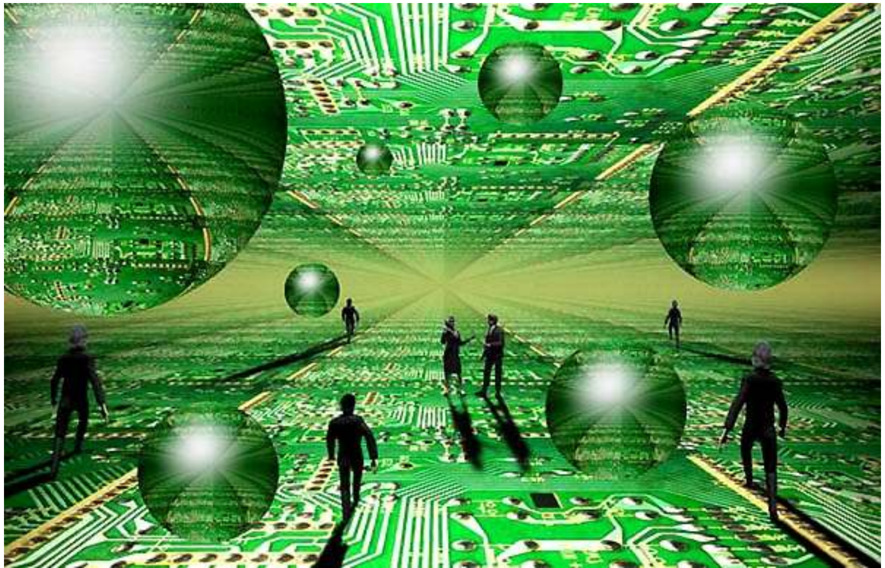
CSCI-GA.3033-015

Virtual Machines: Concepts & Applications

Lecture 7: HLL VM - II

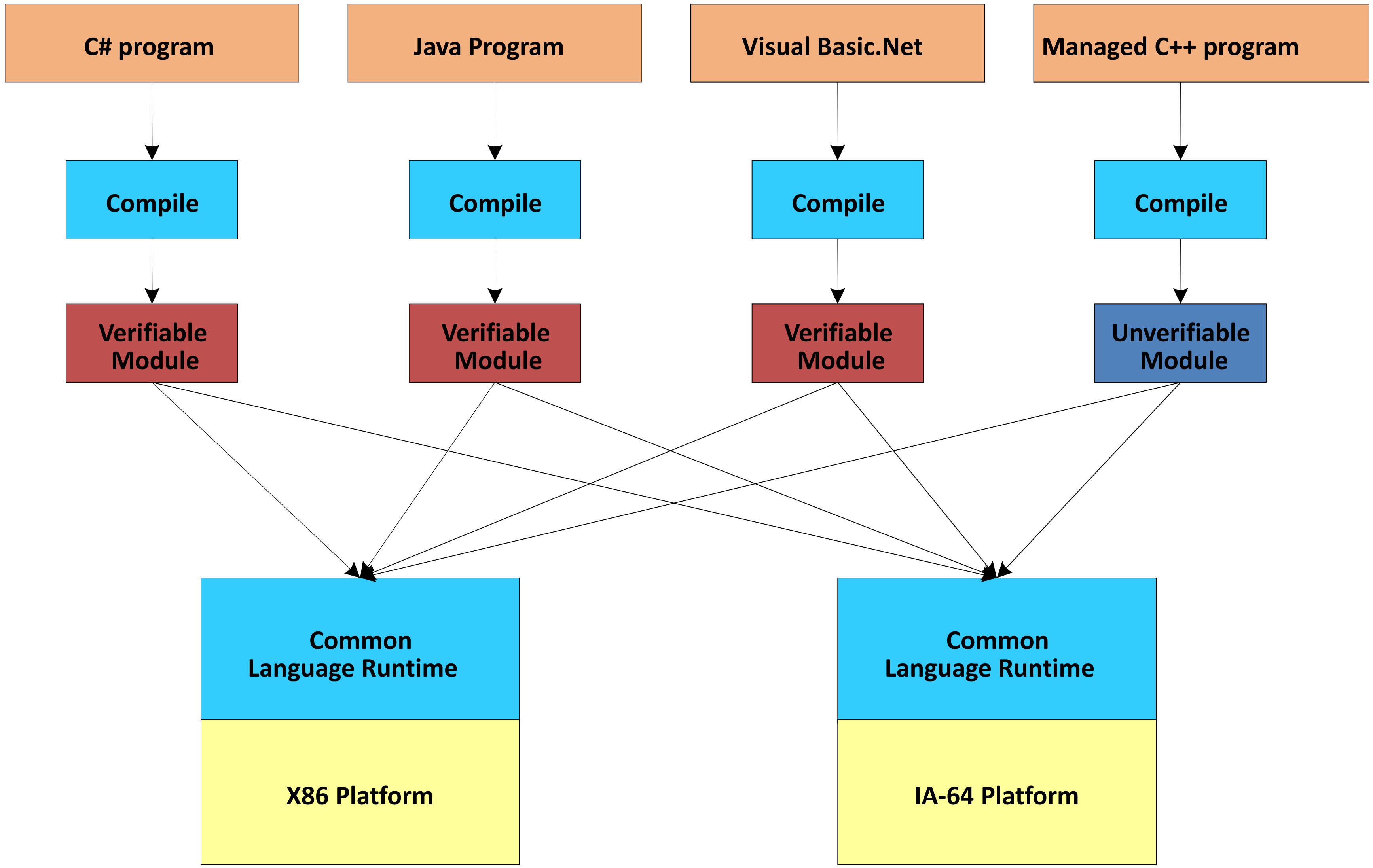
Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com



Microsoft CLI

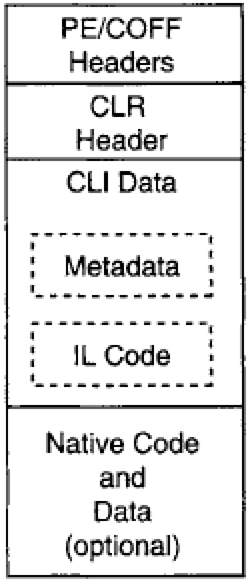
- Common Language Infrastructure
- Part of .NET framework
- Allows multiple HLLs and multiple Platforms
- Common Language Runtime (CLR): MS implementation of CLI
- Strives for HLL independence and platform independence

Microsoft CLI Interoperability



A Module

- The analog of Java binary class
- Contains metadata and code
- Encoded in Microsoft Intermediate Language (MSIL)
- Can be generated by a number of languages
- Programmer can assign attributes to any item in a module.



