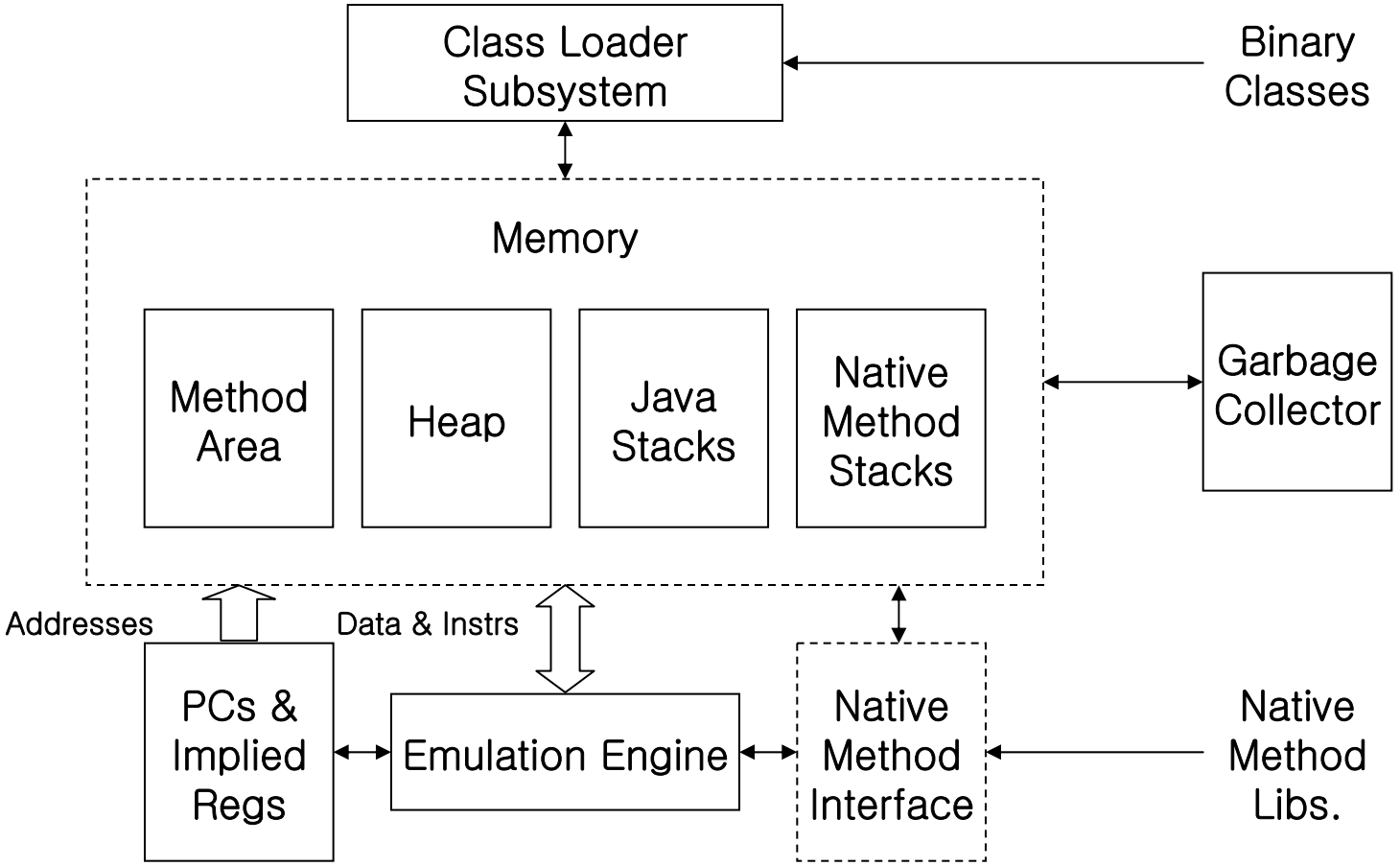
HLL VM Implementation

Contents

- Typical JVM implementation
- Dynamic class loading
- Basic Emulation
- High-performance Emulation
  - Optimization Framework
  - Optimizations

Typical JVM implementation



Typical JVM Major Components

- Class loader subsystem
- Memory system
  - Including garbage-collected heap
- Emulation engine

