Lecture 14

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

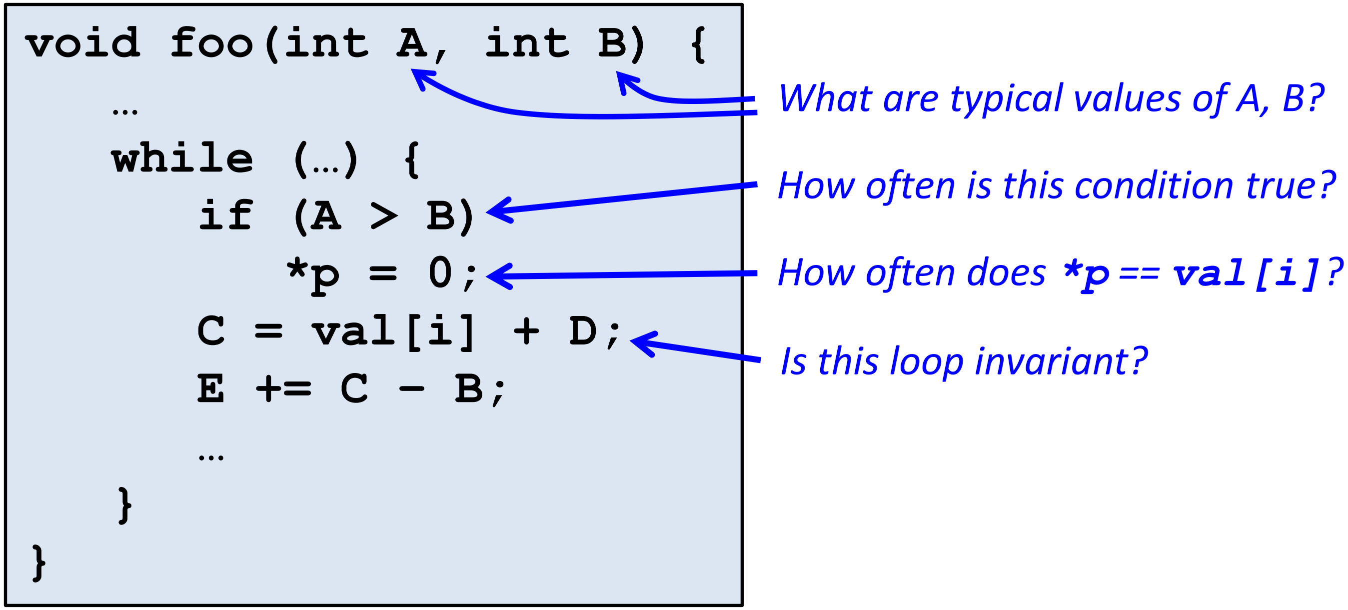
1) Profile-based Compiler: high-level → binary, static
   - Uses (dynamic=runtime) information collected in profiling passes

2) Interpreter: high-level, emulate, dynamic

3) Dynamic compilation / code optimization: high-level → 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



```
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?

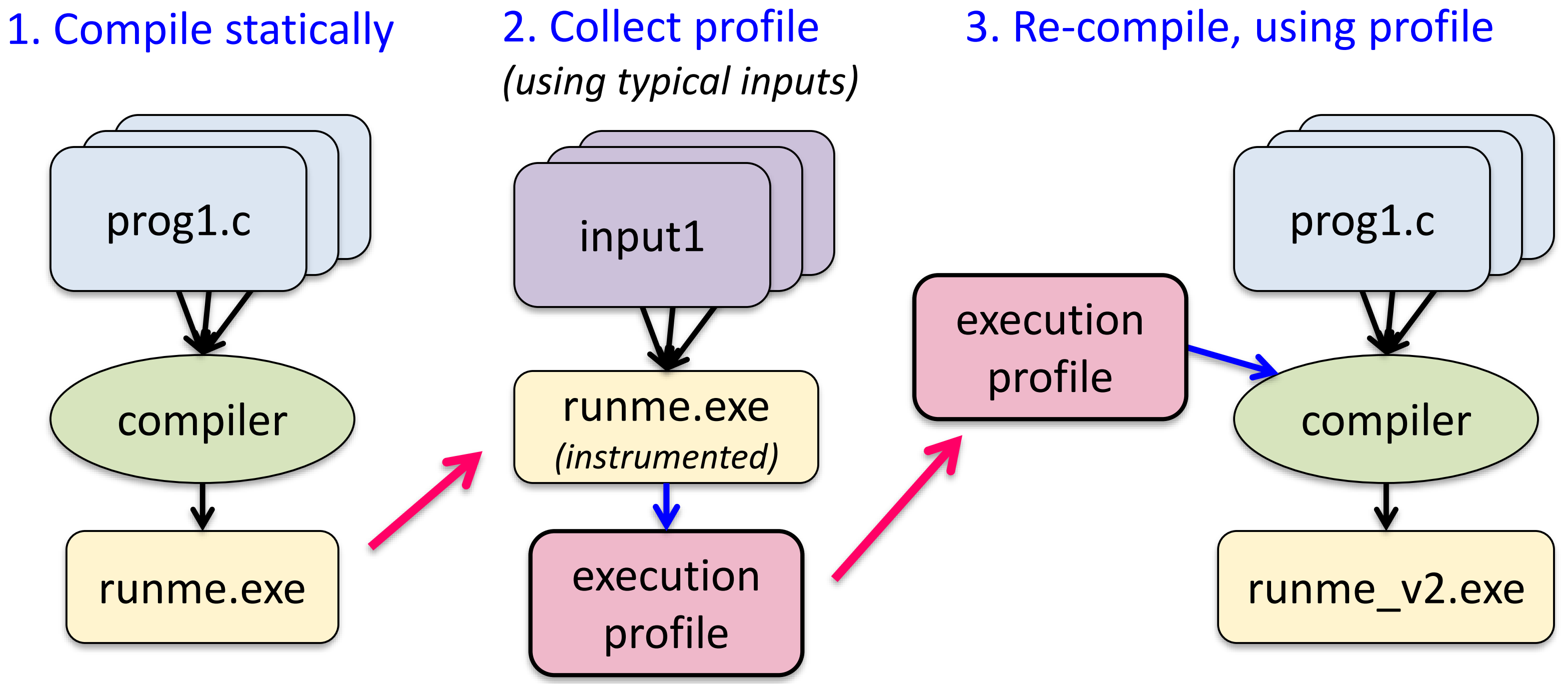
How often is this condition true?

How often does ***`*p`*** *==* ***`val[i]`****?*

Is this loop invariant?

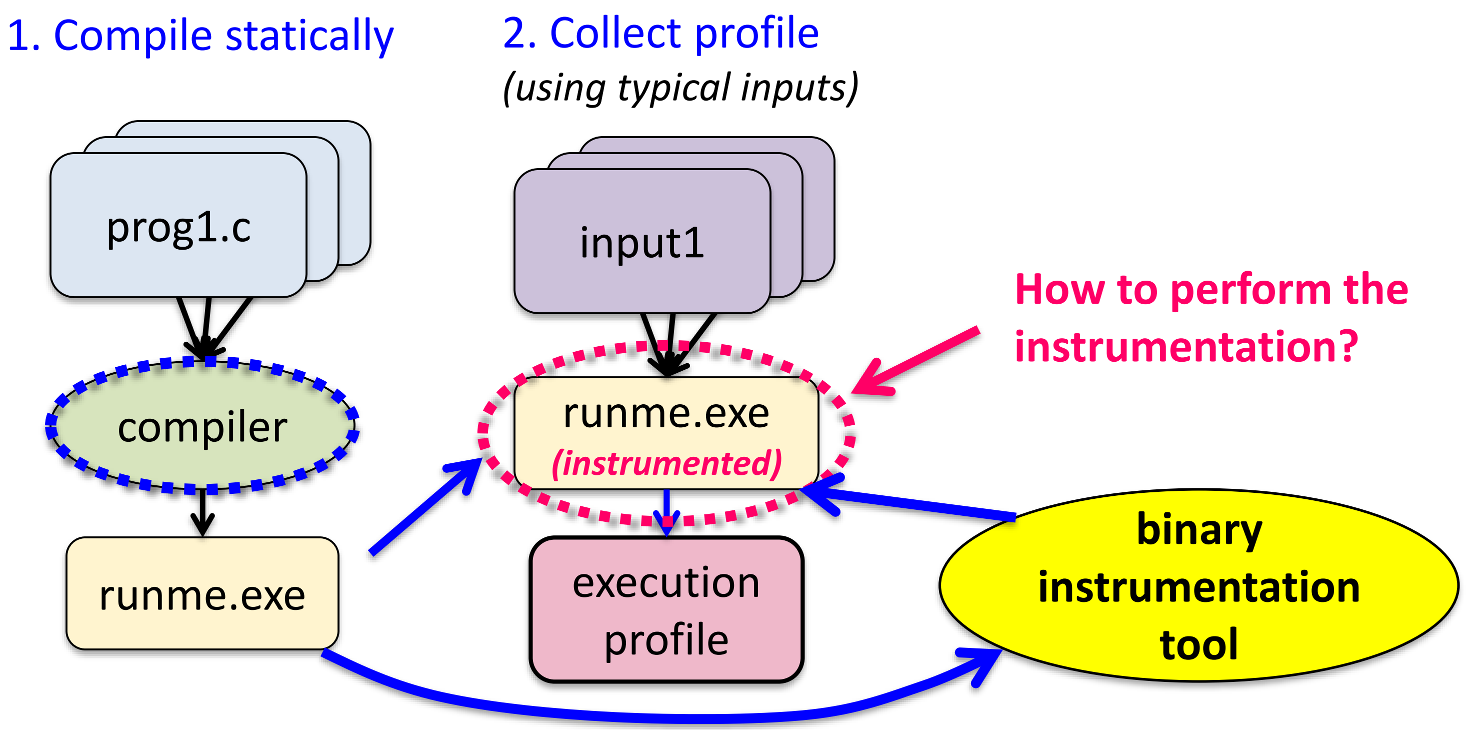
- Profile-based compile-time optimizations
  - e.g., speculative scheduling, cache optimizations, code specialization

Profile-Based Compile-time Optimization



- 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

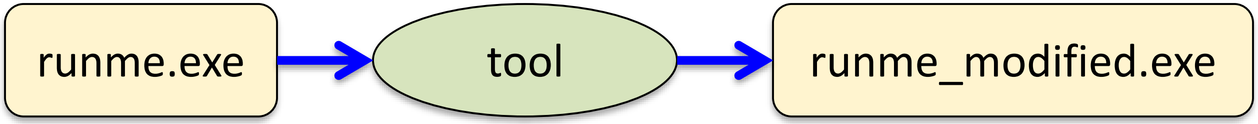
Instrumenting Executable Binaries



1. 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



runme.exe

tool

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