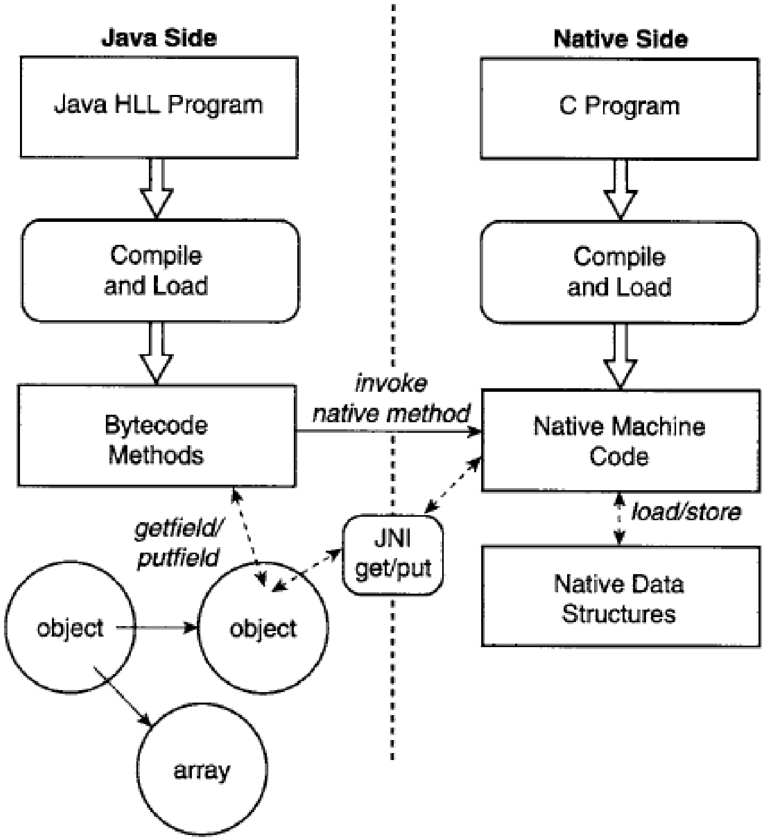
Verifiable Module

- Allows the common language runtime (CLR) to guarantee that the code does not violate current security settings
- CLI allows both verifiable and unverifiable modules (class files)
  - Verifiability is different from validity
  - Unverifiable modules must be trusted by user
  - Verifiable and unverifiable modules can be mixed (but the program becomes unverifiable)

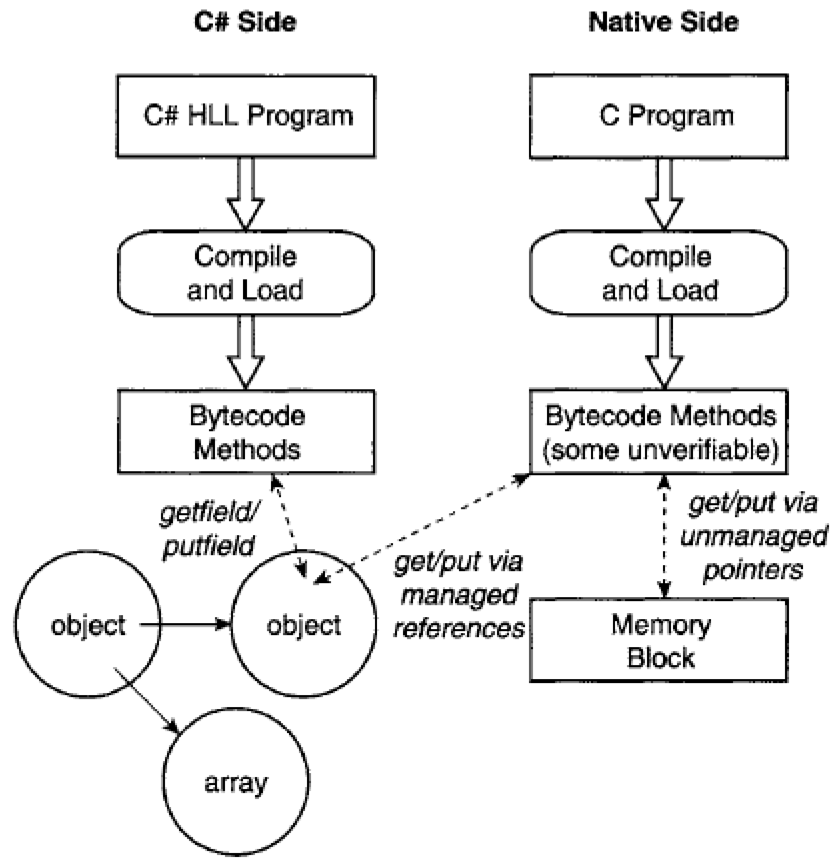
Microsoft CLI and MSIL

- Similar to Java and JVM
  - Object oriented
  - Stack-based ISA
- Some differences
  - Much broader in scope
  - ISA not meant for interpretation
  - Module can be valid (but not verifiable), verifiable, or invalid