Class loader subsystem

- Convert the class file into an implementation-dependent memory image
- Find binary classes
- Verify correctness and consistency of binary classes
- Part of the security system

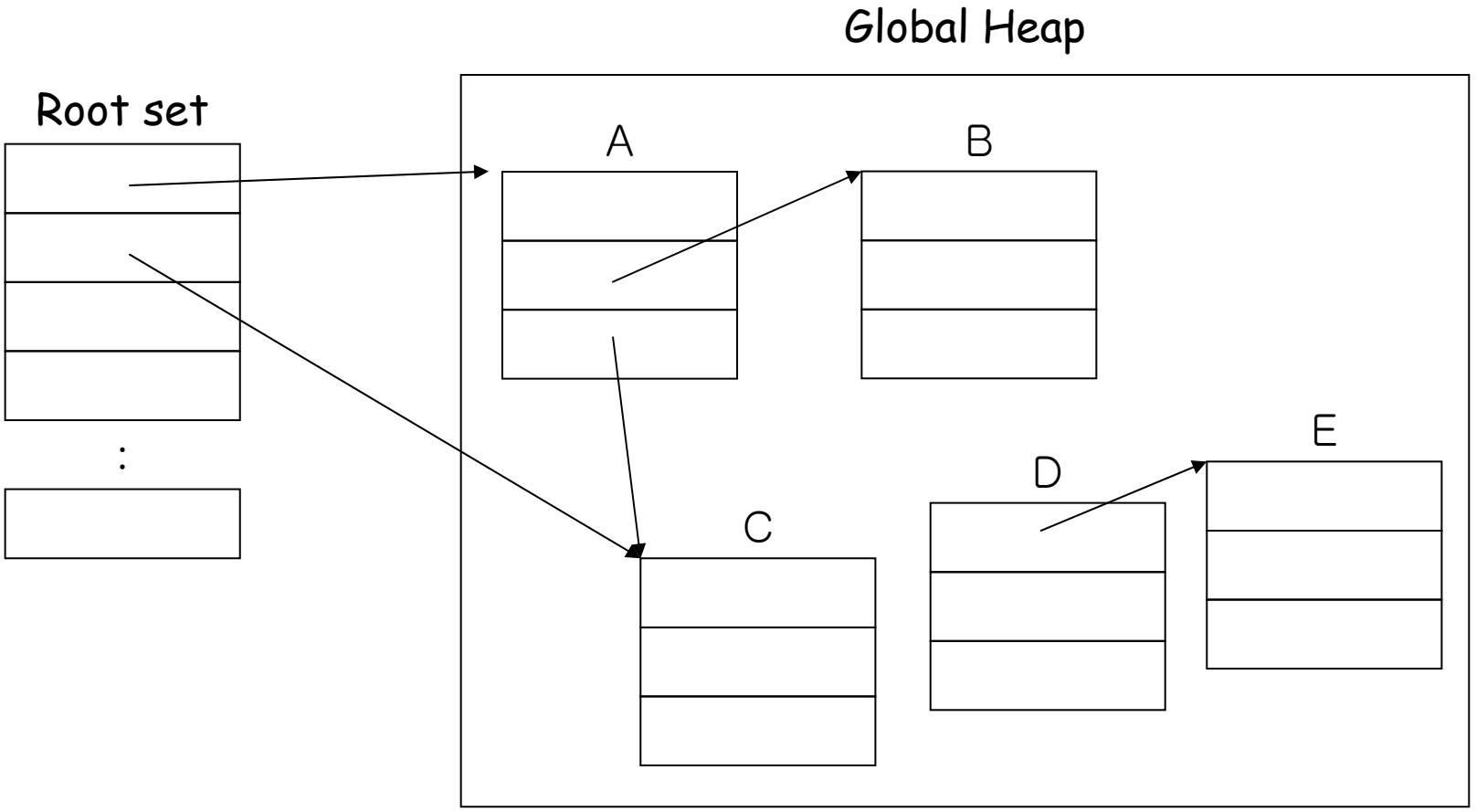
Dynamic class loading

- Locate the requested binary class
- Check the integrity
  - Make sure the stack values can be tracked statically
  - Static type checking
  - Static branch checking
- Perform any translation of code and metadata
  - Check the class file format
  - Check arguments between caller and callee
  - Resolve fully qualified references

