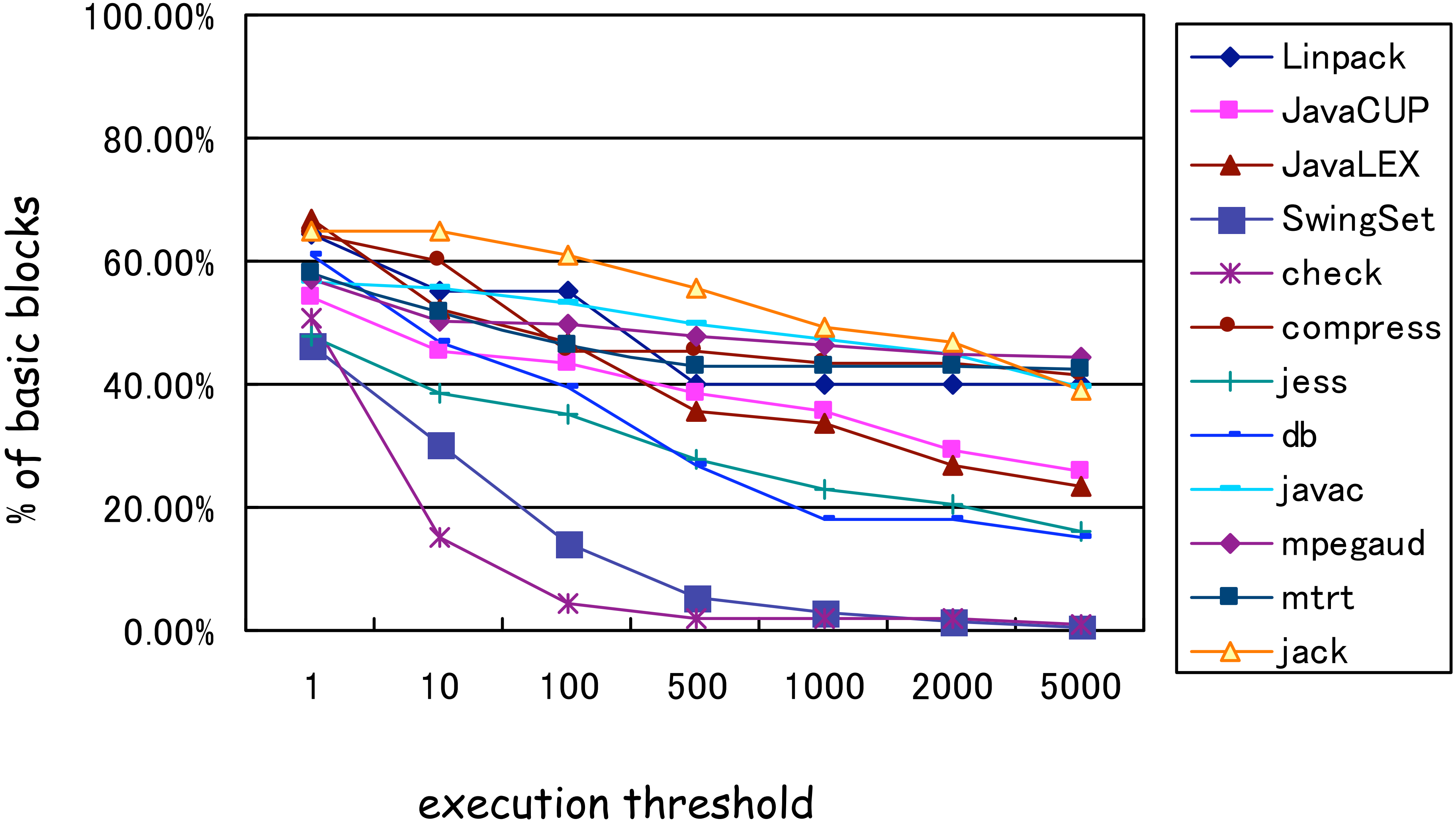
Optimize hot “regions”, not methods

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

Method-at-a-time Strategy



Actual Basic Blocks Executed

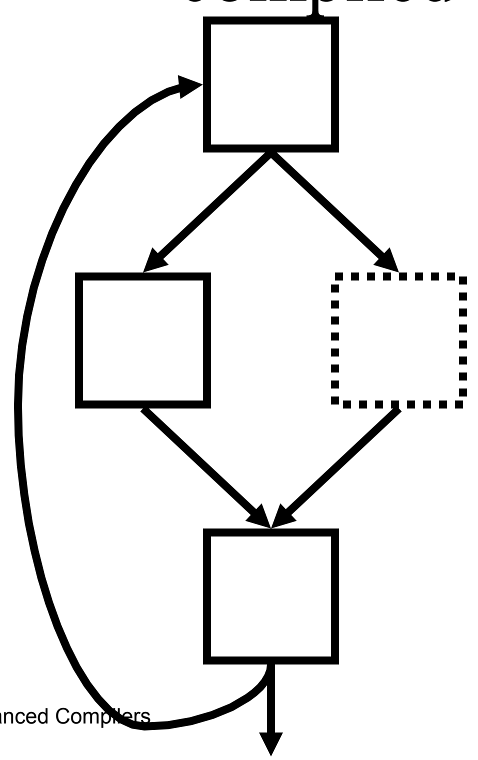


Dynamic Code Transformations