Interoperability: Java Vs. CLI



Java

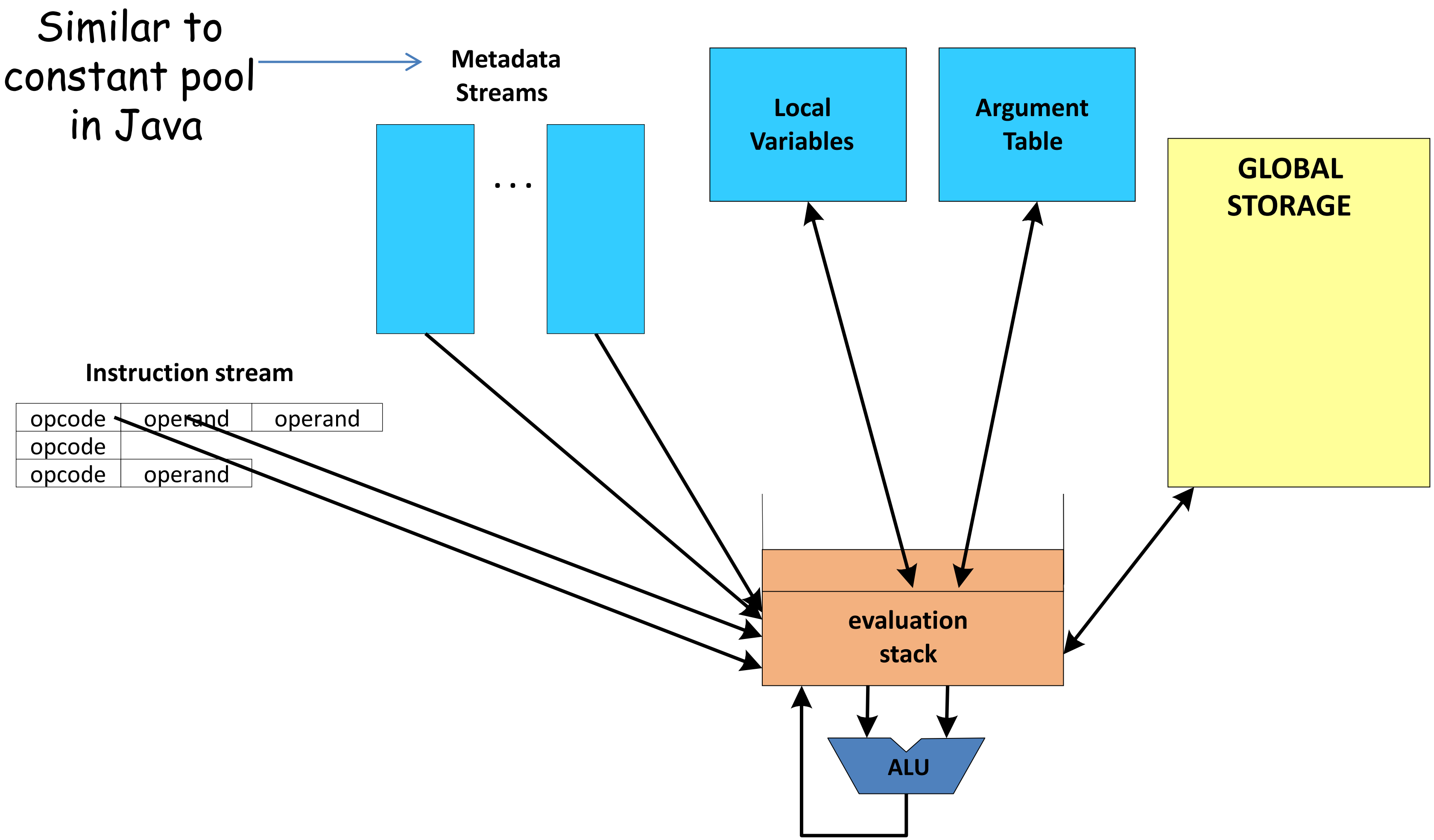


CLI

Microsoft Intermediate Language (MSIL)

- Similar in concept to Java byte codes
- Stack oriented
- Locals and Arguments not part of stack
- Metadata streams hold constant information

MSIL Memory Architecture



For a given method

Metadata Access

- Through tokens
- A token contains 4 bytes
  - one byte: metadata stream identifier
  - the other three: point to a particular entry

Comparison: MSIL & Java bytecodes

- Similar for most memory/ALU instructions
- "Generic" arithmetic instructions in MSIL
  - Better suited for JIT than interpretation (because inferring the type of operands of an instruction takes time in interpretation)

java	MSIL
`0: iconst_2`	`0: ldc.i4.2`
`1: aload_0`	`1: ldarg.0`
`2: getfield #2`	`2: ldobj <token>`
`5: iconst_0`	`5: ldc.i4.0`
`6: iaload`	`6: ldelem.i4`
`7: aload_0`	`7: ldarg.0`
`8: getfield #2`	`8: ldobj <token>`
`11: iconst_1`	`11: ldc.i4.1`
`12: iaload`	`12: ldelem.i4`
`13: iadd`	`13: add`
`14: imul`	`14: mul`
`15: ireturn`	`15: ret`

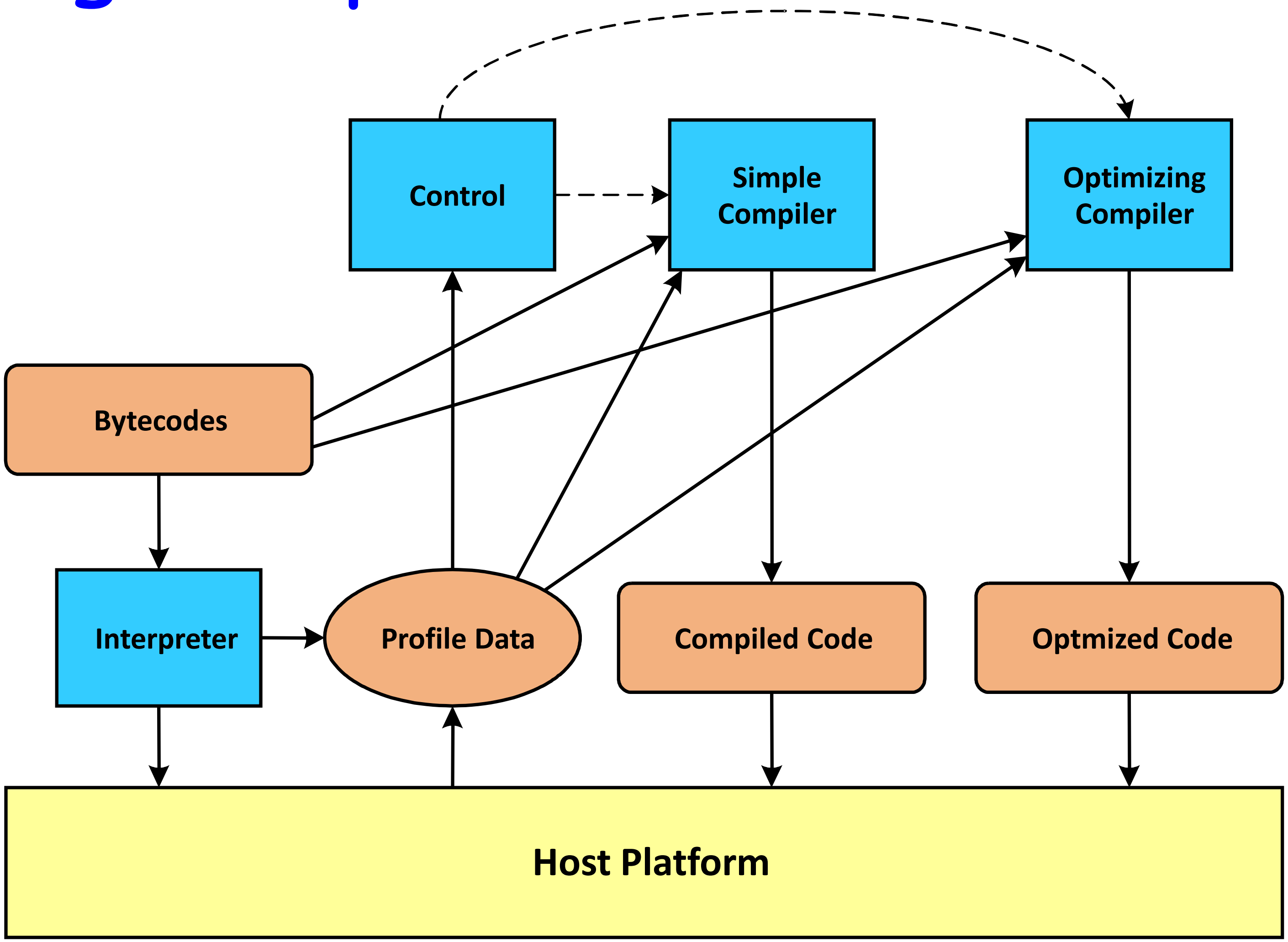
Two Challenges Facing HLL-VMs

1. offset the run-time optimization overhead with the program execution-time improvement.
2. to make an object-oriented program go fast.
   - OO programs typically include frequent use of addressing indirection for both data and code
   - OO programs include frequent use of small methods (which suffer the relatively high overhead of method invocation).

