

Trace Compilation

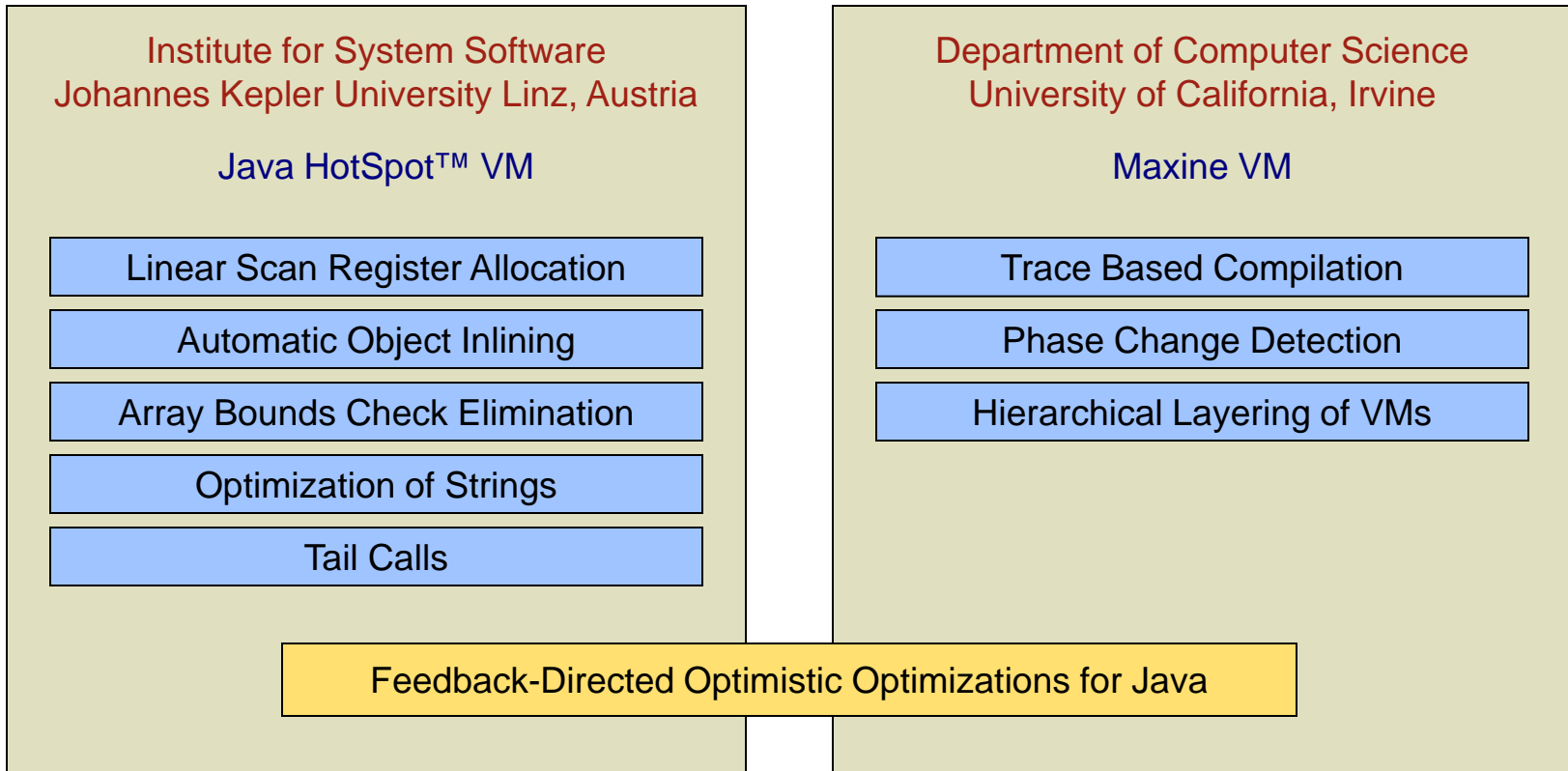
Christian Wimmer
cwimmer@uci.edu
www.christianwimmer.at

September 2009



Department of Computer Science
University of California, Irvine

Background





Commercial 1 – Tail Call Optimization

- Credit: Arnold Schwaighofer, JKU Linz
- Hard tail calls
 - Prefix for invoke bytecodes (reuse “wide” prefix)
 - Semantics checked by bytecode verifier
- Prototype implementation for Java HotSpot VM
 - Interpreter, Client Compiler, Server Compiler
 - Problem: Fixed size of compiled frames, parameters part of caller
 - Not enough space for parameter of tail calls
 - Non-sibling tail calls require adapter frames
- Protection domains
 - Security information is attached to stack frames
 - Problem: Tail calls across protection domains are security hole
 - Solution 1: Don't allow, but throw exception
 - Solution 2: Perform normal call and compress stack later if needed

Commercial 2 – Client Compiler Visualizer



The screenshot displays the Java HotSpot Client Compiler Visualizer interface for the `String.hashCode()` method. The interface is divided into several panes:

- Compiled Methods:** A tree view on the left showing the method's compilation stages: `String.hashCode()`, `After Generation of HIR`, `Before Register Allocation`, `After Register Allocation`, and `Before Code Generation`. Other methods like `String.charAt`, `String.indexOf`, and `Random.nextDouble` are also listed.
- Control Flow Graph (CFG):** A graph showing basic blocks (B0, B1, B2, B3, B4, B5, B6, B7) and their interconnections.
- Intermediate Representation (HIR):** A graph showing nodes (i31, i17, v9, i18, i17, v1E, i12, i26, i20, i24, a6, i10, a11, i16, i22, i21, i23) and their relationships.
- Stack Map:** A grid showing stack frames (B1, B3(1), B4(1)) and instructions (43 [ecx|], 44 [es|], 45 [edx|], 46 [ebx|], 47 [eax|], 48 [es|], 49 [ecx|], 50 [ecx|], 51 [es|], 52 [edx|], 53 [ecx|], 57 [stack:1|L], 58 [stack:1|L]).
- Bytecodes:** A list of bytecodes for block B1: `aload_0`, `getfield java.lang.String.offset I (350)`, `istore_2`, `aload_0`, `getfield java.lang.String.value [C (351)`, `astore_3`, `aload_0`, `getfield java.lang.String.count I (348)`, `istore %4`, `iconst_0`, `istore %5`.
- HIR:** A table showing HIR instructions for block B1:

bci	use	tid	result	instr
. 10	1	i10	[R44 I]	a6._12 (I)
. 15	4	a11	[R45 L]	a6._8 (I)
. 20	4	i12	[R46 I]	a6._16 (I)
. 26	0	v14		goto B3
- LIR:** A table showing LIR instructions for block B1:

nr	instr
24	label [label:0x9b1340]
26	move [Base:[R41 L] Disp: 12 I] [R44 I]
28	move [Base:[R41 L] Disp: 8 L] [R45 L]
30	move [Base:[R41 L] Disp: 16 I] [R46 I]
32	move [R44 I] [R48 I]
34	move [R42 I] [R47 I]
36	move [int:0 I] [R49 I]
38	branch [AL] [B3]
- Intervals:** A table showing intervals for block B1:

nr	instr
24	label [label:0x9b1340]
26	move [Base:[R41 L] Disp: 12 I] [R44 I]
28	move [Base:[R41 L] Disp: 8 L] [R45 L]
30	move [Base:[R41 L] Disp: 16 I] [R46 I]
32	move [R44 I] [R48 I]
34	move [R42 I] [R47 I]
36	move [int:0 I] [R49 I]
38	branch [AL] [B3]
- Data Flow:** A table showing data flow for block B1:

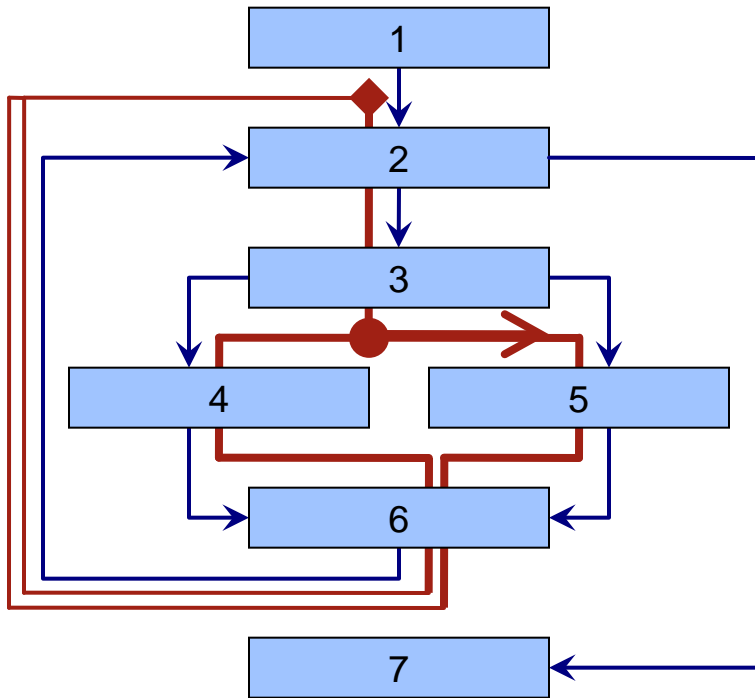
nr	instr
24	label [label:0x9b1340]
26	move [Base:[R41 L] Disp: 12 I] [R44 I]
28	move [Base:[R41 L] Disp: 8 L] [R45 L]
30	move [Base:[R41 L] Disp: 16 I] [R46 I]
32	move [R44 I] [R48 I]
34	move [R42 I] [R47 I]
36	move [int:0 I] [R49 I]
38	branch [AL] [B3]

<https://c1visualizer.dev.java.net/>

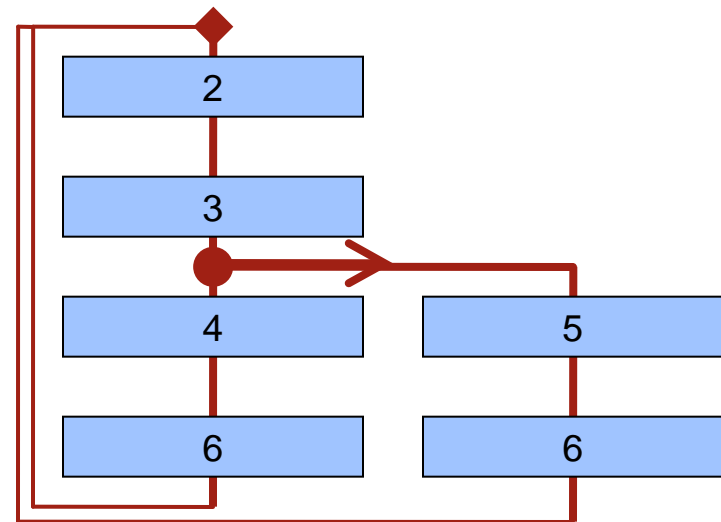
Trace Compilation



Control Flow Graph



Trace Tree





Why Trace Compilation?