Garbage Collection

- Garbage : objects that are no longer accessible
- Collection: memory reuse for new objects
- Root set
  - Set of references point to objects held in the heap
  - Garbage
    - Cannot be reached through a sequence of references beginning with the root set
- When GC occurs, need to check all accessible objects from the root set and reclaim garbages

Root Set and the Heap



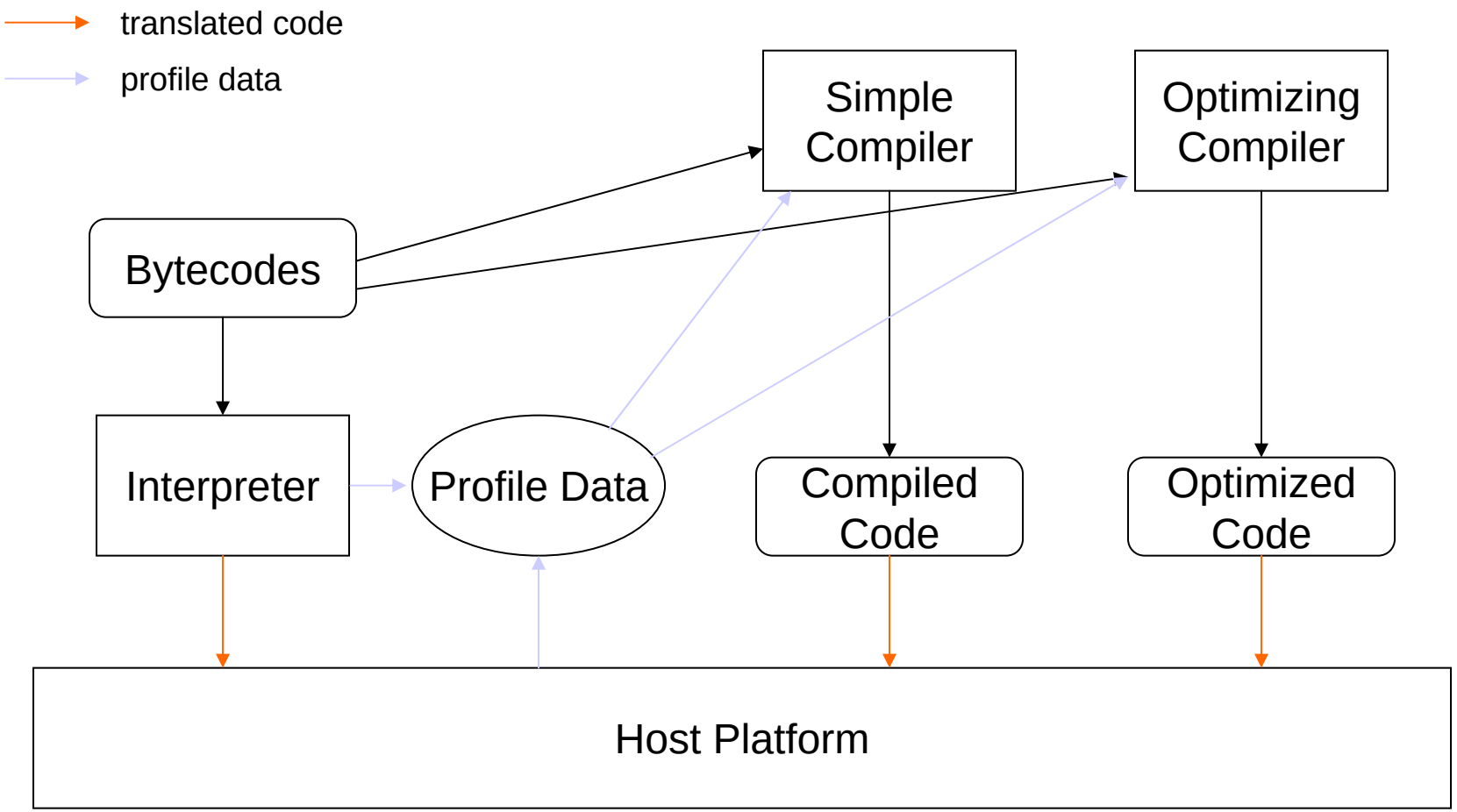
Garbage Collection Algorithms

- Mark-and-Sweep
- Compacting
- Copying
- Generational

Basic Emulation

- The emulation engine in a JVM can be implemented in a number of ways.