High Performance Optimization

- Staged Optimization Philosophy (again)
  - Faster program startup
  - Compilation spread over time
    - Less noticeable to user
  - Compiling only hot code allows optimization time to be used where needed
  - Consumes less memory for compiled code
  - Waiting longer before optimizing gives better profile information

Staged Optimization Framework



Note: Profiling is often done at method level not basic-block level

Optimizations

- some optimizations are performed:
  - directly via the compiler acting on the bytecode program as input
  - dynamically by the runtime system, apart
    - Example: garbage collection, enhance data locality by reorganizing heap objects, …

Optimizations: Code Re-Layout

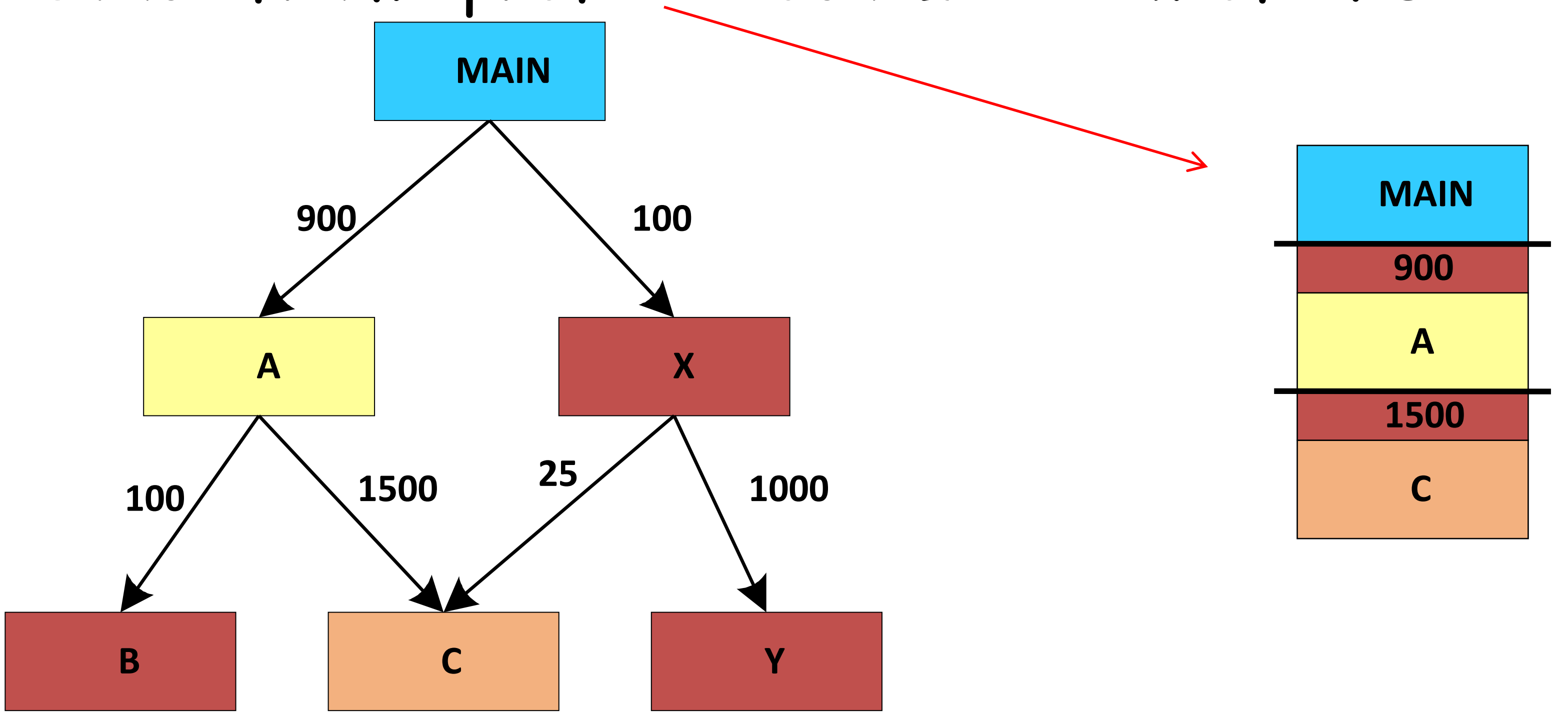
- Code "straightening" as in binary optimization (earlier)
- Code re-layout often provides one of the larger performance benefits among all the optimizations.

Optimizations: Method Inlining

- Benefits:
  - Object-oriented programming tends to encourage many small methods
    - performance can often be improved significantly by avoiding all the overhead code
  - increases the scope over which later code analysis and optimizations can take place
- Drawback: larger binary size
- Method Inlining
  - Small methods: (method size < calling sequence)
    - should always be inlined.
  - Larger methods
    - apply cost-benefit analysis
    - benefit based on profile data

Call Graph Profiling

- Methods are nodes
- Guides method inlining
- Call graph – similar to control flow graph
- Stack frame profile – localized view of CFG



Virtual Function Calls

- Inlining works well if methods are static or final
- How about virtual methods?
- The target of an invokevirtual can change dynamically
  - due to polymorphism
- BUT: often the target does not change
  - use guarded inlining

invokevirtual area

If (target reference == circle) then

inlined code for area of a circle

.

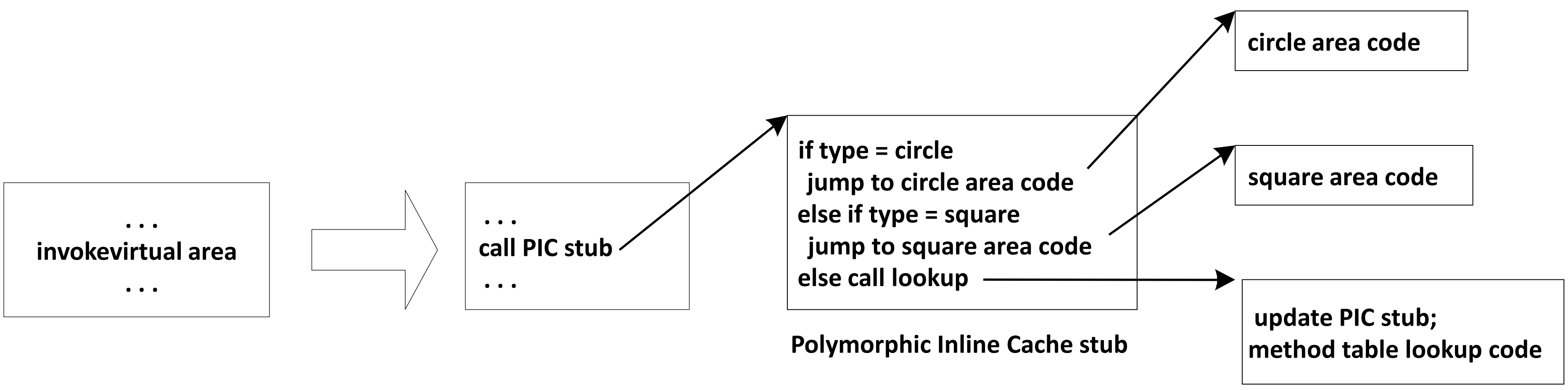
.

.