Method Execution Frequency

100 Methods

150

200

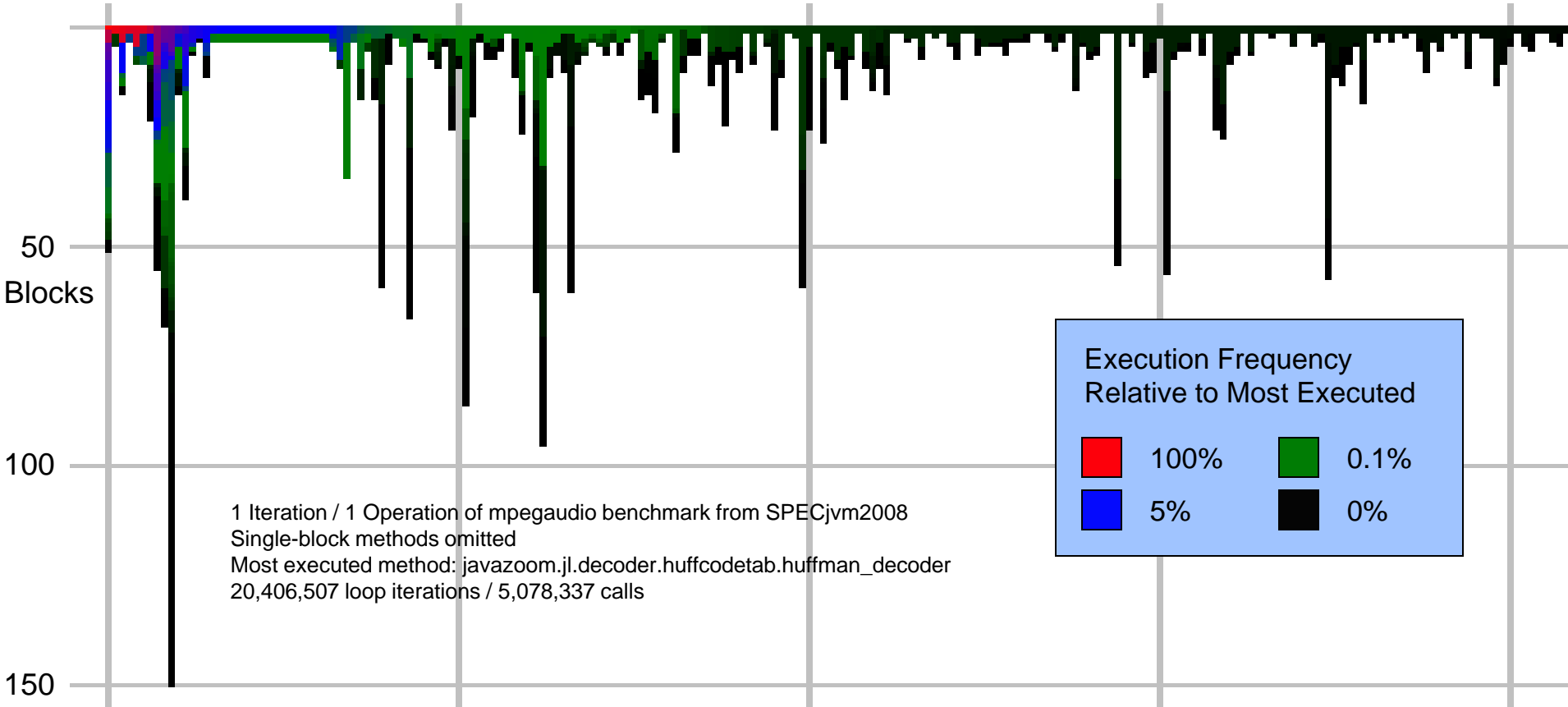


Block Execution Frequency

100 Methods

150

200



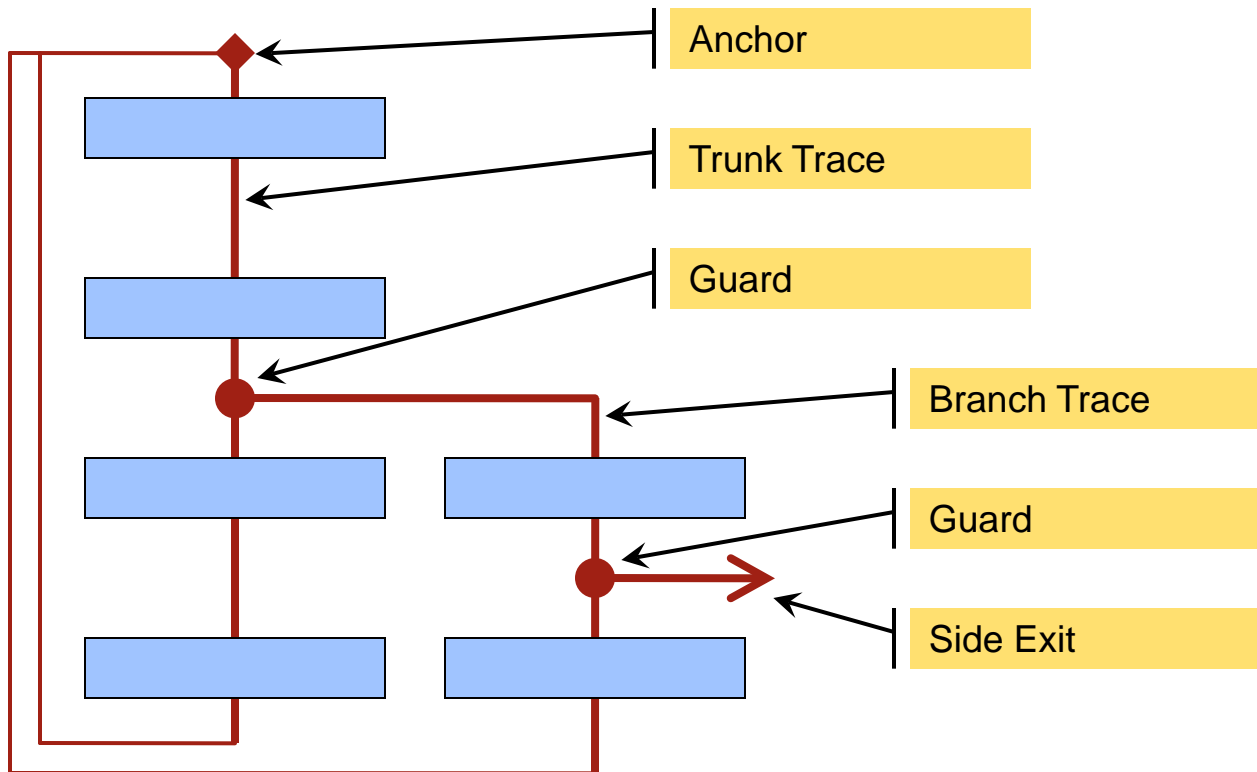
Execution Frequency
Relative to Most Executed

Red	100%	Green	0.1%
Blue	5%	Black	0%

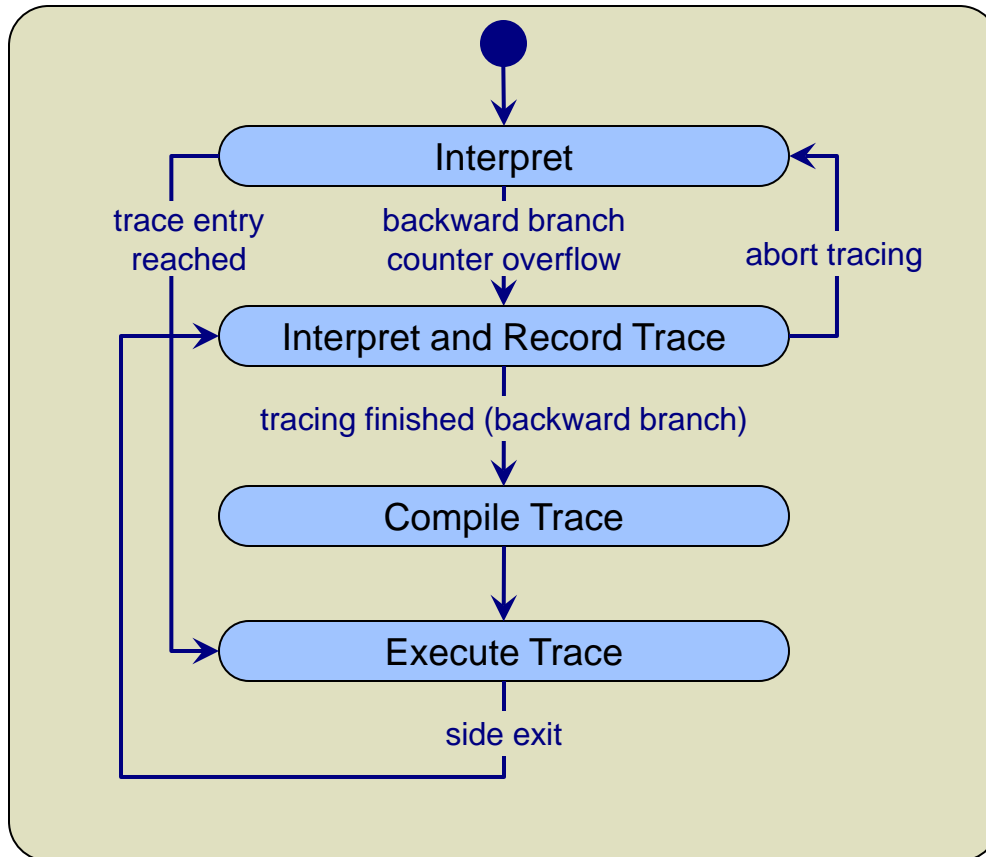
1 Iteration / 1 Operation of mpegaudio benchmark from SPECjvm2008
Single-block methods omitted
Most executed method: javazoom.jl.decoder.huffcodetab.huffman_decoder
20,406,507 loop iterations / 5,078,337 calls