  - interpretation
  - just-in-time compilation
- More efficient strategy
  - apply optimizations selectively to hot spot
- Examples
  - from interpretation : Sun HotSpot, IBM DK
  - from compilation : Jikes RVM

Optimization Framework



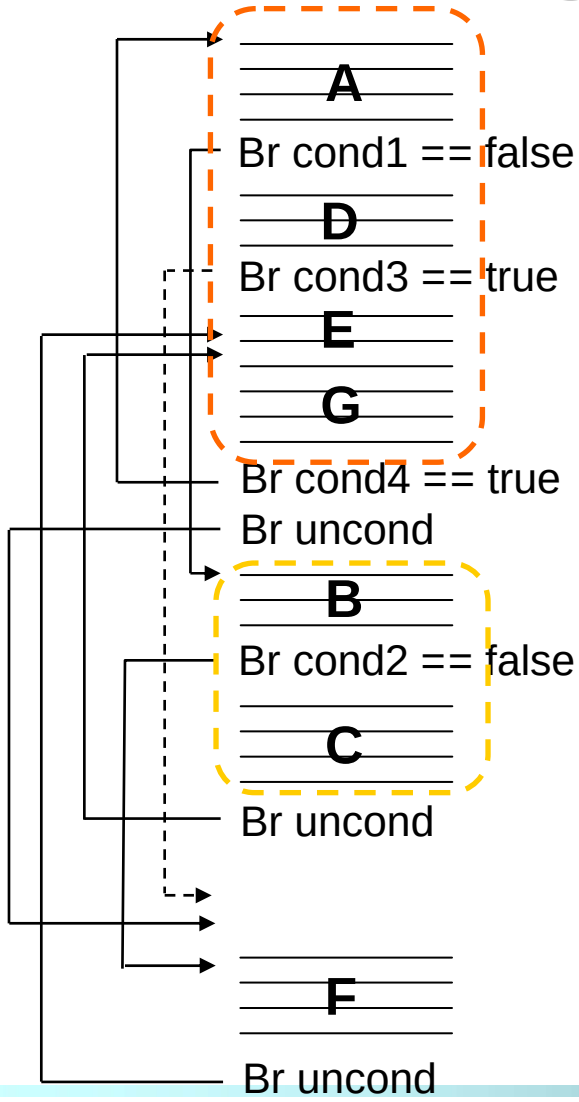
High-performance Emulation

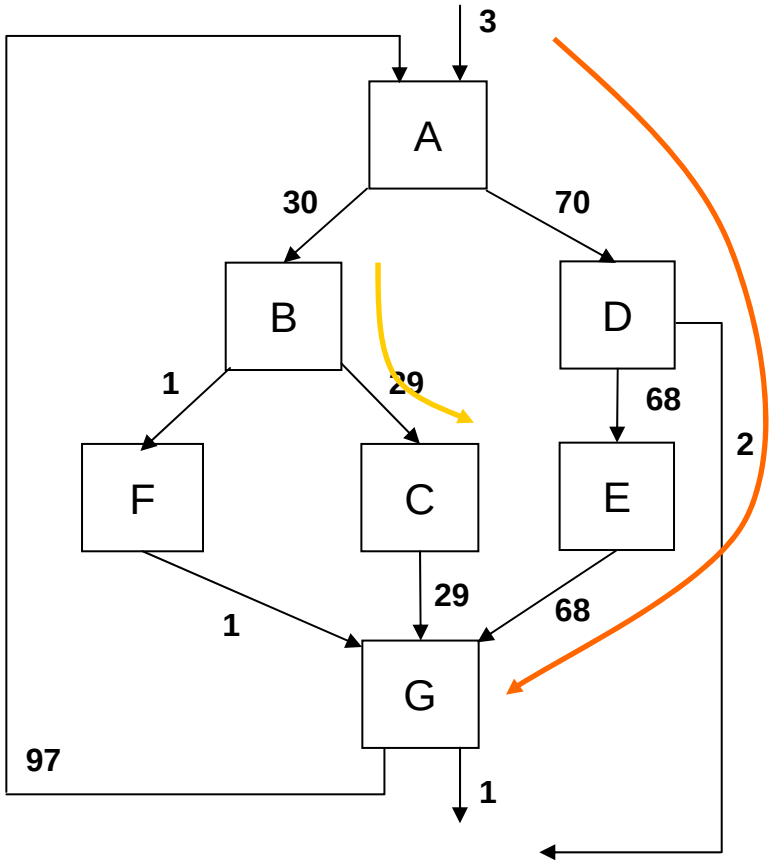
- Code Relayout
- Method Inlining
- Optimizing Virtual Method Calls
- Multiversioning and Specialization
- On-Stack Replacement
- Optimization of Heap-Allocated Objects
- Low-Level Optimizations
- Optimizing Garbage Collection