Else invokevirtual area

Polymorphic Inline Caching

- For use if call is truly polymorphic
  - In OO systems, they are usually implemented with dynamic method table lookup → Can we avoid that?
- Avoids costly method table look-up



Optimizations: Multiversioning and Specialization

- There are two (or more) versions of code, and one version is selected, depending on run-time information, for example, data values or

Suppose that profiling data found that most of A elements are zero

```
for (int i = 0; i < 1000; i++) {

 if (A[i] < 0) B[i] = -A[i]*C[i];
 else B[i] = A[i]*C[i];

}
```

```
for (int i = 0; i < 1000; i++) {

 if (A[i] == 0)

 B[i]= 0;

 if (A[i] < 0) B[i] = -A[i]*C[i];
 else B[i] = A[i]*C[i];

}
```

Optimizations: Deferred Compilation

- Defer compilation of uncommon case until needed

```
for (int i = 0; i < 1000; i++) {

  if (A[i] < 0) B[i] = -A[i]*C[i];
  else B[i] = A[i]*C[i];

}
```

```
for (int i = 0; i < 1000; i++) {

  if (A[i] == 0)

    B[i]= 0;

}
```

Jump to dynamic compiler for deferred compilation

Optimizations: The Stack

- We must differentiate between architected stack and implementation stack
- The contents of these two stacks differ
- The implementation stack contents may depend on the optimization that has been performed

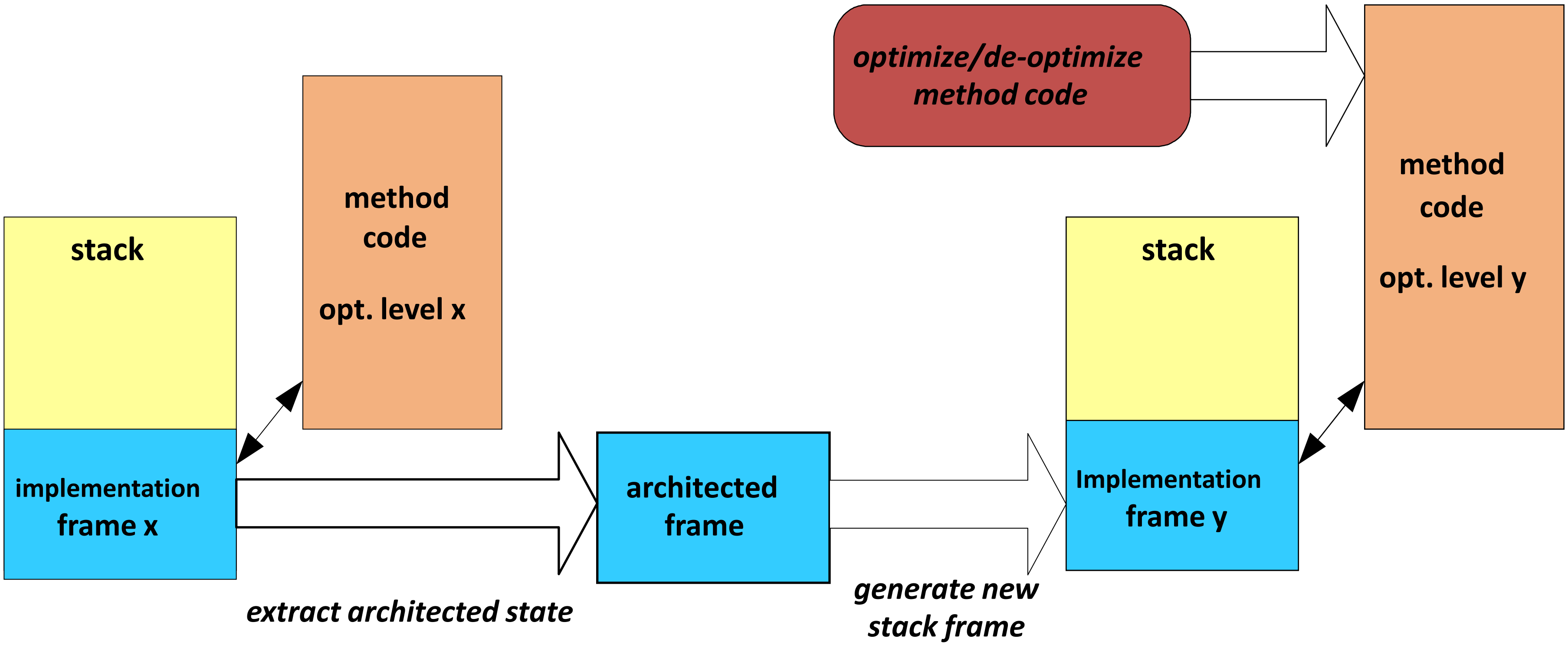
How about dynamic optimizations?

On Stack Replacement

- Some dynamic optimization may require the implementation stack to be modified on the fly
- Optimization (or de-optimization) of currently running method may require changes to stack frame
  - Program dominated by very long-running single loop
    - -- can't wait for next method call
  - Deferred compilation requires immediate replacement
  - Debugging may require de-optimization