Elements of a Trace Tree



Trace Compilation

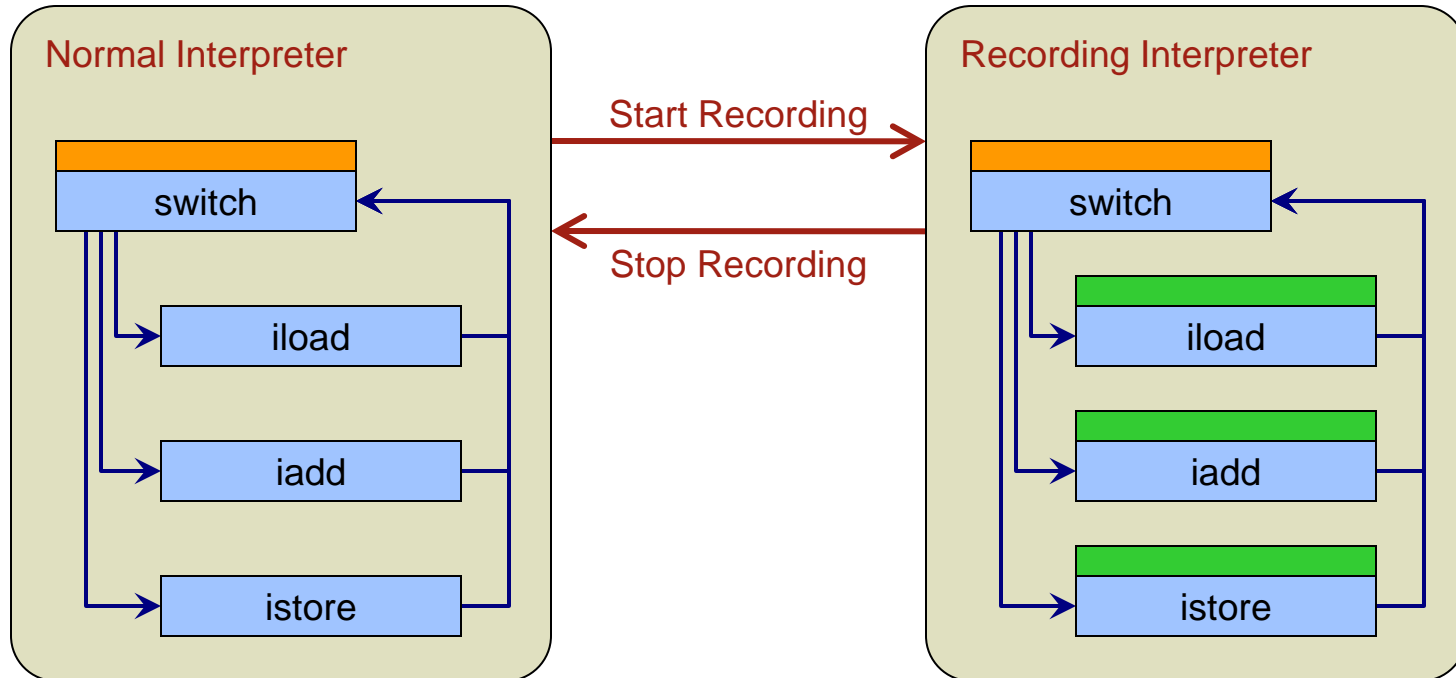




Flavors of Trace Compilers

- Dynamic Languages
 - TraceMonkey: Mozilla's JavaScript engine for Firefox
 - Elimination of dynamic type checks
 - Low compilation threshold
 - Fast compilation, no expensive optimizations (yet)
- Java – Maxine VM
 - Evolved from a research prototype (Hotpath VM)
 - Maxine has no interpreter – where to integrate trace recording?
 - Traces currently cover only loops, no merge points
- Java – HotSpot VM
 - Research funding requested, but still pending
 - Stay tuned...