```
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

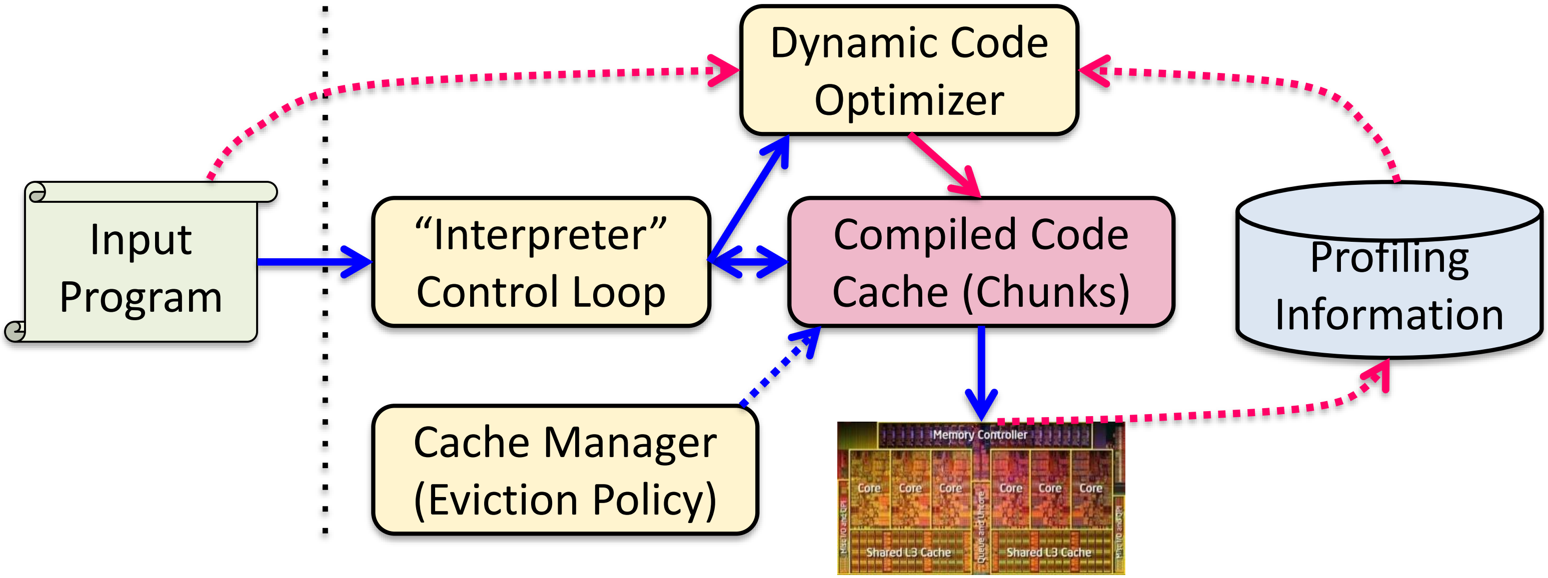
3) Dynamic Compiler

```
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



Input Program

"Interpreter" Control Loop

Dynamic Code Optimizer

Compiled Code Cache (Chunks)

Profiling Information

Cache Manager (Eviction Policy)

- 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

- $\Delta T_{total} = T_{compile} - (n_{executions} * T_{improvement})$
  - 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: but at cost of increasing $T_{compile}$)
  - Focus on large $n_{executions}$ (compile/optimize hot spots)

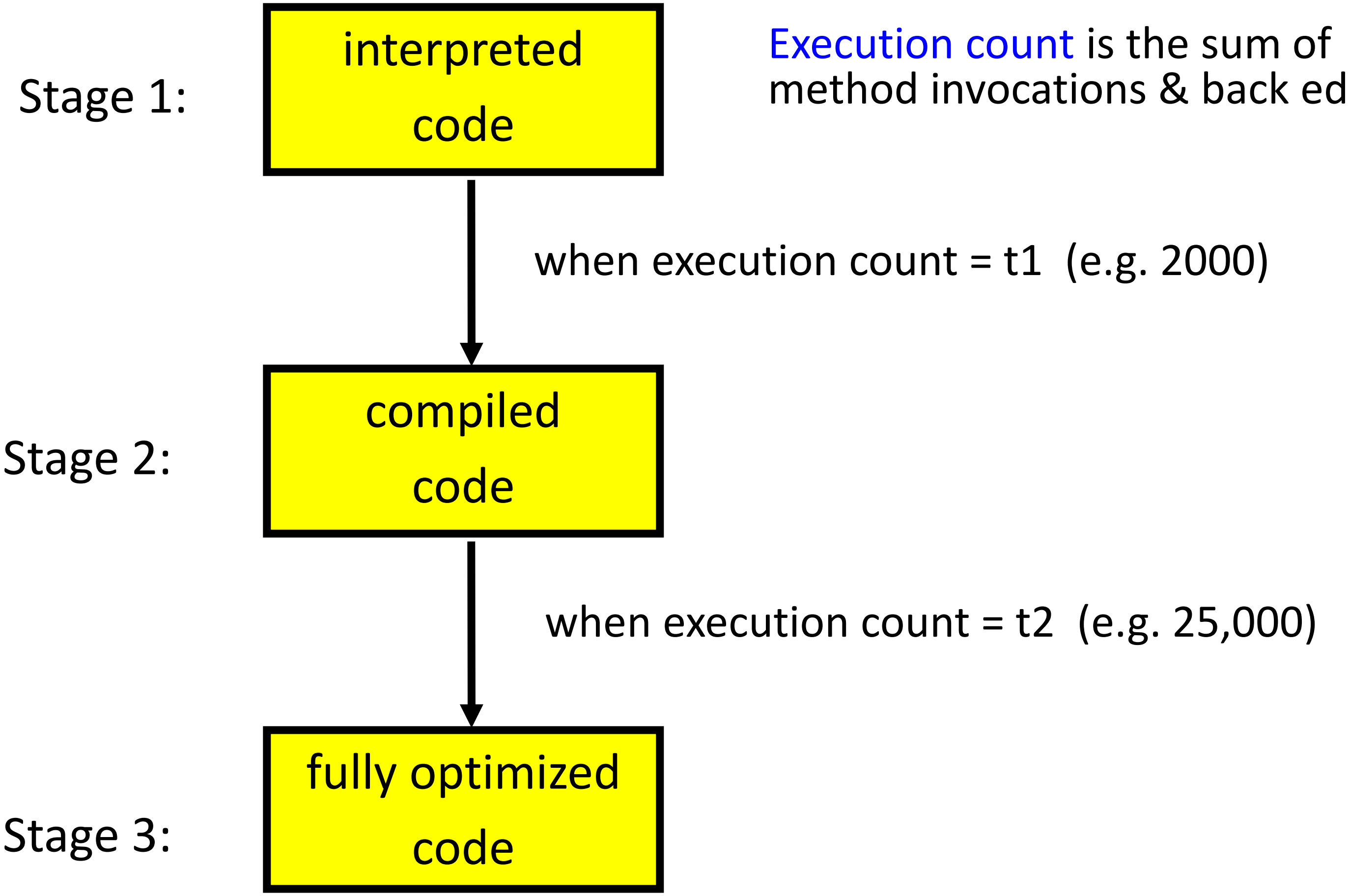
- 80/20 rule: Pareto Principle
  - 20% of the work for 80% of the advantage

Latency vs. Throughput

- Tradeoff: startup speed vs. execution performance

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

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