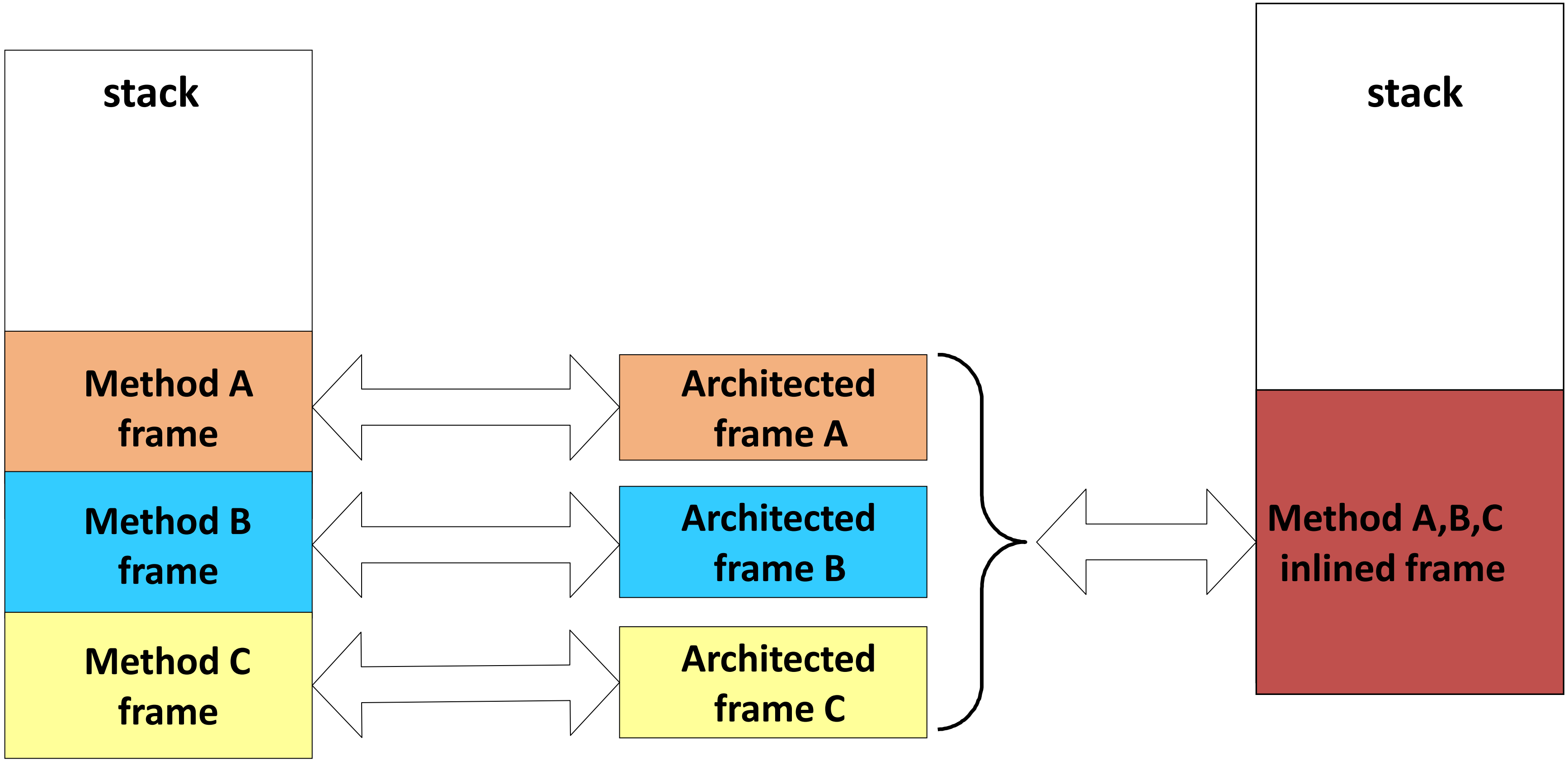
On Stack Replacement

- Steps



On Stack Replacement

- Example: method inlining
  - For optimization or de-optimization



Optimizations: Heap Allocated Objects

- Creating objects and garbage collection have relatively high costs
- Accessing different fields requires several levels of indirections → overhead adds up

Optimizations: Heap Allocated Objects

- Replace object field with scalar
- Requires "escape analysis"
  - No other references outside optimization region

```
class A {
int x;
int y;
}
void foo() {
 A a = new A();
 a.x = 1;
 a.y = a.x + 2;
 System.out.println(a.y);
}
```

```
void foo() {
 int t1 = 1;
 int t2 = t1 + 2;
 System.out.println(t2);
}
```

Optimizations: Heap Allocated Objects

- Replace object field with scalar
- Requires "escape analysis"
  - No other references outside optimization region

```
a = new square;
b = new square;
c = a;
  ...
a.side = 5;

b.side = 10;
z = c.side;
```

```
a = new square;
b = new square;
c = a;
  ...
t1 = 5;
a.side = t1;
b.side = 10;
z = t1;
```

Low Level Optimizations

- Many optimizations similar to conventional binary optmizations
  - dead code removal, copy & constant propagation etc.
- Some are extended to null checks and array range checks
- Example: hoist array range check

```
for (int i = 0; i < j; i++) {
      sum += A[i];   <range check A>
      }
```

```
If (j < A.length)
then for (int i = 0; i < j; i++) {
   sum += A[i];
   }
else for (int i = 0; i < j; i++) {
   sum += A[i];   <range check A>
   }
```

Low Level Optimizations

- Example: redundant null check removal
  - Null check itself isn't costly
    - Use out-of-range address for null
    - Trap will happen automatically
  - *Relaxes precise state constraint*

```
p  := new Z
q  := new Z
r   := p
  . . .
p.x := ...     <null check p>
...  := p.x    <null check p>
  . . .
q.x := ...     <null check q>
  . . .
r.x  := ...    <null check  r (p)>
```

```
p  := new Z
q  := new Z
r   := p
  . . .
p.x := ...     <null check p>
...  := p.x
  . . .
r.x  := ...
q.x := ...     <null check q>
```

Low Level Optimizations

- Loop peeling

```
for (int i = 0; i < 100; i++) {
      r = A[i];
      B[i] = r*2;
      p.x += A[i] ;   <null check p>
 }
```

⟹

```
r = A[0];
B[0] = r*2;
p.x = A[0];   <null check p>
for (int i = 1; i < 100; i++) {
      r =A[i]
      p.x += A[i] ;
      B[i] = r*2;
 }
```

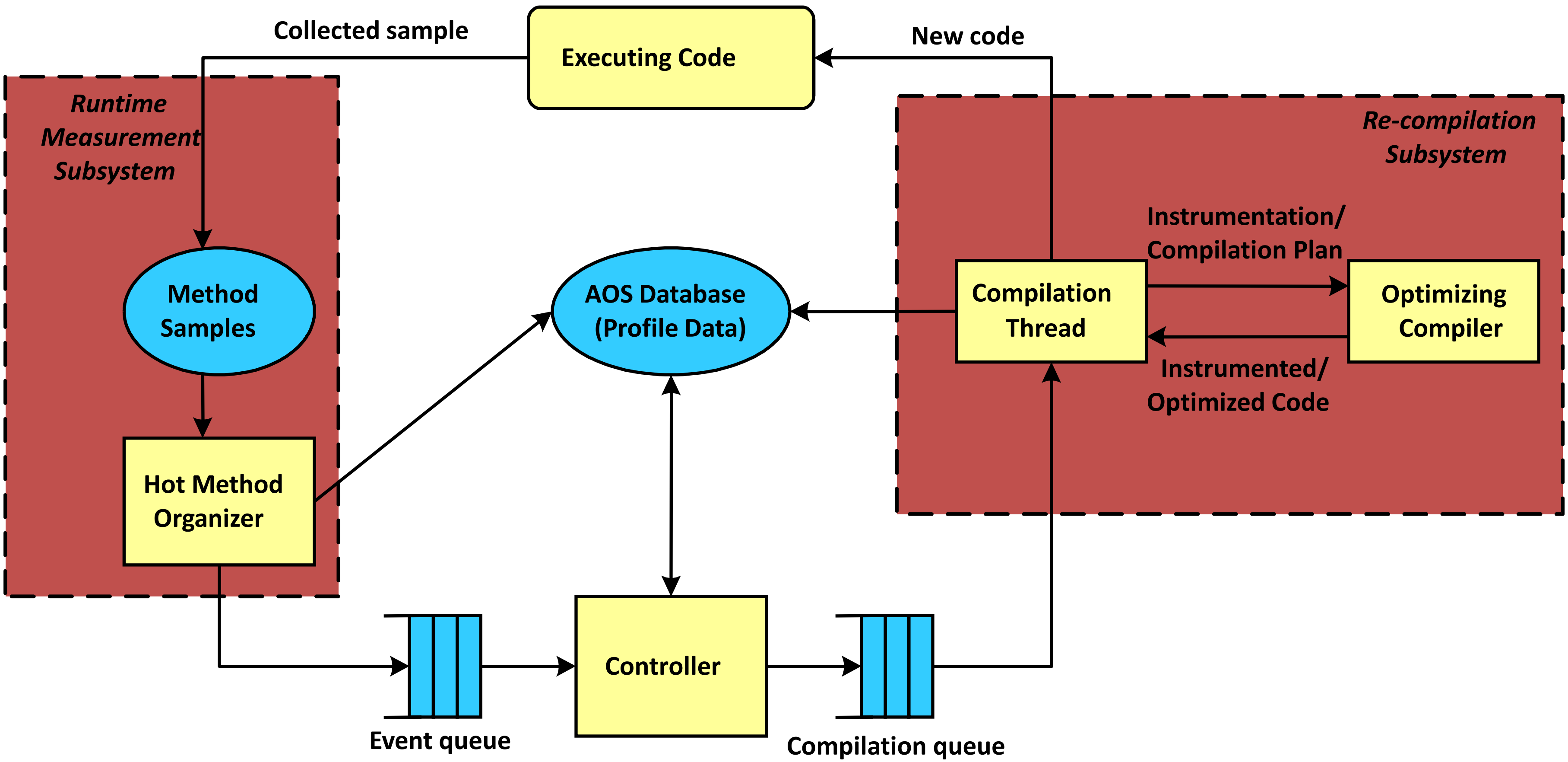
NULL checks in the loop body can be eliminated