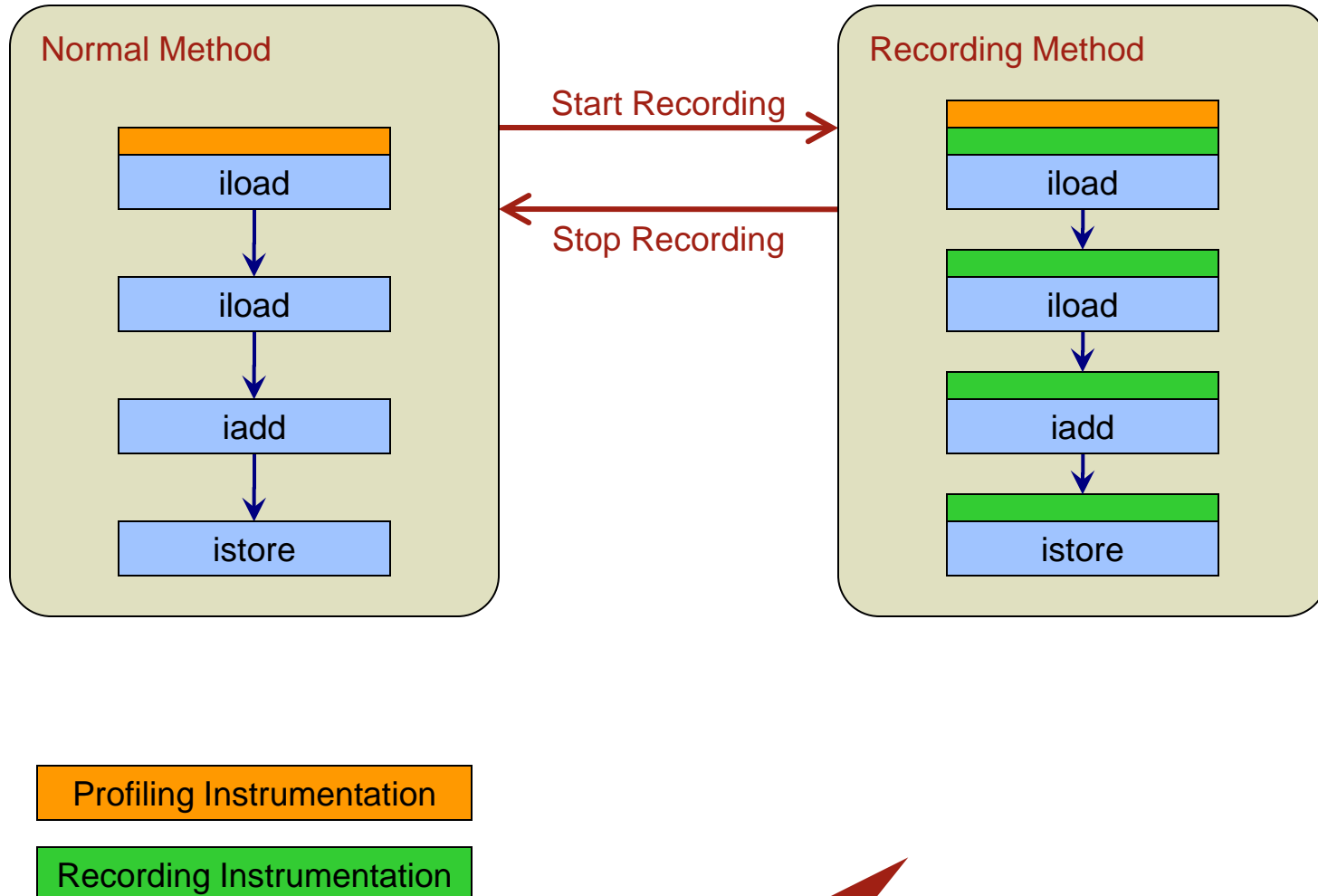
Trace Recording in Interpreter



Profiling Instrumentation

Recording Instrumentation

Trace Recording in Template JIT

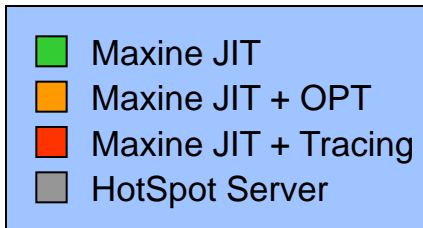
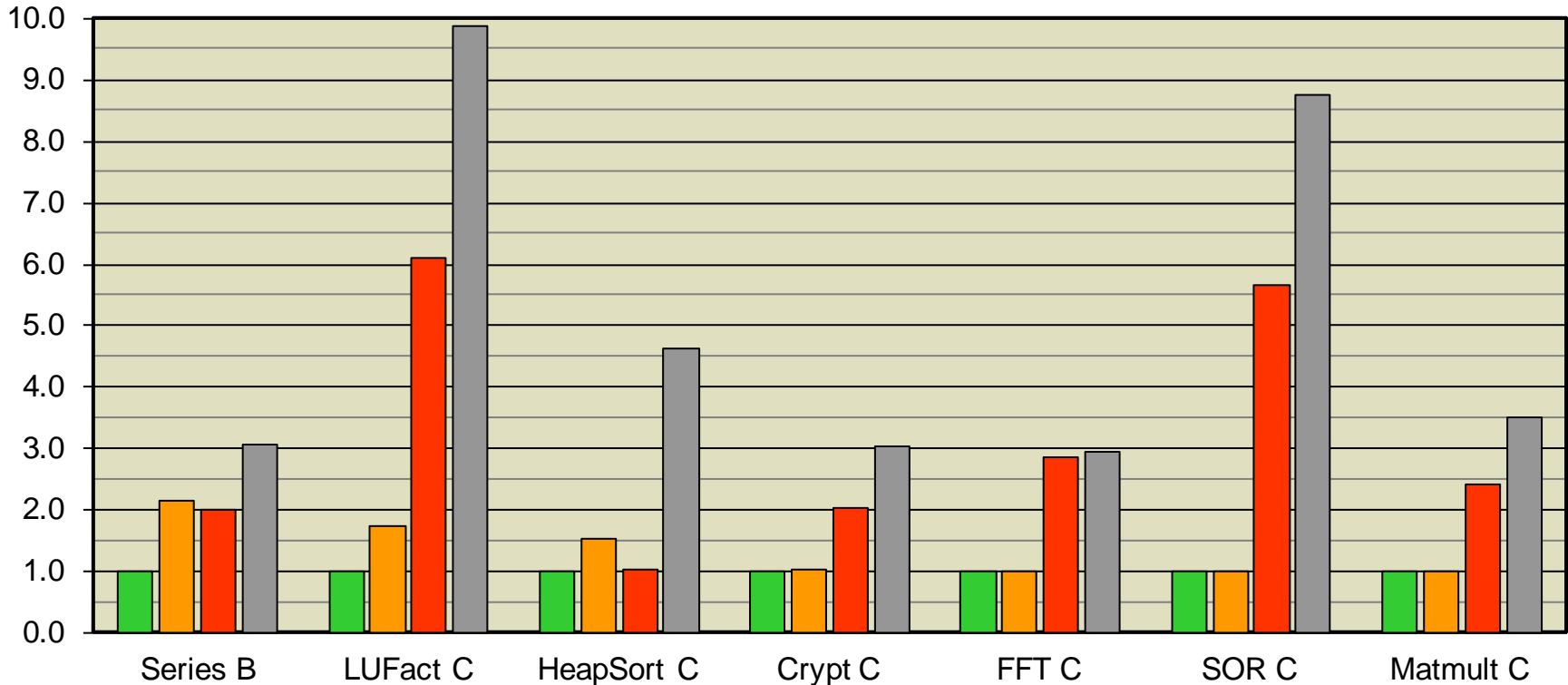


Evaluation Trace Compilation



Speedup relative to Maxine baseline compilation

Preliminary Numbers!