Optimization
Code Relayout

- the most commonly followed control flow paths are in contiguous location in memory
- improved locality and conditional branch predictability

FLASHBACK
Code Relayout





Optimization
Method Inlining

- Two main effects
  - calling overheads decrease.
    - passing parameters
    - managing stack frame
    - control transfer
  - code analysis scope expands.
    → more optimizations are applicable.
- effects can be different by method's size
  - small method : beneficial in most of cases
  - large method : sophisticated cost-benefit analysis is needed
    → *code explosion* can occur
    : poor cache behavior, performance losses

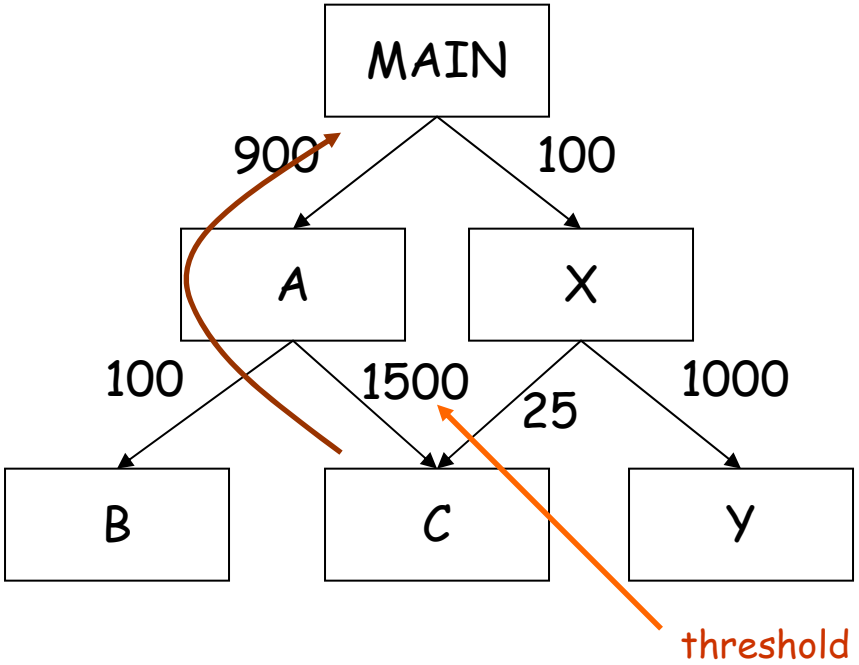
Optimization
Method Inlining (cont'd)

- General processing sequence

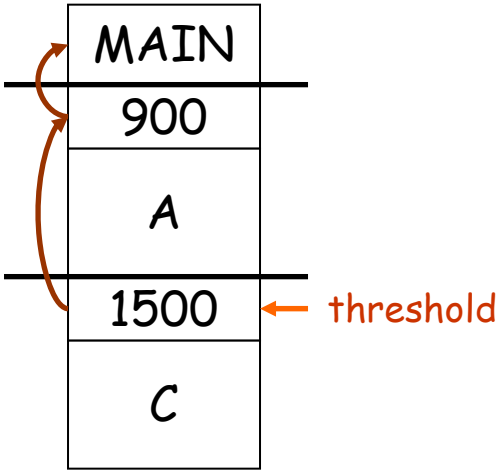
  1. profiling by instrument
  2. constructing call-graph at certain intervals
  3. if # of call exceeds the threshold, invoke dynamic optimization system