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

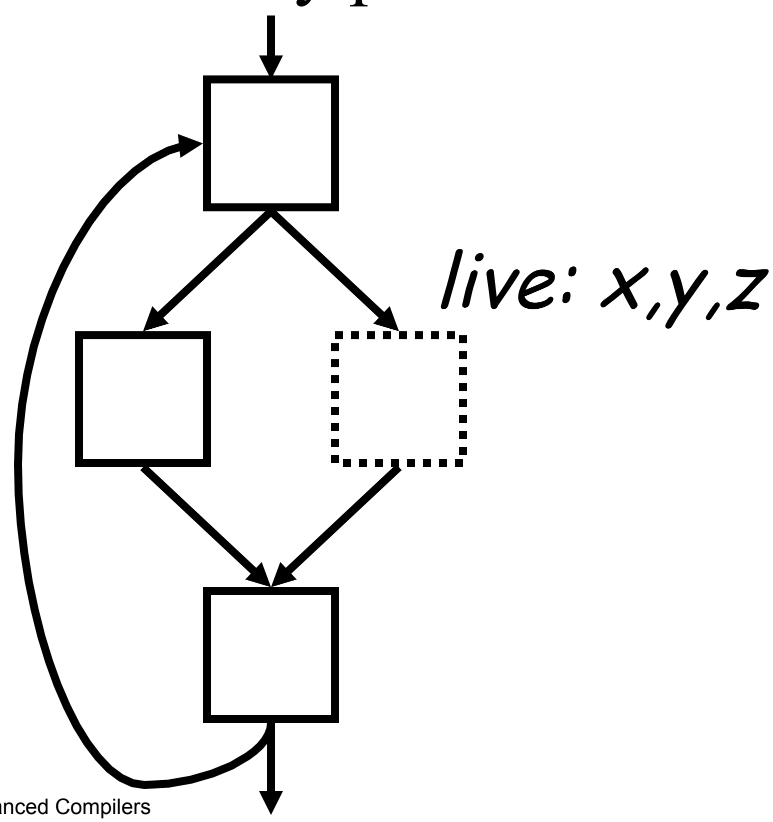
IV. Partial Method Compilation

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



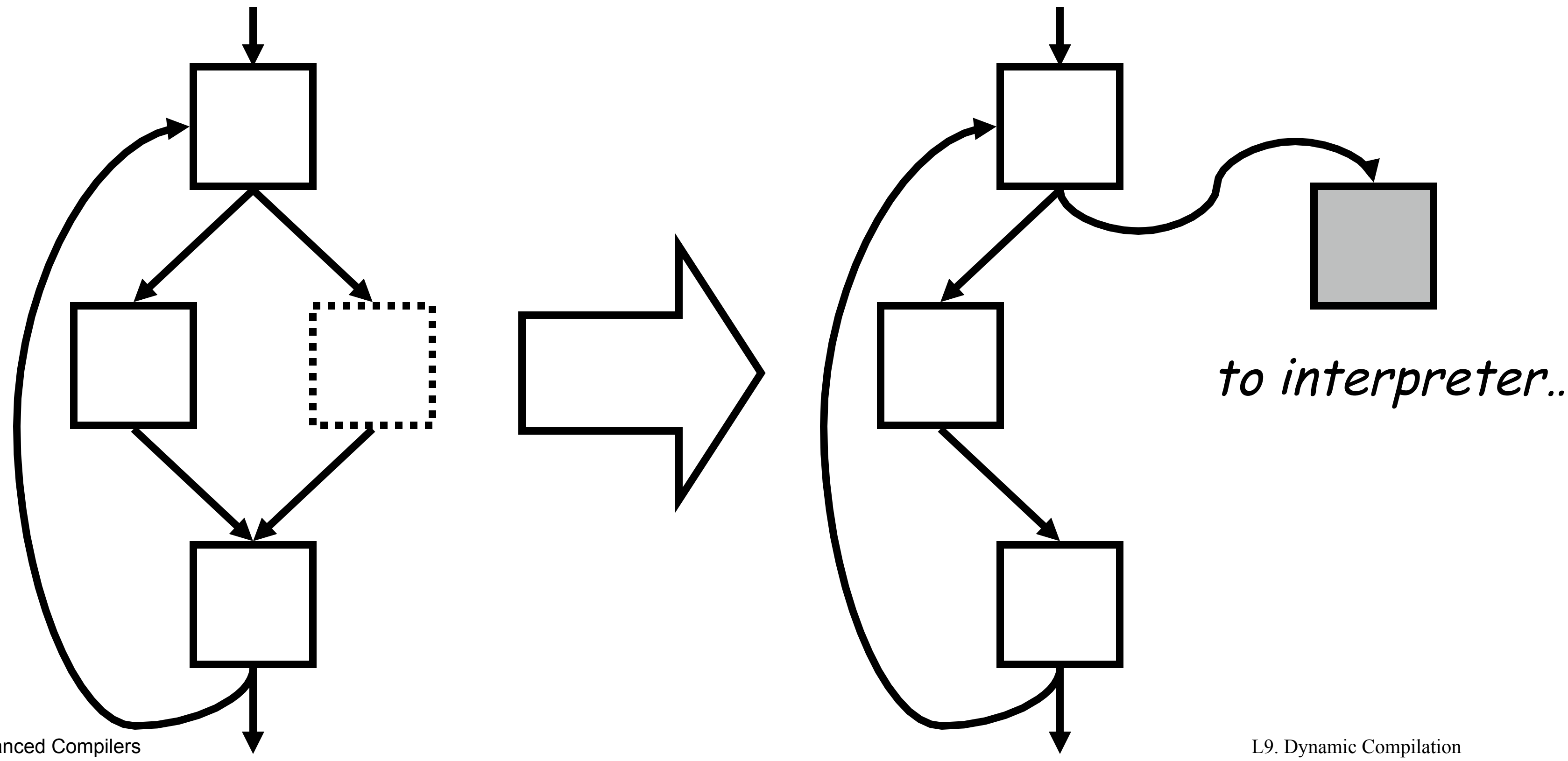
Partial Method Compilation