2 x 2.8GHz Quad-Core Intel Xeon Mac Pro
Mac OS X 10.5.6, 64 bit JVMs



Problems and Future Work

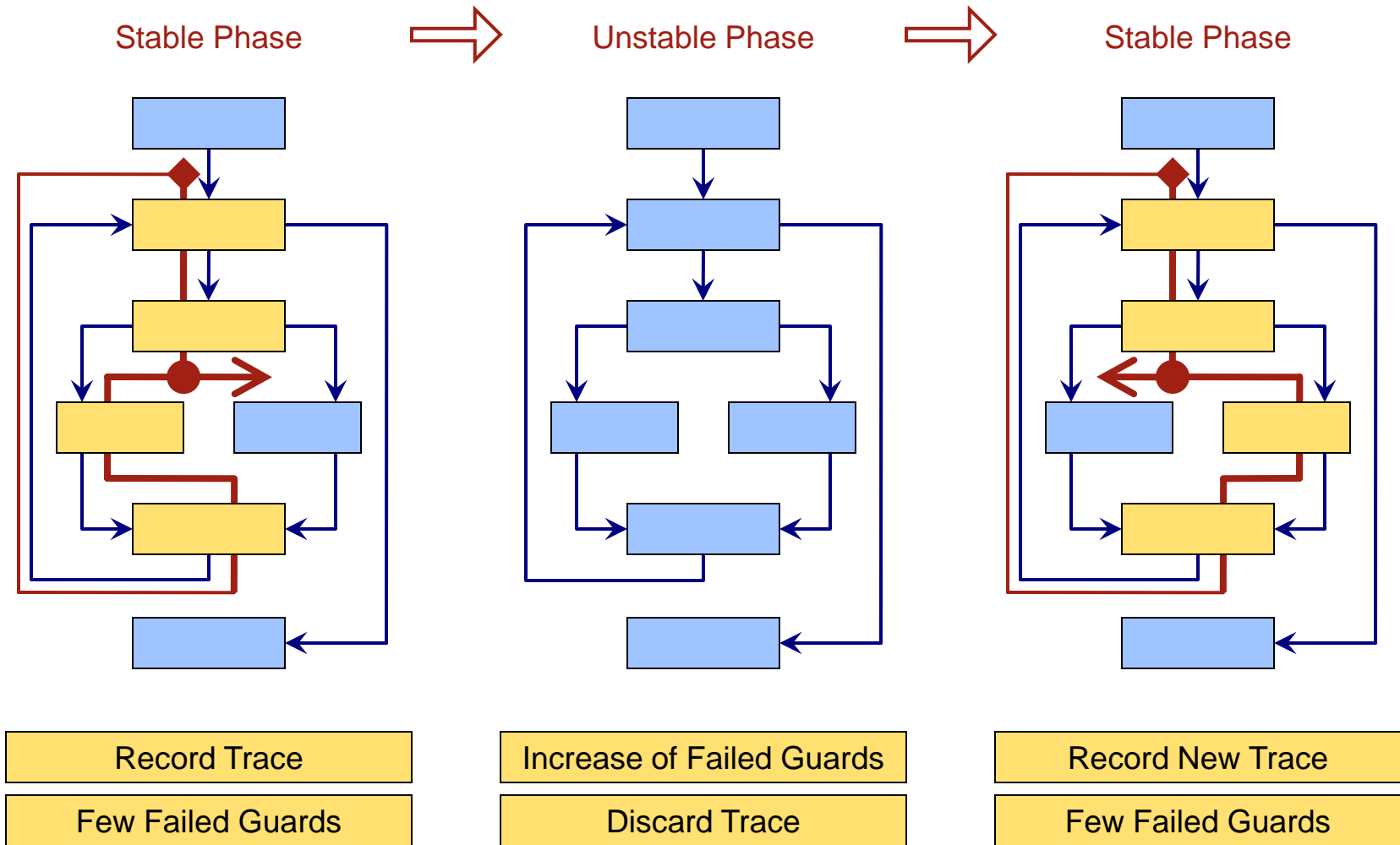
- Loop-centric approach
 - Trace and compile only loops
 - Good for numerical applications
 - Loops are not the first-order element of object-oriented applications
 - Need to optimize also on per-method basis if necessary

- Tail duplication
 - Trace trees lead to duplication of join blocks
 - Block count can explode for complex control flow
 - Need to support join blocks in traces