Case Study: IBM Jikes (Jalapeno) JVM

- Dynamic Compiler developed at IBM Research
- Uses Compile-Only strategy (no interpretation)
  - Baseline compiler
    - Straight translation to native code
    - Simulates operand stack; no register allocation
  - Optimizing compiler
    - Translates to intermediate representation
    - Performs simple register allocation
    - Three levels of optimization

Compiler	*Bytecode Bytes /Millisecond*
Baseline	**274.14**
Opt. Level 0	**8.77**
Opt. Level 1	**3.59**
Opt. Level 2	**2.07**

Big Picture



Collected sample
Executing Code
New code
Runtime
Measurement
Subsystem
Re-compilation
Subsystem
Instrumentation/
Compilation Plan
Method
Samples
AOS Database
(Profile Data)
Compilation
Thread
Optimizing
Compiler
Instrumented/
Optimized Code
Hot Method
Organizer
Controller
Event queue
Compilation queue

Runtime Measurement Subsystem

- Gathers raw performance data
  - via software instrumentation
    - or hardware performance counters
  - Sampling done at thread switch time
  - Current method at time of switch is sampled
- Summarizes information
  - via organizer threads
- Passes summary to the controller or AOS database

Controller

- Coordinates activities of runtime measurement system and recompilation system
- Can instruct measurement system to continue or change profiling strategy
  - Can direct recompilation system to insert/remove profile code
- Constructs compilation plans using profile data
  - Uses analytical cost-benefit model
  - Sends plan to recompilation subsystem

Recompilation Subsystem

- Contains compilation threads
  - Takes place concurrent with execution
  - Uses optimization plan generated by controller
    - Which optimizations
    - Profile info for feedback-directed optimizations
    - Instrumentation to be inserted

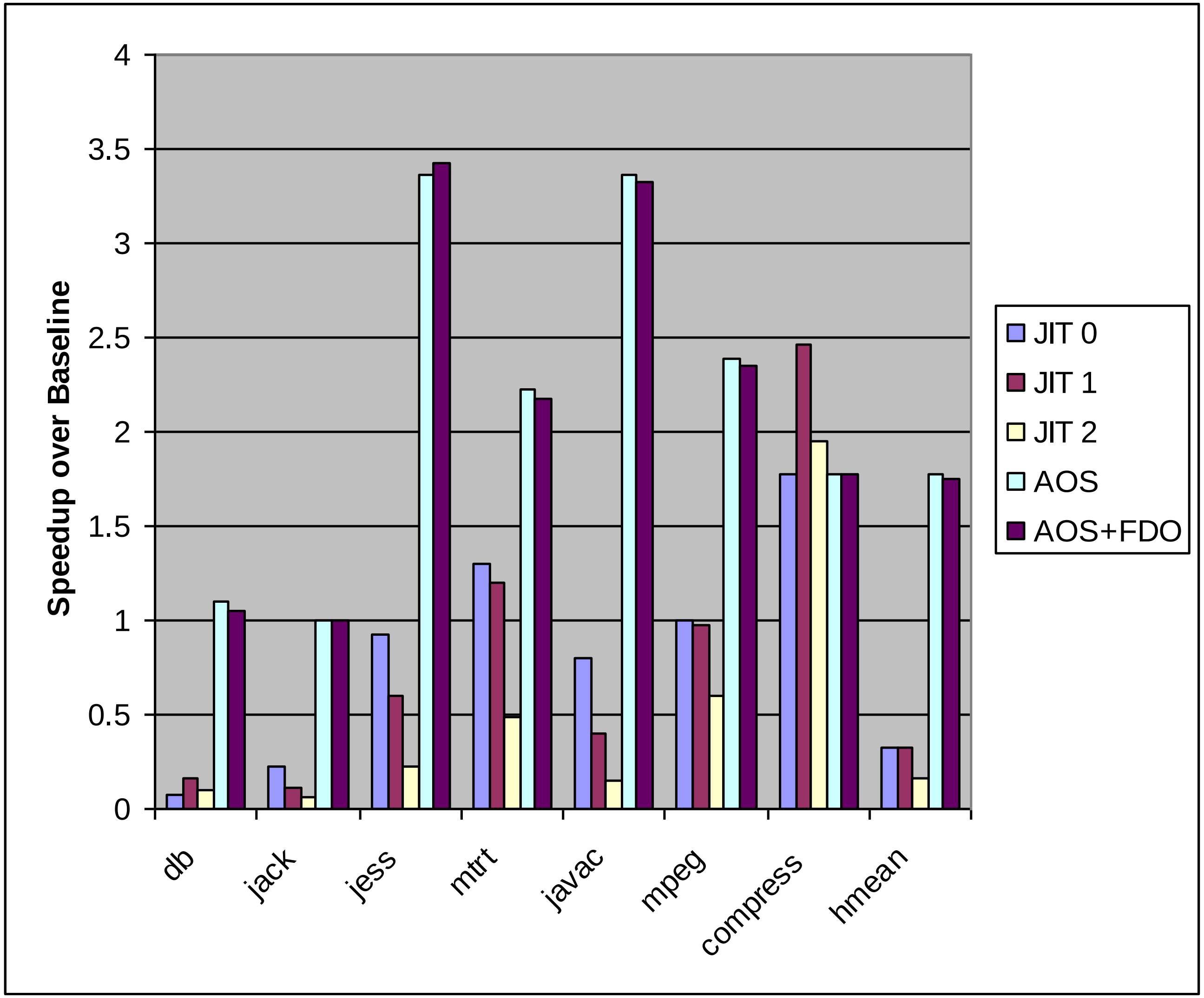
Feedback-Directed Inlining

- Build dynamic call graph using samples
- Identify hot edges via sample mechanism
  - *Edge listener* walks call graph to find hot edges
- Samples passed to *dynamic call graph organizer*
- Dynamic call graph organizer periodically invokes *adaptive inline organizer*
- *Adaptive inline organizer*
  - Identifies candidate methods
  - By considering edges that exceed hotness threshold
- Controller estimates *boost factor* and applies cost/benefit model