- To reduce analysis overhead
  - profile counters are included in one's stack frame.
  - Once meet the threshold, "walk" backward through the stack

Optimization
Method Inlining (cont'd)



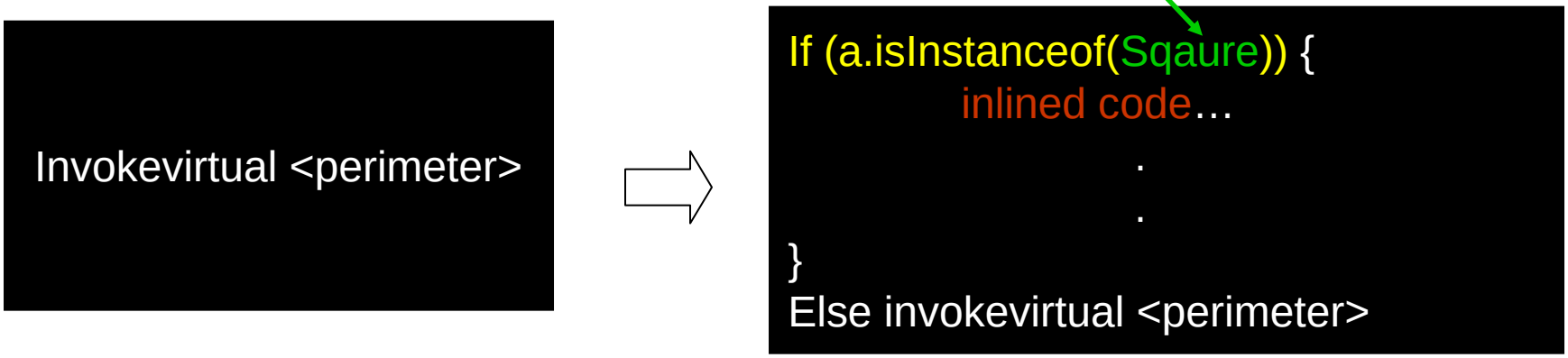
using call-graph



via stack frame

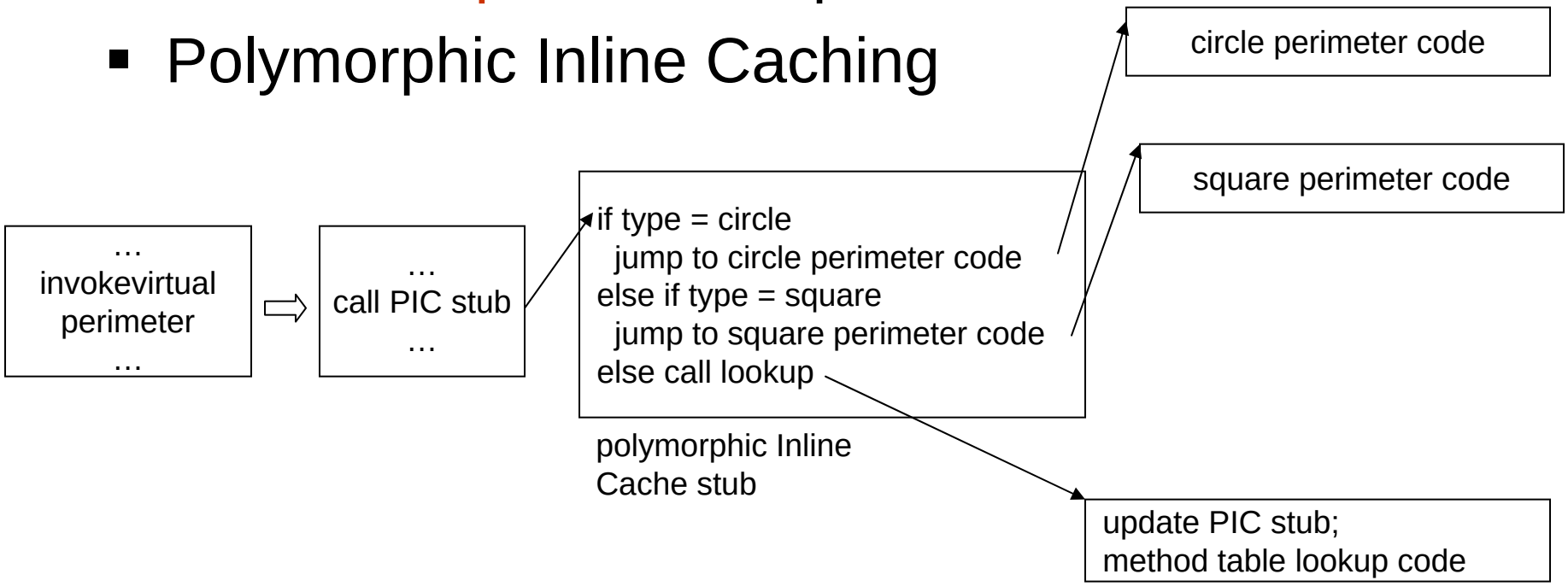
Optimization
Optimizing Virtual Method Calls