```
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 */
  }
}
```

Hot loop →

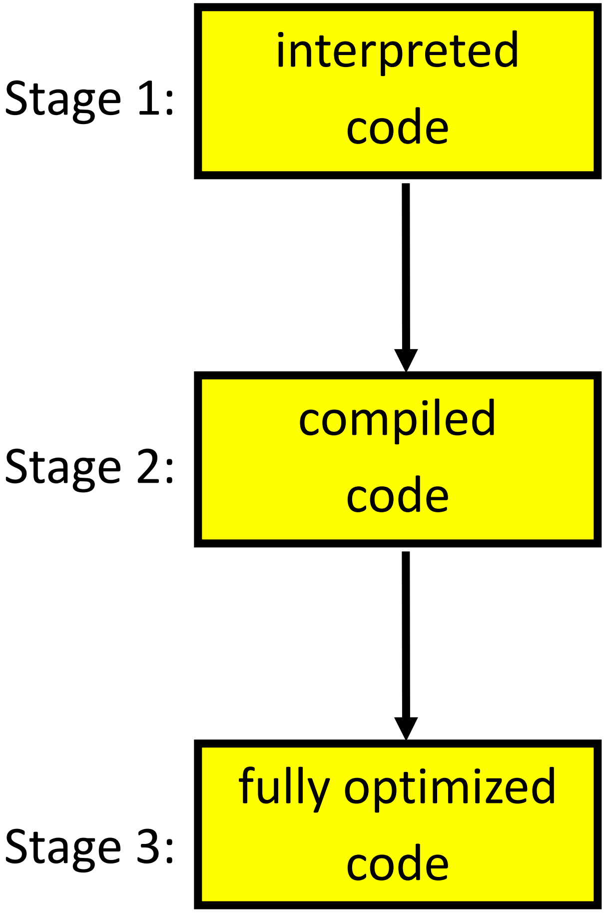
Example: SpecJVM98 db