- Trace calling convention
 - Switch to and from traces with low overhead



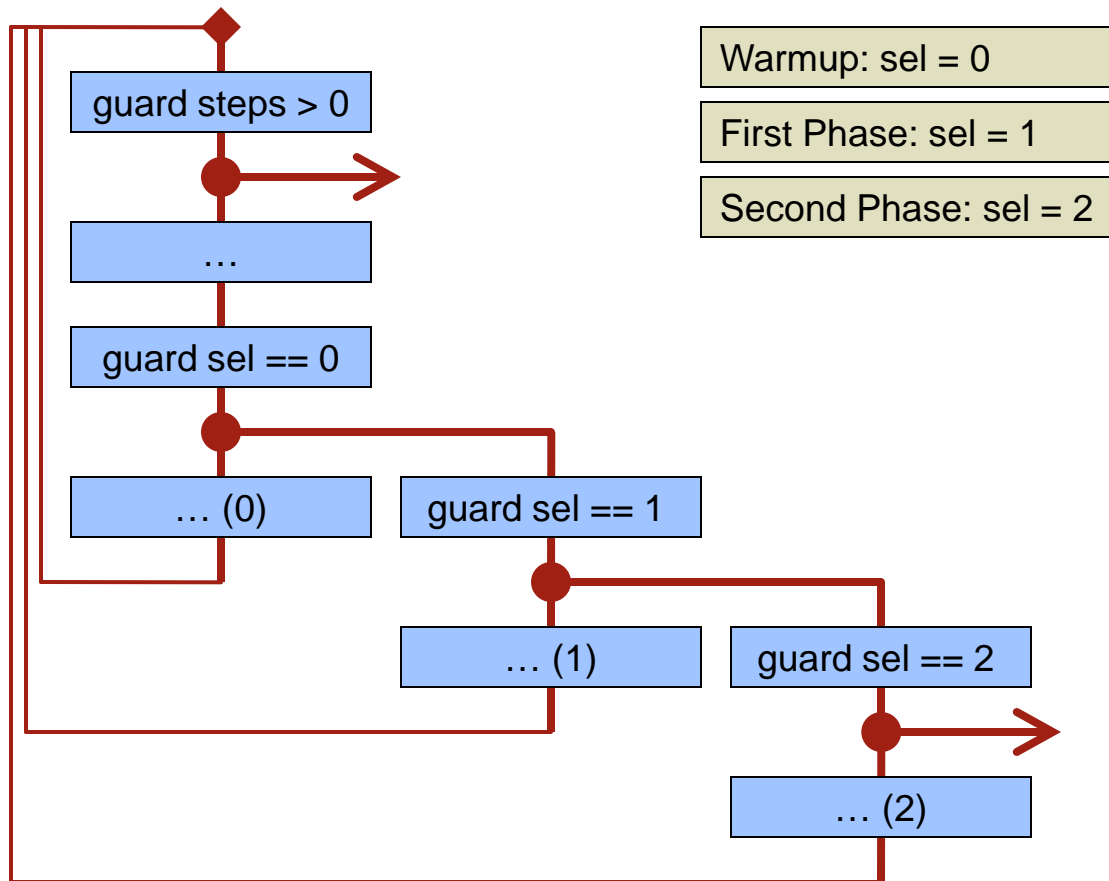
Phase Detection using Trace Compilation



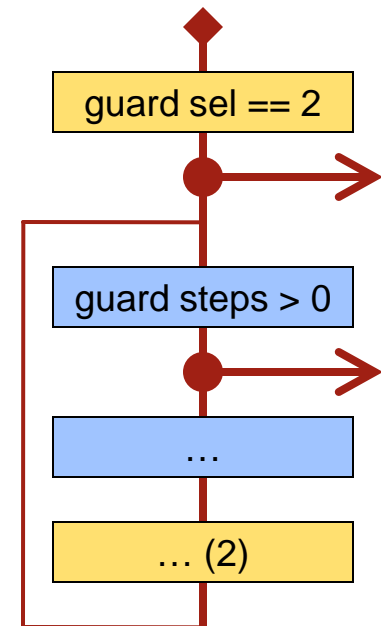


Phases Based on Control Flow

Without Phase Detection

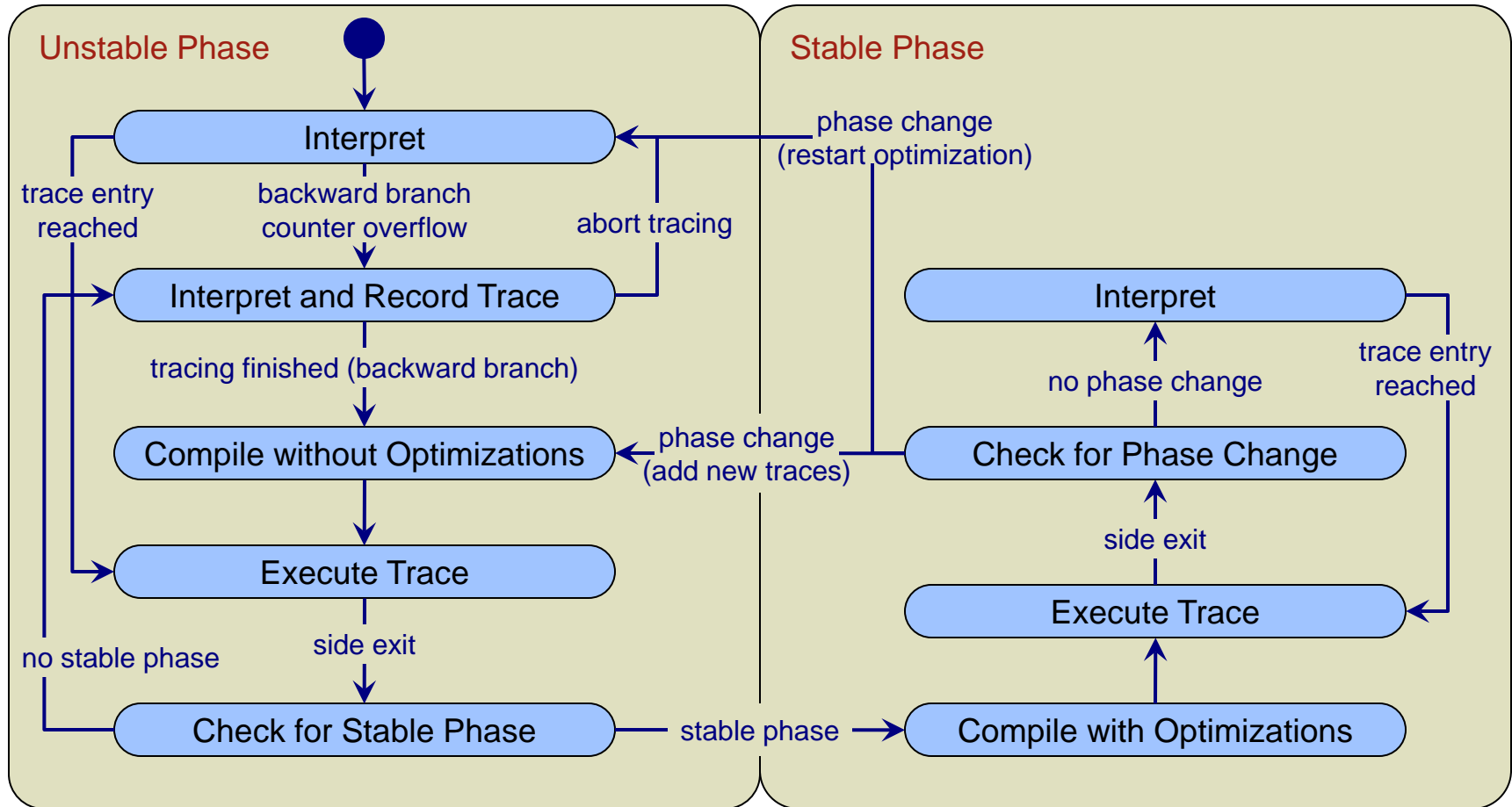


With Phase Detection



Example from Java Grande Series benchmark (simplified and adapted)

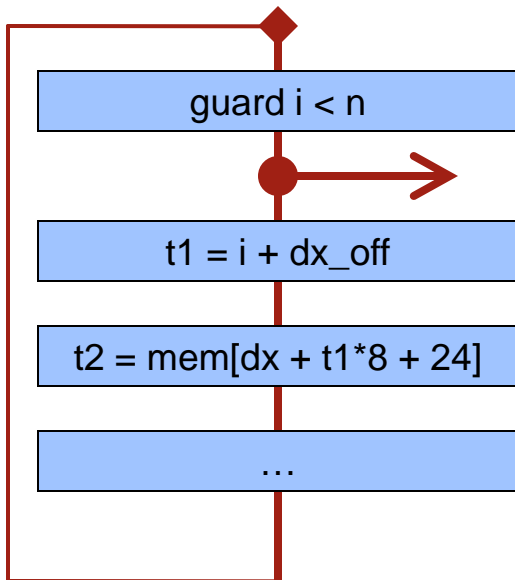
Phase Detection Structure





Phases Based on Data Flow

Without Phase Detection

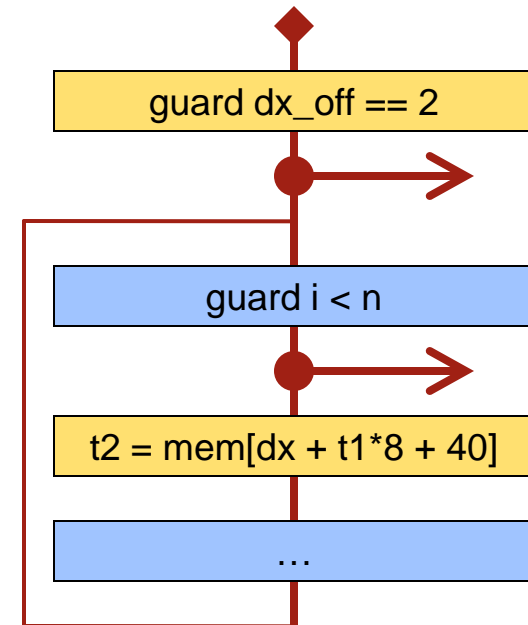


First Phase:
`dx_off = 1`

Second Phase:
`dx_off = 2`

...