- What if the method code called changes?
  - Which code should be inlined?
    → we always deal "the most common case".
  - Determination of which code to use is done at run time via a dynamic method table lookup.



Optimization
Optimizing Virtual Method Calls (cont'd)

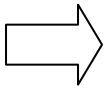
- If inlining is not useful, just removing method table lookup is also helpful
- Polymorphic Inline Caching



Optimization
Multiversioning and Specialization

- Multiversioning by specialization
  - If some variables or references are always assigned data values or types known to be constant (or from a limited range), then simplified, specialized code can sometimes be used in place of more complex, general code.

```
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;
}
```

general code

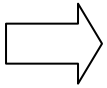
```
if(A[i]<0) B[i] = -A[i]*C[i];
else B[i] = A[i]*C[i];
```

specialized code

Optimization
Multiversioning and Specialization (cont'd)

- compile only a single code version and to defer compilation of the general case

```
for (int i=0;i<1000;i++) {
  if(A[i]<0) B[i] = -A[i]*C[i];
  else B[i] = A[i]*C[i];
}
```



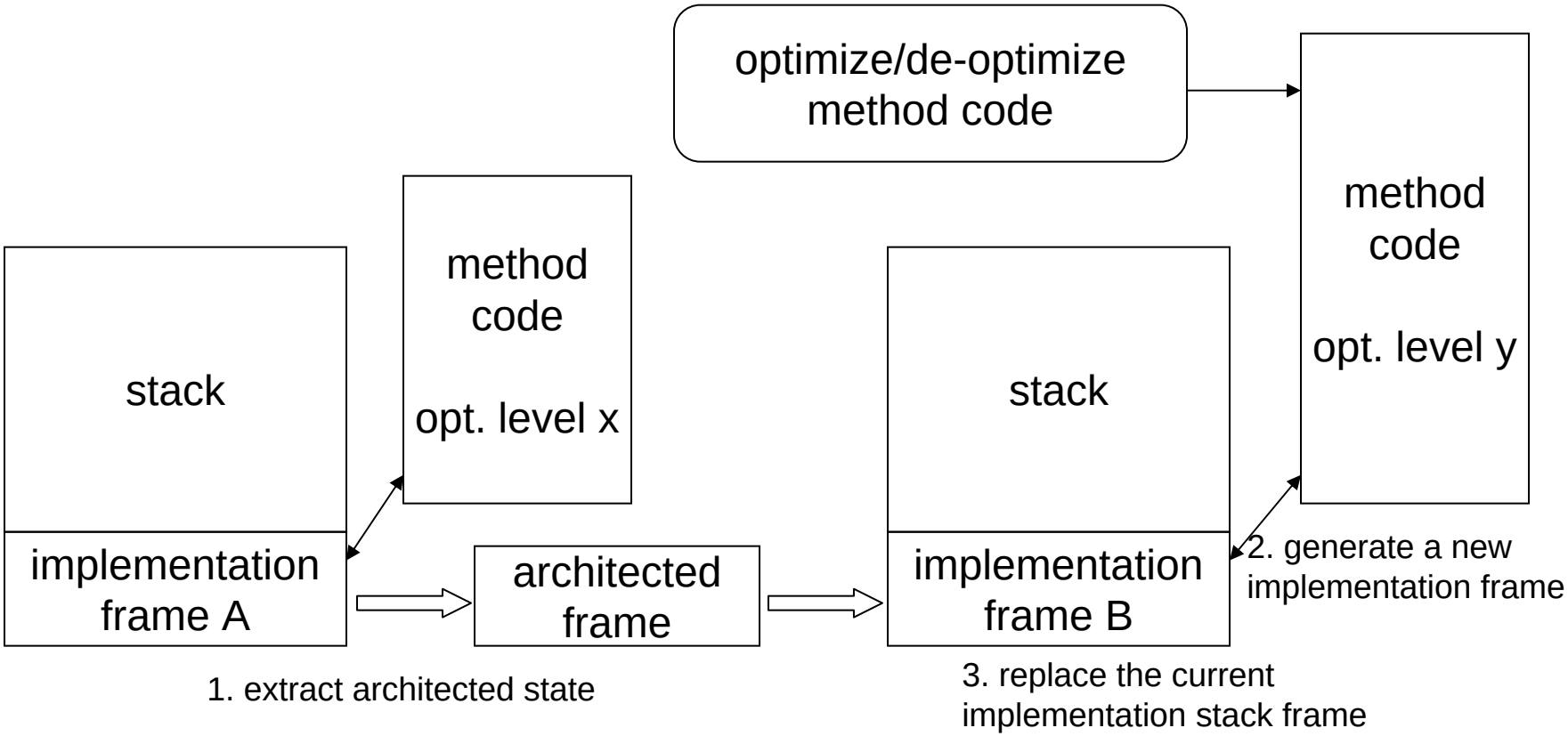
```
for (int i=0;i<1000;i++) {
  if (A[i] ==0 )
    B[i] = 0;
}
```

jump to dynamic compiler for deferred compilation

Optimization
On-Stack Replacement

- When do we need on-stack replacement?
  - After inlining, want executing inlined version right away
  - Currently-executing method is detected as a hot spot
  - Deferred compilation occurs
  - Debugging needs un-optimized version of code
- Implementation stack needs to be modified on the fly to dynamically changing optimizations
  - E.g., inlining: merge stacks into a single stack
  - E.g., JIT compilation: change stack and register map