```
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



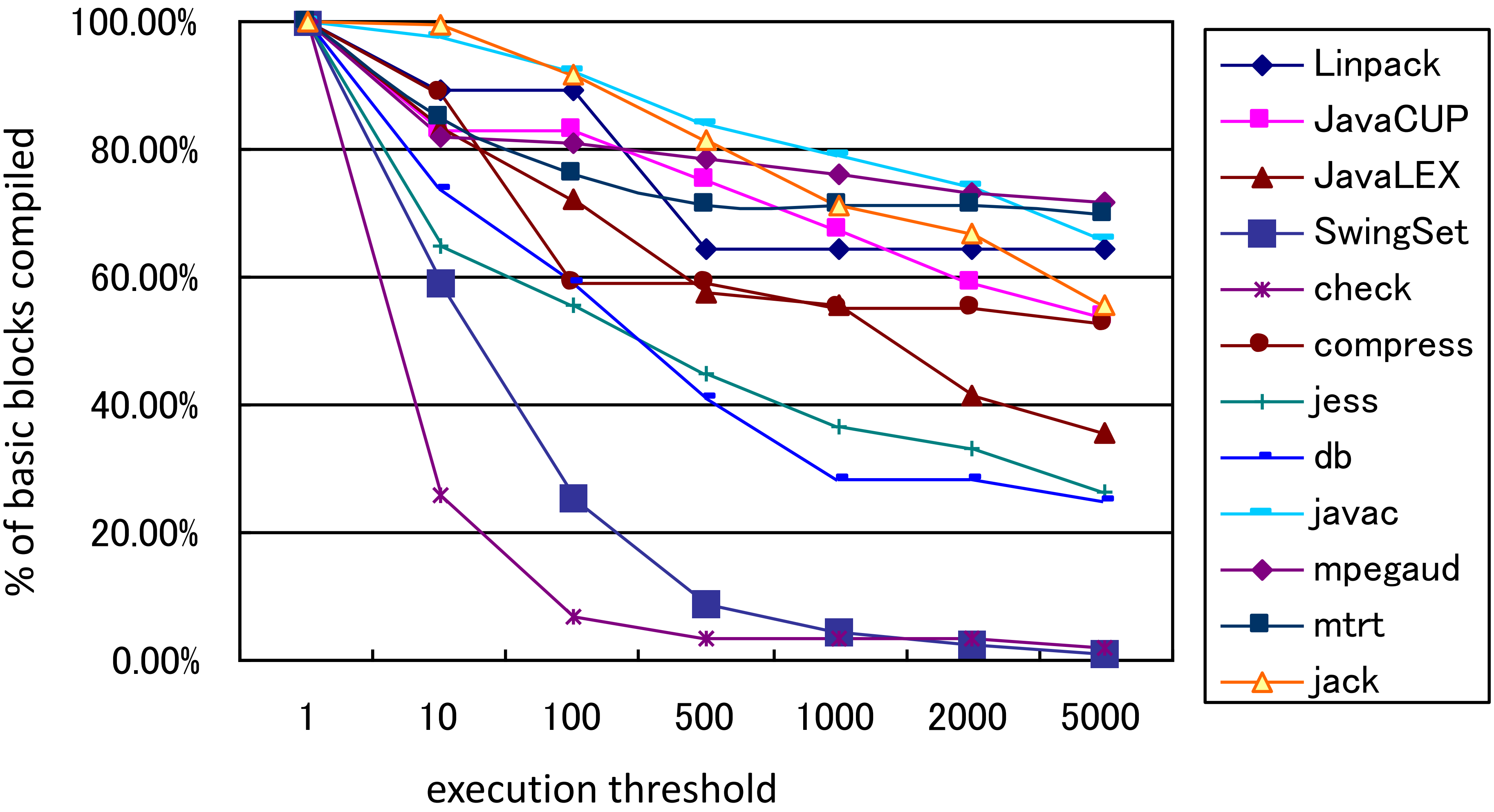
Stage 1: interpreted code

Stage 2: compiled code

Stage 3: fully optimized code

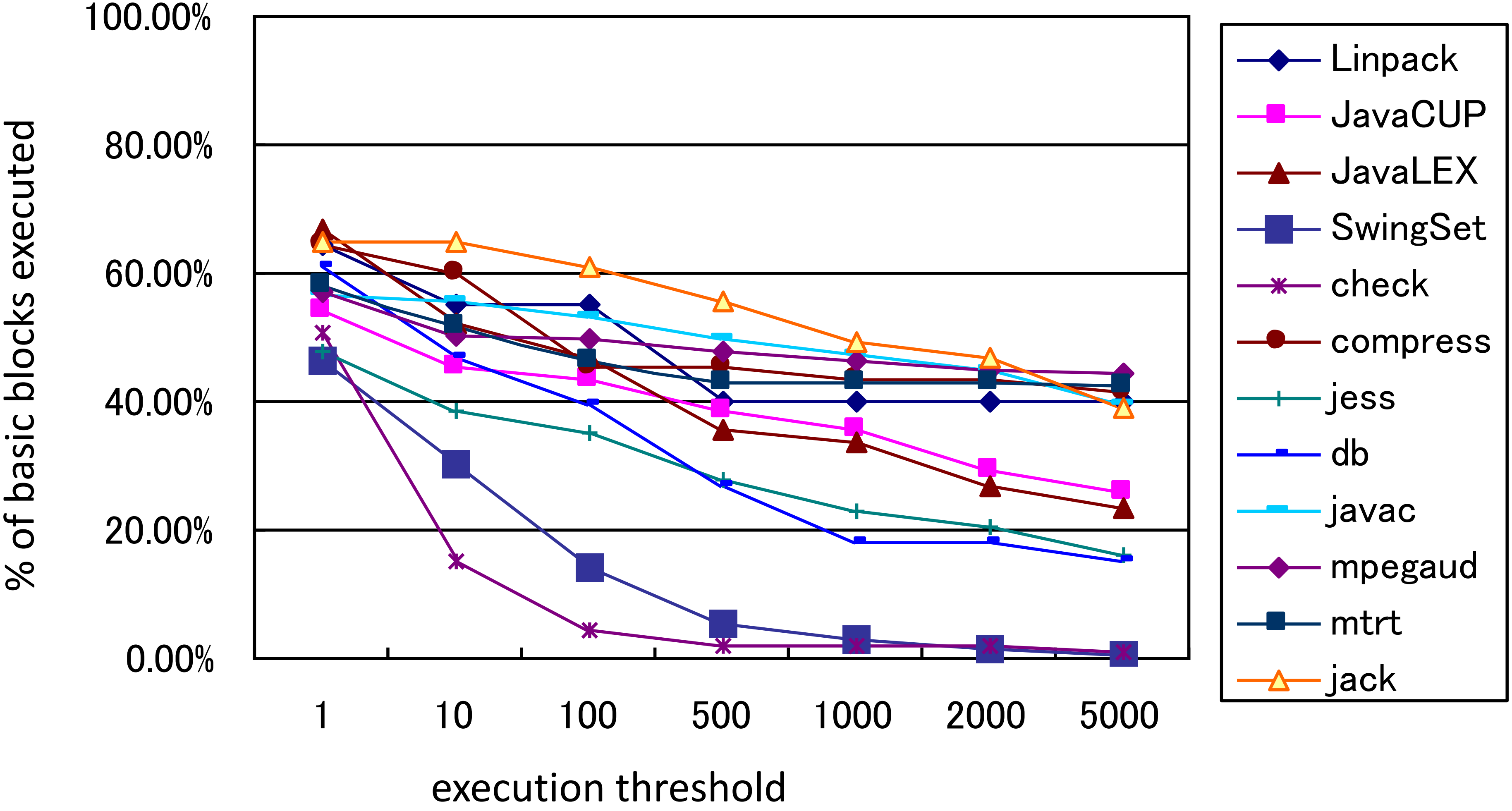
% of Basic Blocks in Methods that are Executed > Threshold Times

(hence would get compiled under per-method strategy)



% of Basic Blocks that are Executed > Threshold Times

(hence get compiled under per-basic-block strategy)

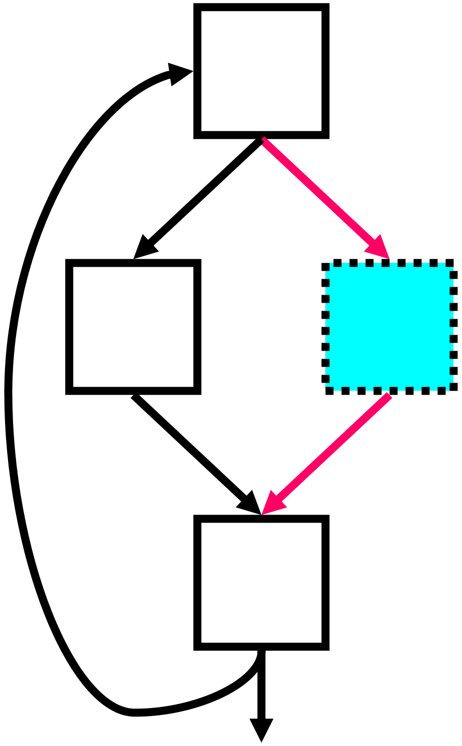


Dynamic Code Transformations

- 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



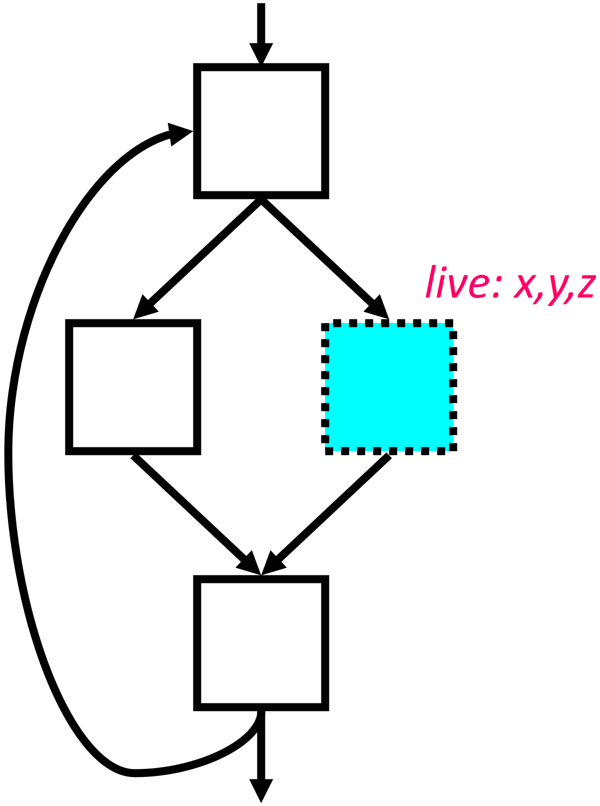
Goal: Program runs correctly with white blocks compiled and blue blocks interpreted

What are the challenges?