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



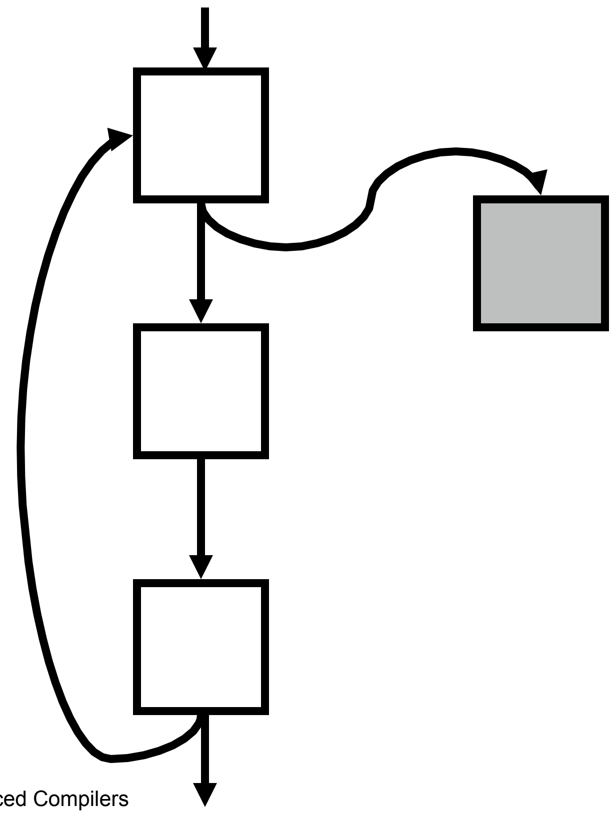
Partial Method Compilation

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



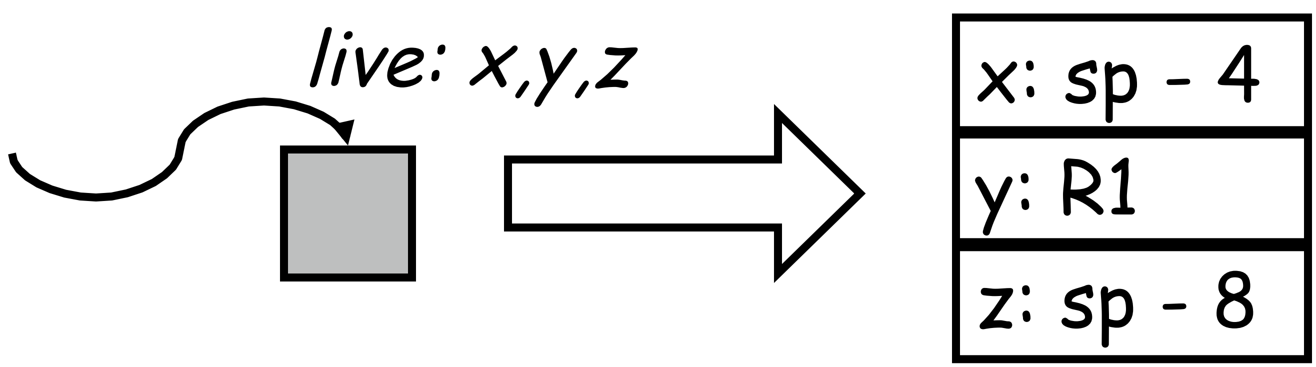
Partial Method Compilation