With Phase Detection

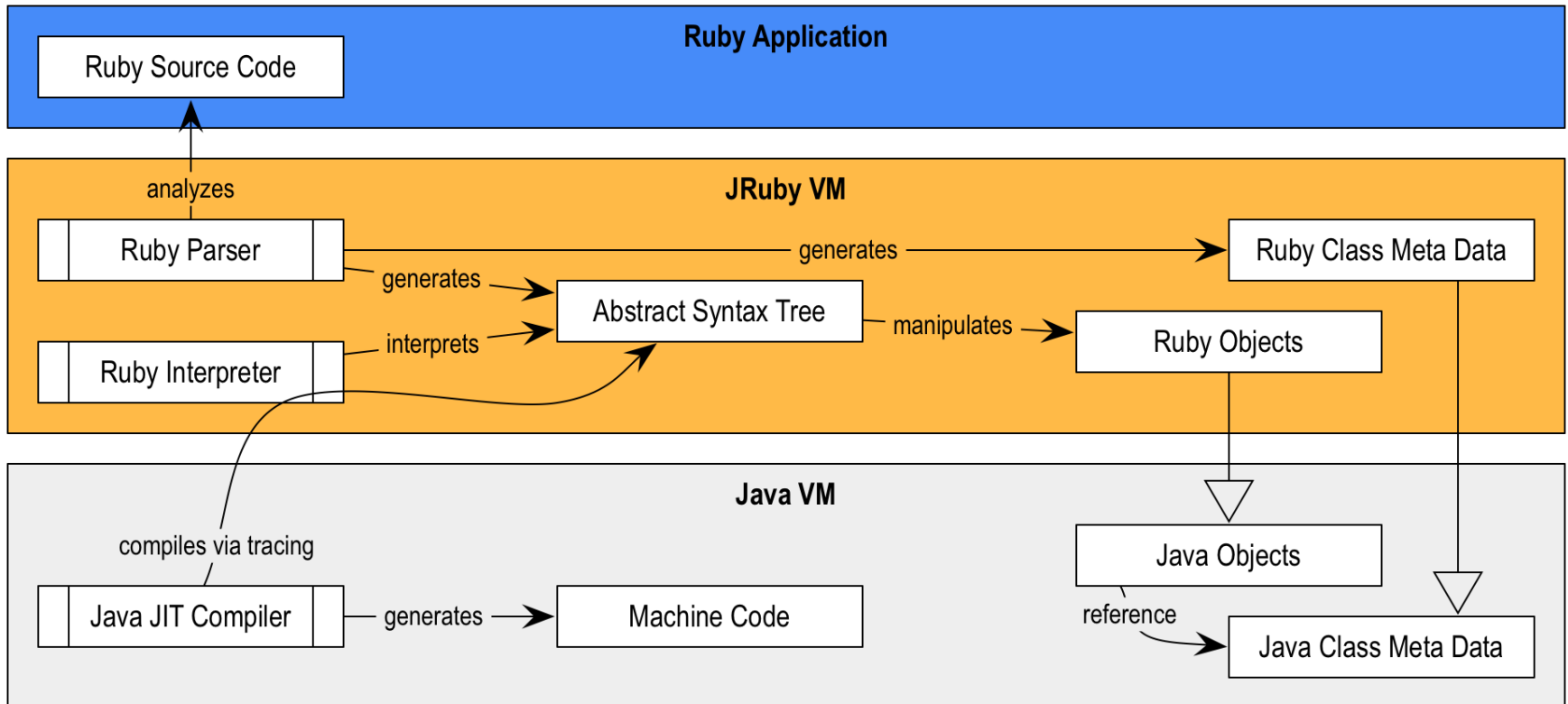


Example from Java Grande LUFact benchmark (simplified)



Possibilities of Trace Compilation

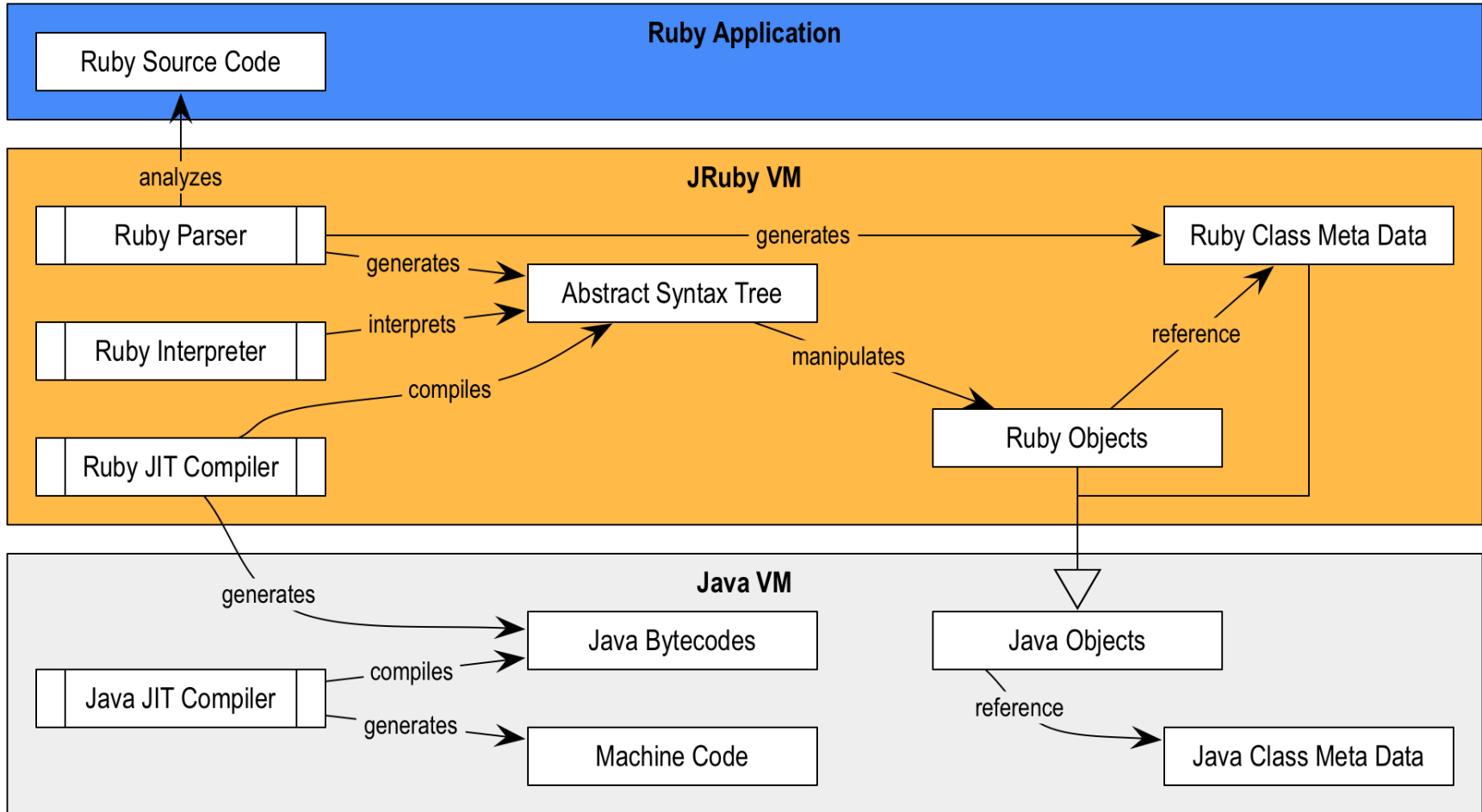
- Hierarchical layering of virtual machines





Possibilities of Trace Compilation

- Hierarchical layering of virtual machines – Current structure



Summary