- How to transition from white to blue
- How to transition from blue to white
- How to compile/optimize ignoring blue

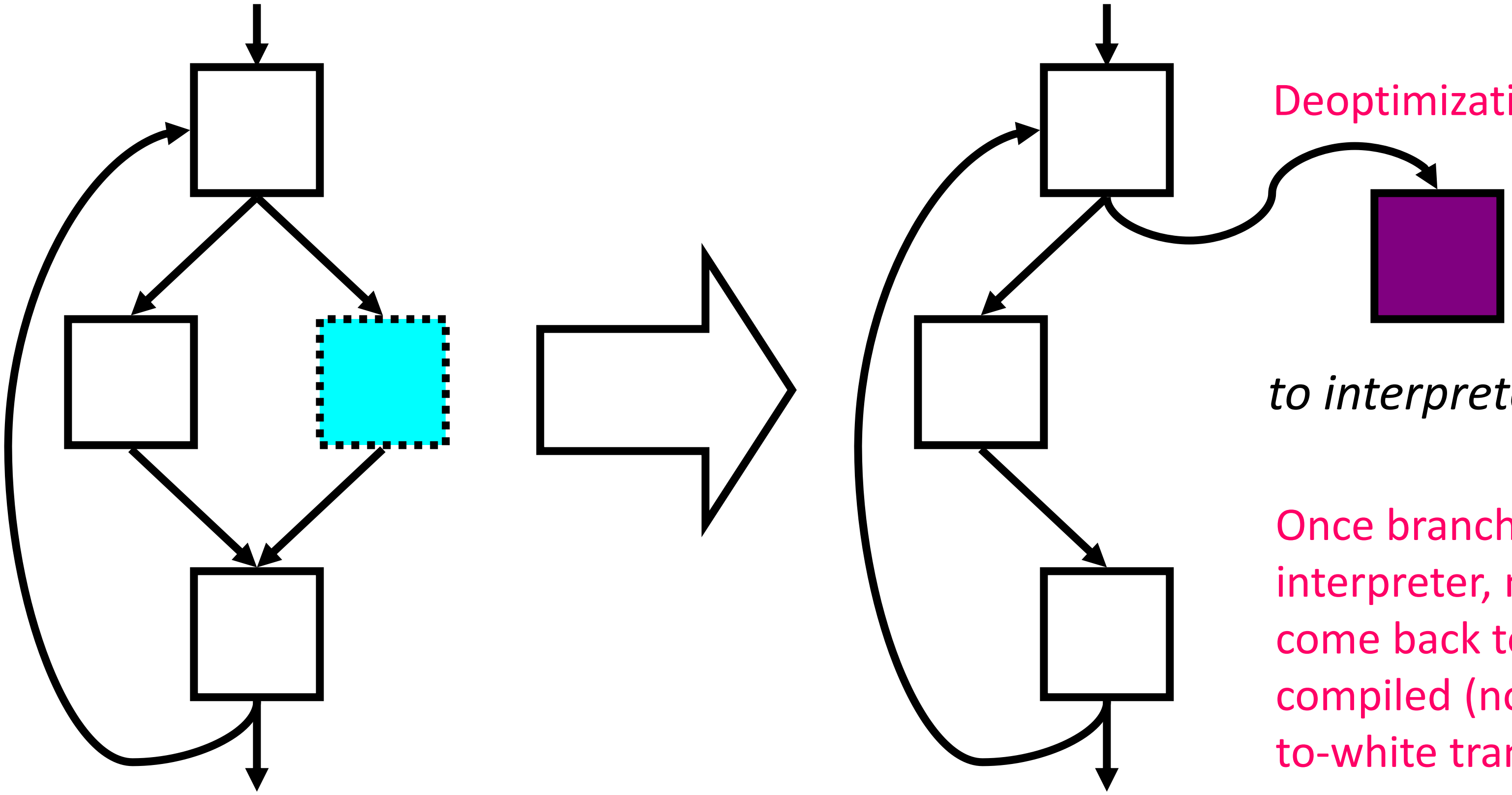
Partial Method Compilation

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



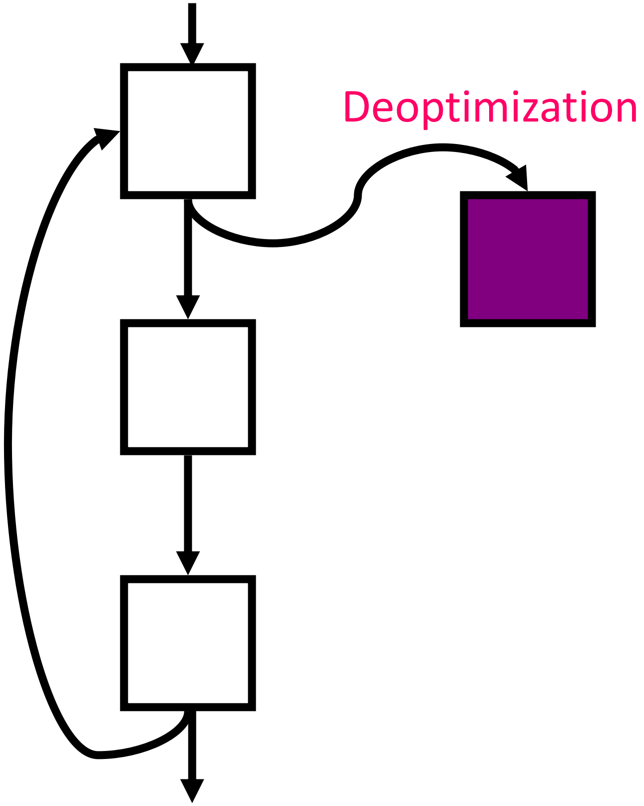
Deoptimization

to interpreter…

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

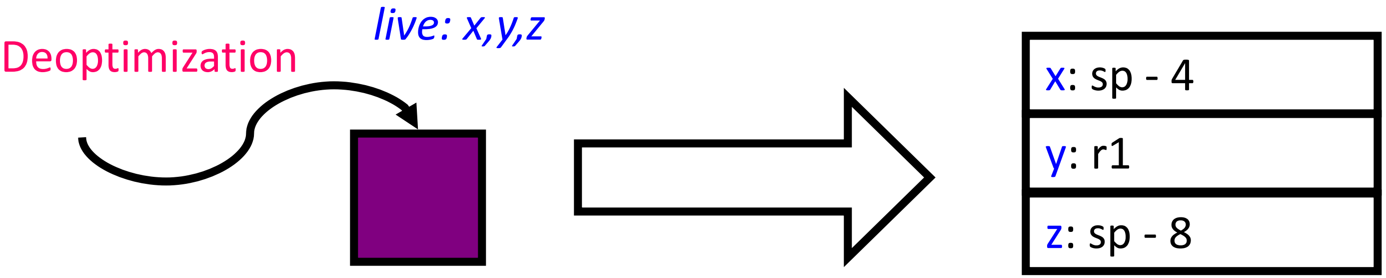
Partial Method Compilation

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
   - Used to reconstruct the interpreter state