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;
    ...
}
```

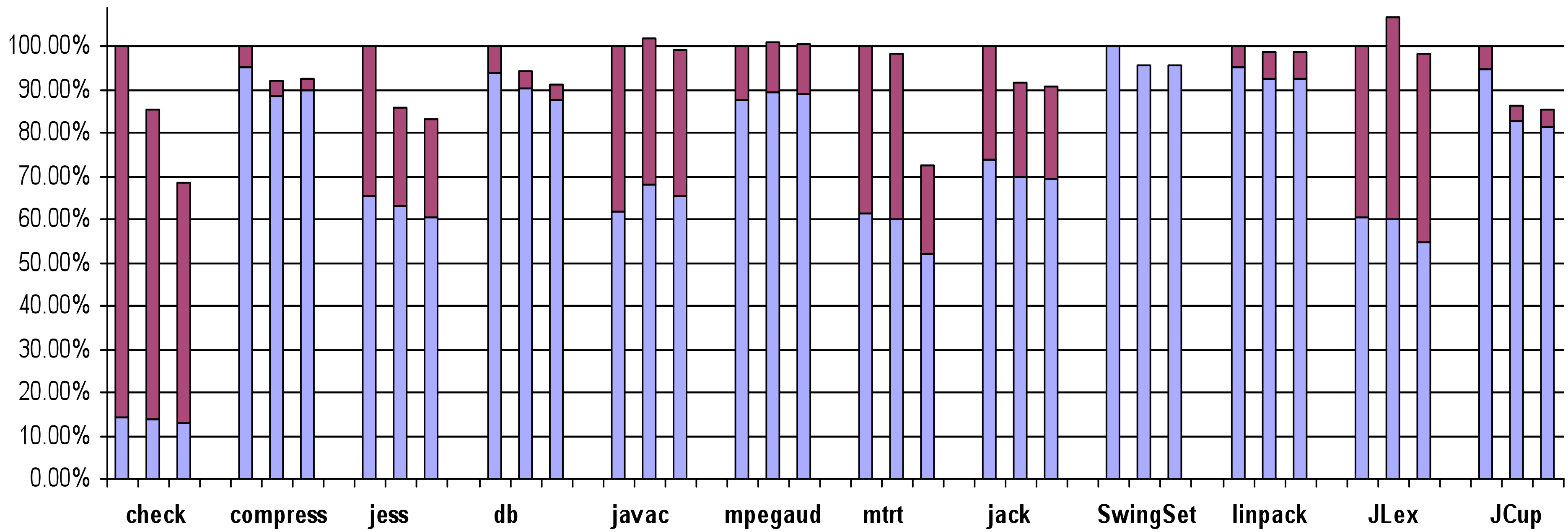
VI. Escape Analysis

- 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.

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