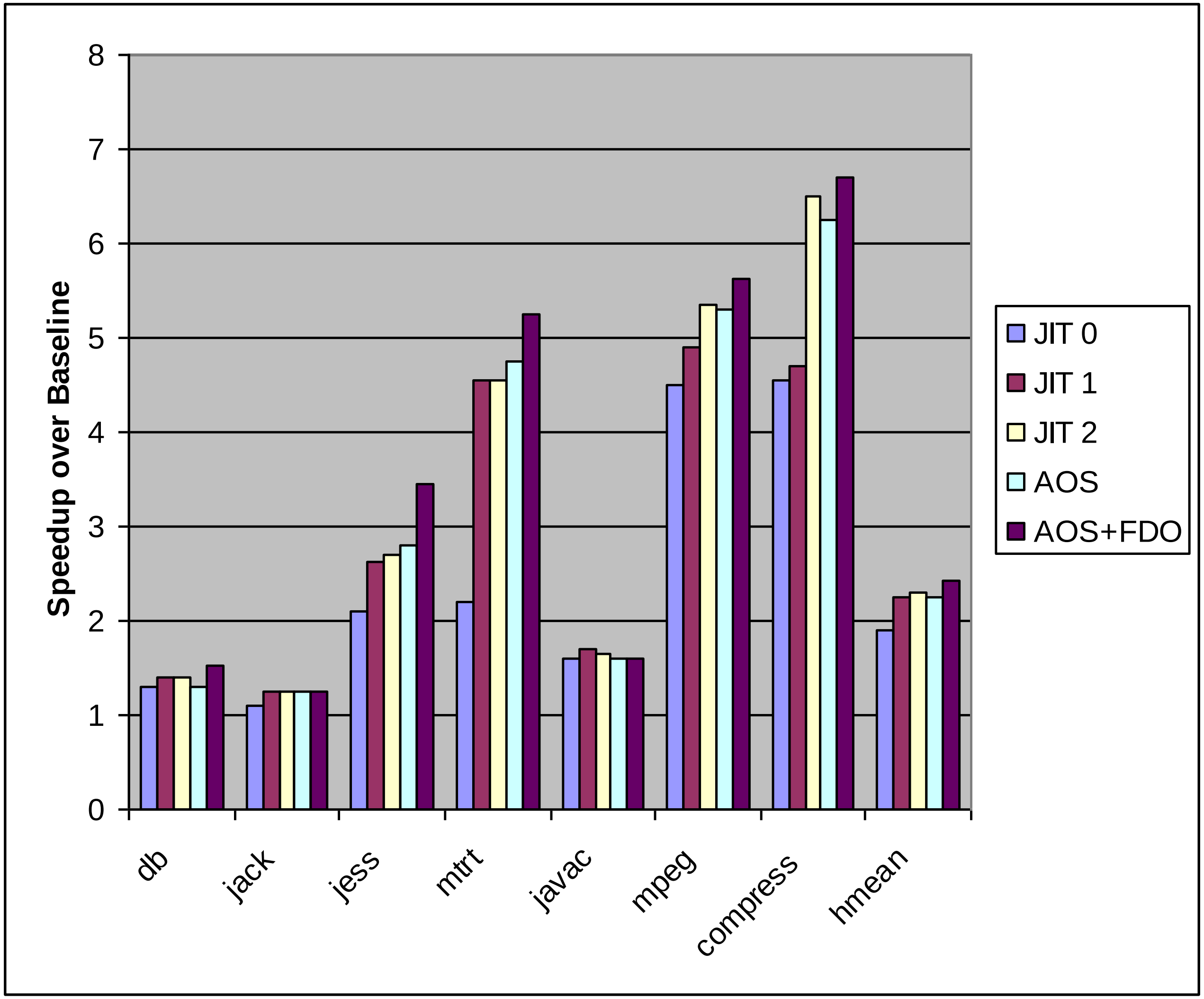
Compilers studied

- Baseline as a JIT
- Level 0 optimization as a JIT
- Level 1 optimization as a JIT
- Level 2 optimization as a JIT
- Adaptive multi-level
  - Method oriented
- Adaptive + Feedback Directed Optimization
  - Code region oriented; more focused

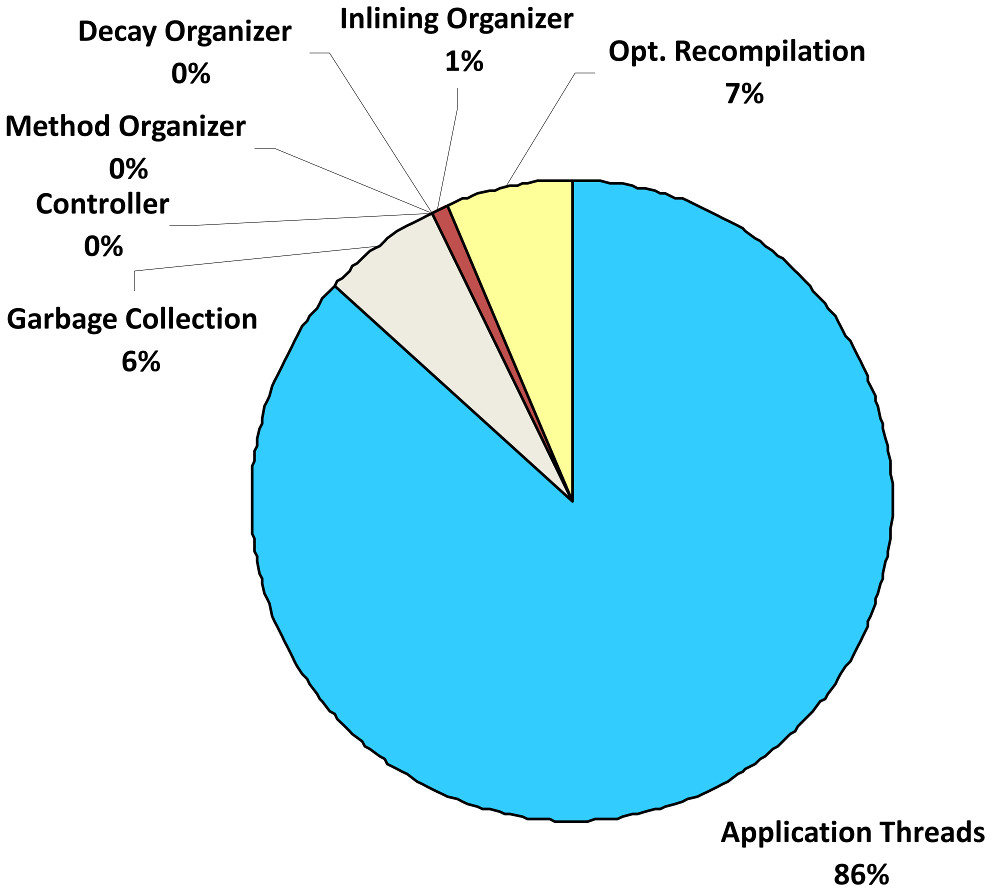
Startup Performance



Steady State Performance



Results: Overhead



Conclusions: HLL VMs vs. Process VMs

- Memory architecture
  - Object model is less implementation-dependent
    - No compatibility problems due to size limitations/differences
- Memory protection
  - Pointers very carefully controlled
    - No rogue load/stores
- Precise Exceptions
  - Exception checking is explicit (no masks)
  - Operand stack imprecise within a method
  - Locals imprecise if exception goes to higher level

Conclusions: HLL VMs vs. Process VMs

- Instruction set dependences
  - No registers
  - No condition codes
- Code discovery
  - Restricted, explicit control flow
  - All code can be discovered at method entry
- Self Modify-Referencing Code
  - Simply doesn't exist