Deoptimization

live: x,y,z

x: sp - 4

y: r1

z: sp - 8

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;
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;
    ...
}
```

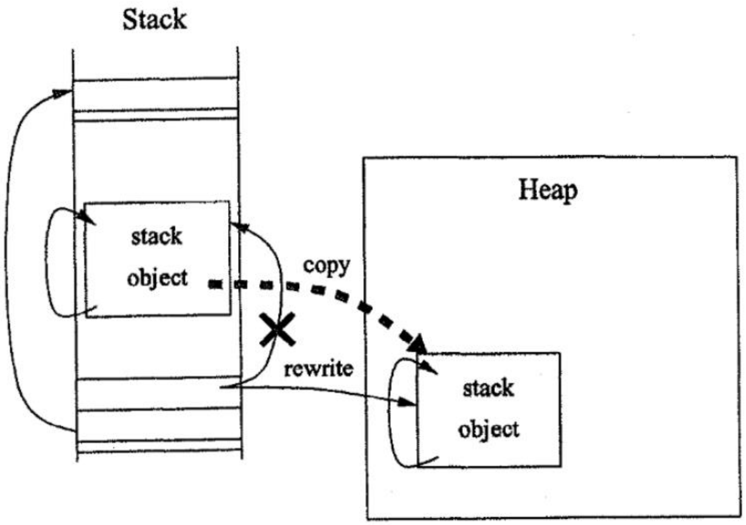
May in fact undo an optimization done by the compiler (that did not know branch was rare)

V. Escape Analysis

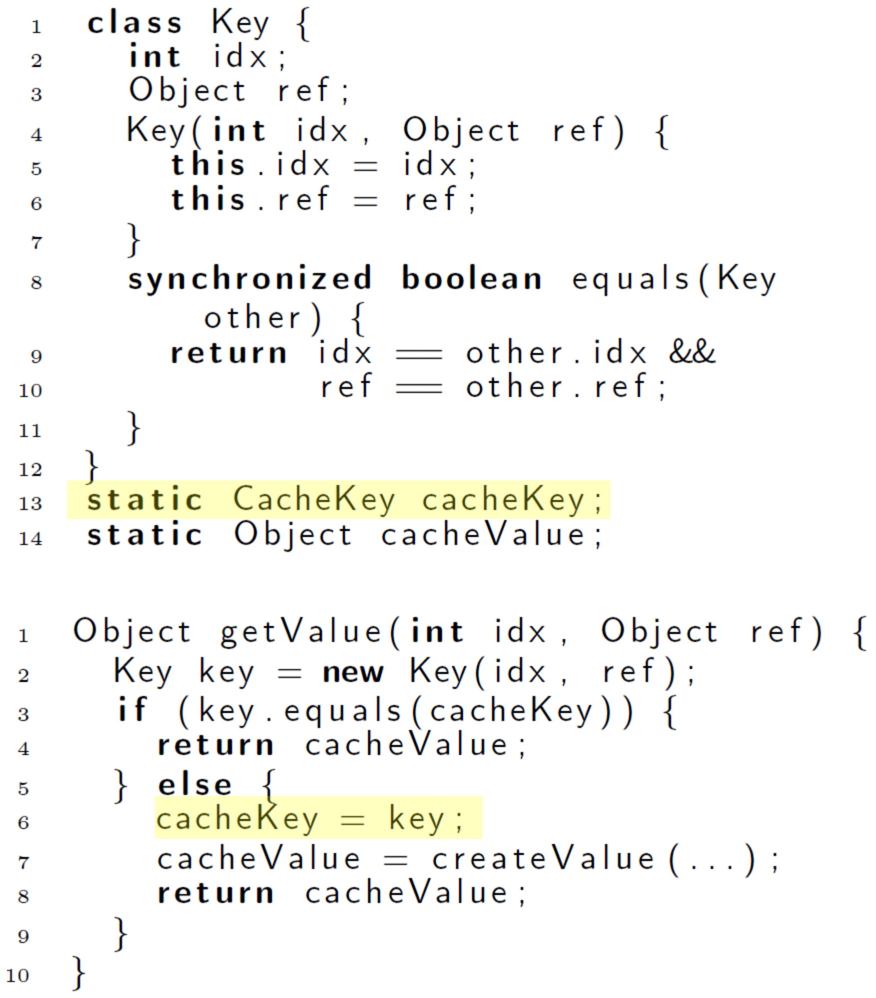
- Escape analysis finds objects that do not escape a method or a thread
  - "Captured" by method:
    - can be allocated on the stack or in registers, avoiding heap allocation
    - scalar replacement: replace the object's fields with local variables
  - "Captured" by thread:
    - can avoid synchronization 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 (i.e., non-rare) 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



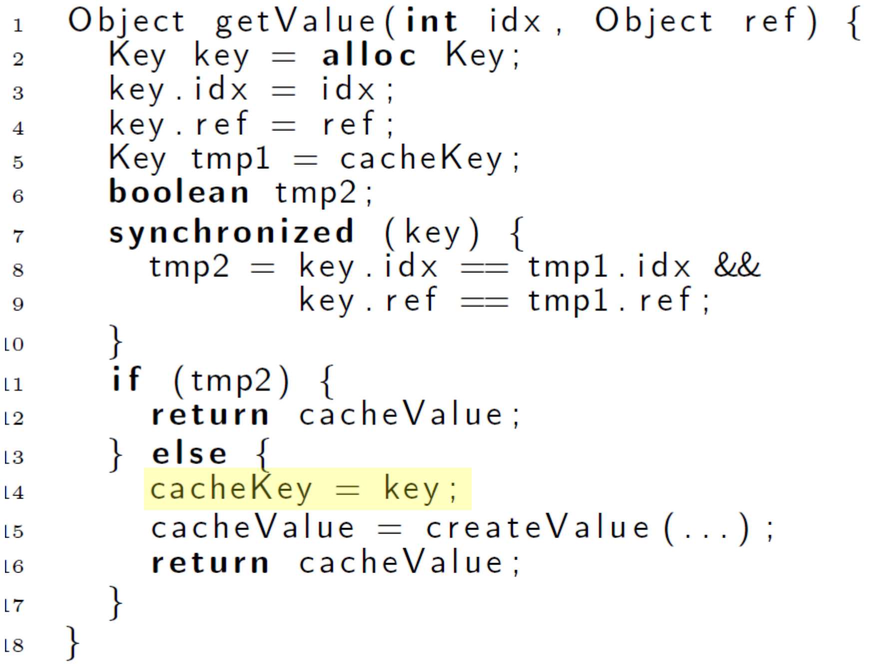
Partial Escape Analysis Example



```
class Key {
  int idx;
  Object ref;
  Key(int idx, Object ref) {
    this.idx = idx;
    this.ref = ref;
  }
  synchronized boolean equals(Key other) {
    return idx == other.idx &&
           ref == other.ref;
  }
}
static CacheKey cacheKey;
static Object cacheValue;
```

```
Object getValue(int idx, Object ref) {
  Key key = new Key(idx, ref);
  if (key.equals(cacheKey)) {
    return cacheValue;
  } else {
    cacheKey = key;
    cacheValue = createValue(...);
    return cacheValue;
  }
}
```

Listing 4: Complex example.

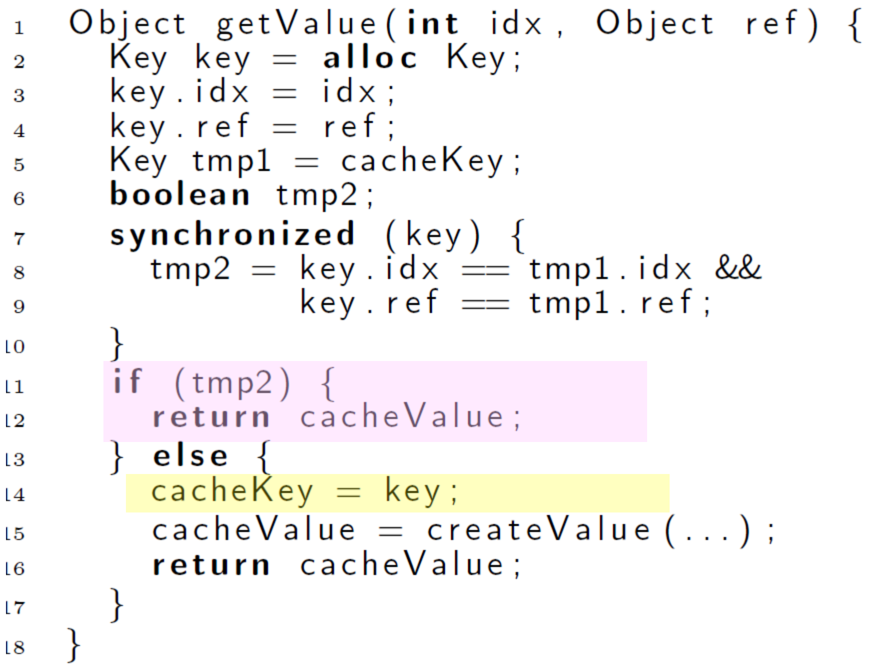


```
Object getValue(int idx, Object ref) {
  Key key = alloc Key;
  key.idx = idx;
  key.ref = ref;
  Key tmp1 = cacheKey;
  boolean tmp2;
  synchronized (key) {
    tmp2 = key.idx == tmp1.idx &&
           key.ref == tmp1.ref;
  }
  if (tmp2) {
    return cacheValue;
  } else {
    cacheKey = key;
    cacheValue = createValue(...);
    return cacheValue;
  }
}
```

Listing 5: Example from Listing 4 after inlining.

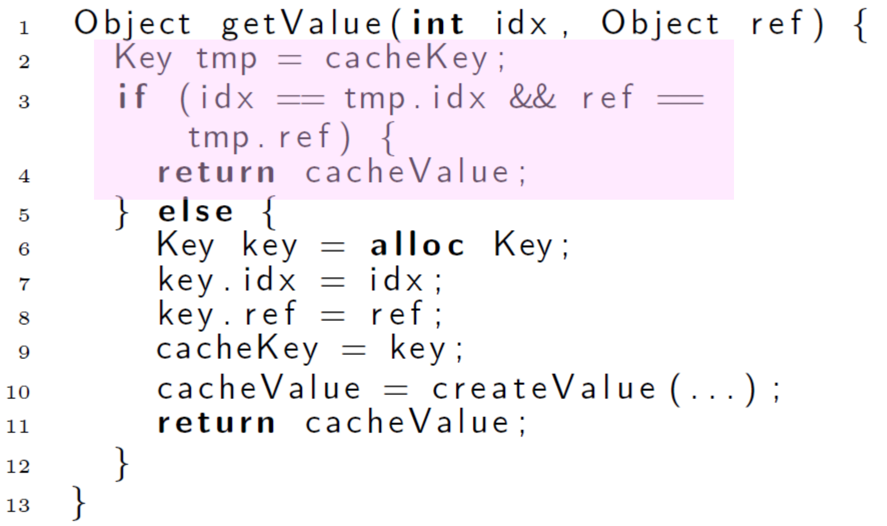
Allocated object escapes into global variable cacheKey

Partial Escape Analysis Example (cont.)



```
Object getValue(int idx, Object ref) {
  Key key = alloc Key;
  key.idx = idx;
  key.ref = ref;
  Key tmp1 = cacheKey;
  boolean tmp2;
  synchronized (key) {
    tmp2 = key.idx == tmp1.idx &&
           key.ref == tmp1.ref;
  }
  if (tmp2) {
    return cacheValue;
  } else {
    cacheKey = key;
    cacheValue = createValue(...);
    return cacheValue;
  }
}
```

Listing 5: Example from Listing 4 after inlining.



```
Object getValue(int idx, Object ref) {
  Key tmp = cacheKey;
  if (idx == tmp.idx && ref ==
      tmp.ref) {
    return cacheValue;
  } else {
    Key key = alloc Key;
    key.idx = idx;
    key.ref = ref;
    cacheKey = key;
    cacheValue = createValue(...);
    return cacheValue;
  }
}
```

Listing 6: Example from Listing 5 after Partial Escape Analysis.

Considering only the **if** branch, the allocated object does NOT escape

- In the **if** branch, avoid the allocation and remove the synchronization

Oracle HotSpot JVM and Graal Dynamic Compiler

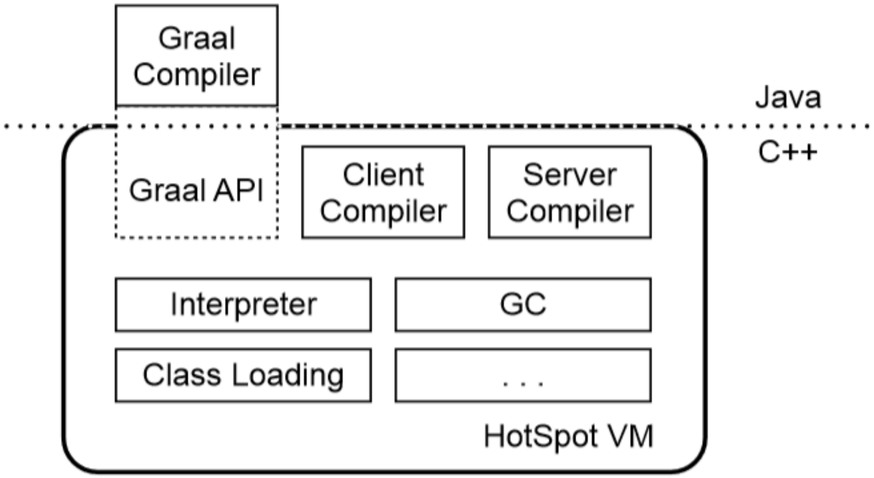
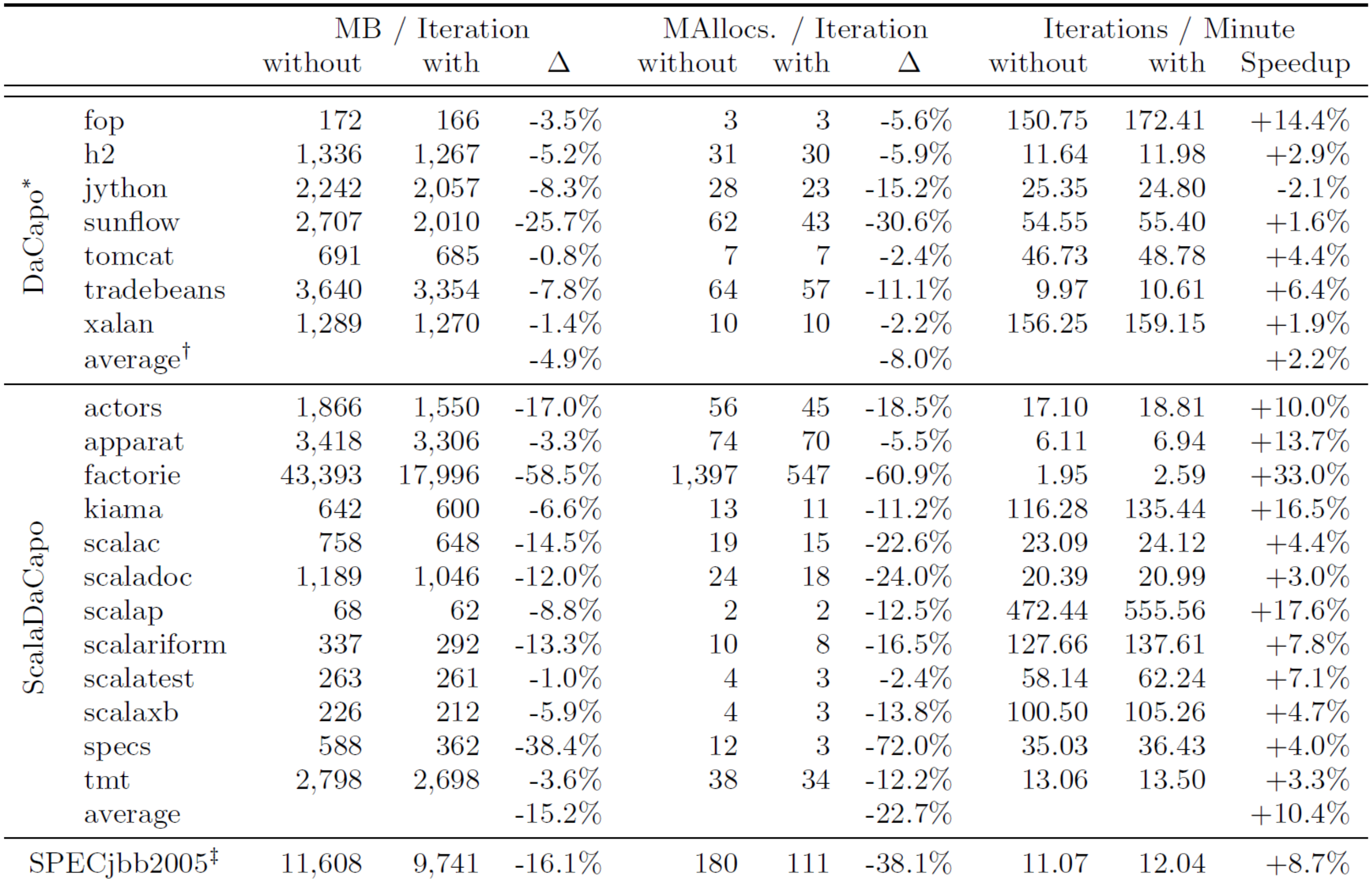


Figure 1: Overview of HotSpot and Graal.

Partial Escape Analysis implemented as an optimization on the Graal Compiler IR

Benefits from Partial Escape Analysis



		MB / Iteration			MAllocs. / Iteration			Iterations / Minute		
		without	with	Δ	without	with	Δ	without	with	Speedup
DaCapo*	fop	172	166	-3.5%	3	3	-5.6%	150.75	172.41	+14.4%
	h2	1,336	1,267	-5.2%	31	30	-5.9%	11.64	11.98	+2.9%
	jython	2,242	2,057	-8.3%	28	23	-15.2%	25.35	24.80	-2.1%
	sunflow	2,707	2,010	-25.7%	62	43	-30.6%	54.55	55.40	+1.6%
	tomcat	691	685	-0.8%	7	7	-2.4%	46.73	48.78	+4.4%
	tradebeans	3,640	3,354	-7.8%	64	57	-11.1%	9.97	10.61	+6.4%
	xalan	1,289	1,270	-1.4%	10	10	-2.2%	156.25	159.15	+1.9%
	average†			-4.9%			-8.0%			+2.2%
ScalaDaCapo	actors	1,866	1,550	-17.0%	56	45	-18.5%	17.10	18.81	+10.0%
	apparat	3,418	3,306	-3.3%	74	70	-5.5%	6.11	6.94	+13.7%
	factorie	43,393	17,996	-58.5%	1,397	547	-60.9%	1.95	2.59	+33.0%
	kiama	642	600	-6.6%	13	11	-11.2%	116.28	135.44	+16.5%
	scalac	758	648	-14.5%	19	15	-22.6%	23.09	24.12	+4.4%
	scaladoc	1,189	1,046	-12.0%	24	18	-24.0%	20.39	20.99	+3.0%
	scalap	68	62	-8.8%	2	2	-12.5%	472.44	555.56	+17.6%
	scalariform	337	292	-13.3%	10	8	-16.5%	127.66	137.61	+7.8%
	scalatest	263	261	-1.0%	4	3	-2.4%	58.14	62.24	+7.1%
	scalaxb	226	212	-5.9%	4	3	-13.8%	100.50	105.26	+4.7%
	specs	588	362	-38.4%	12	3	-72.0%	35.03	36.43	+4.0%
	tmt	2,798	2,698	-3.6%	38	34	-12.2%	13.06	13.50	+3.3%
	average			-15.2%			-22.7%			+10.4%
SPECjbb2005‡		11,608	9,741	-16.1%	180	111	-38.1%	11.07	12.04	+8.7%

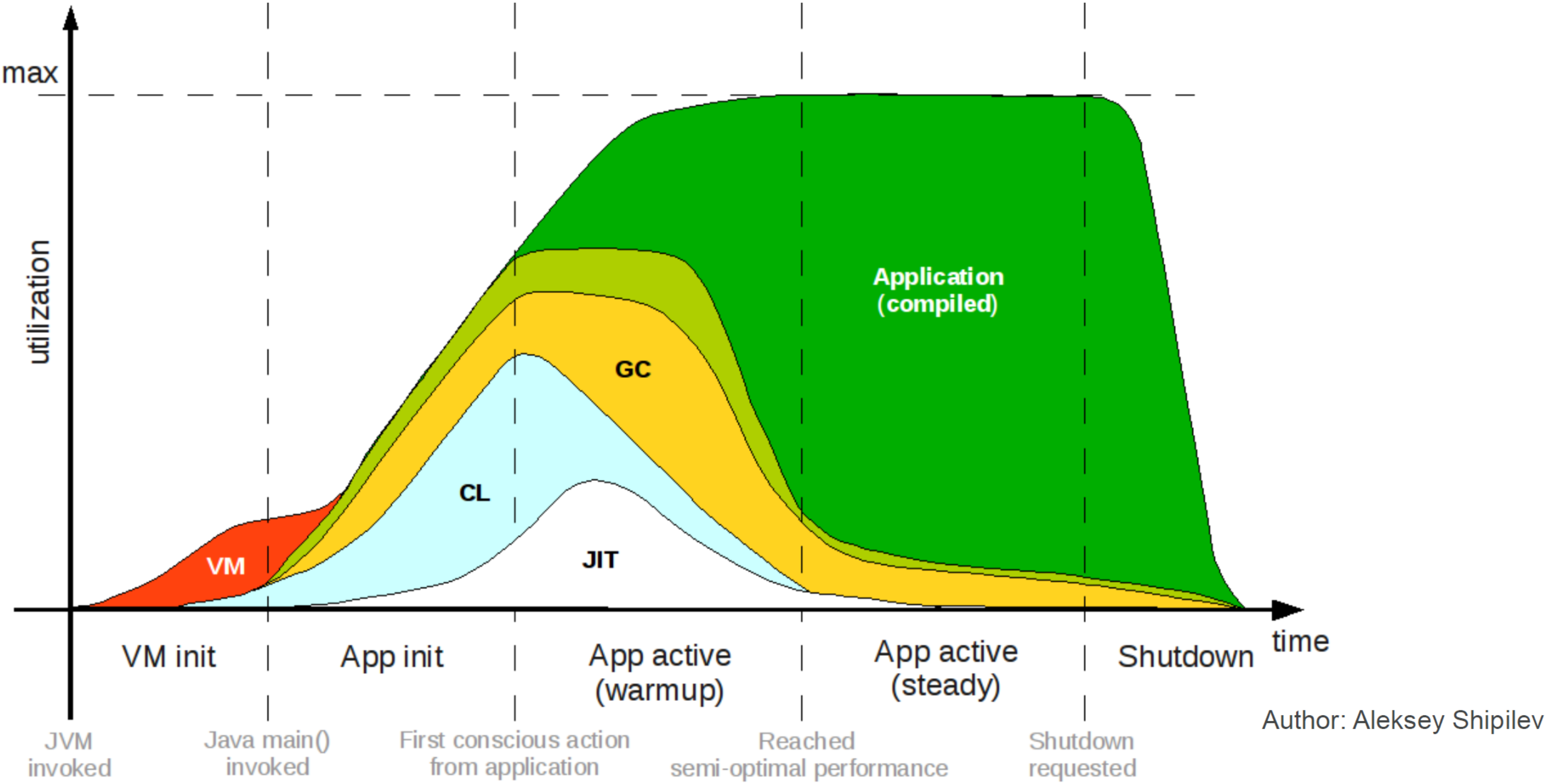
Table 1: Evaluation of size and number of allocations, and performance on (Scala)DaCapo and SPECjbb2005

Carnegie Mellon

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
    - untaken 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



Author: Aleksey Shipilev

Summary: Beyond Static Compilation

1) Profile-based Compiler: high-level → binary, static
   - Uses (dynamic=runtime) information collected in profiling passes

2) Interpreter: high-level, emulate, dynamic

3) Dynamic compilation / code optimization: high-level → 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

Wednesday's Class

- Memory Hierarchy Optimizations
  - ALSU 7.4.2-7.4.3, 11.2-11.5