- Complicated, but sometimes useful optimization

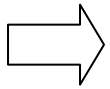
Optimization
On-Stack Replacement (cont'd)



Optimization
Optimization of Heap-Allocated Objects

- the code for the heap allocation and object initialization can be inlined for frequently allocated objects
- scalar replacement
  - escape analysis, that is, an analysis to make sure all references to the object are within the region of code containing the optimization.

```
class square {
int side;
int area;
}
void calculate() {
  a = new square();
  a.side = 3;
  a.area = a.side * a.side;
  System.out.println(a.area);
}
```



```
void calculate() {
  int t1 = 3;
  int t2 = t1 * t1;
  System.out.println(t2);
}
```

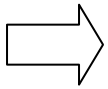
Optimization
Optimization of Heap-Allocated Objects(cont'd)

- field ordering for data usage patterns
  - to improve D-cache performance
  - to remove redundant object accesses

redundant getfield (load) removal

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

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

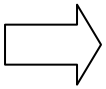


```
a = new square;
b = new square;
c = a;
…
t1 = 5;
a.side = t1;
b.side = 10
z = t1;
```

Optimization
Low-Level Optimizations

- Array range and null reference checking may incur two drawbacks.
  - checking overhead itself
  - Disable some optimizations for a potential exception thrown

```
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)>
```

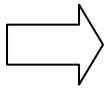


```
p = new Z
q = new Z
r = p
 …
p.x = …          <null check p>
… = p.x
 …
r.x = …
q.x = …          <null check q>
```

Optimization
Low-Level Optimizations (cont'd)

- Hoisting an Invariant Check
  - checking can be hoisted outside the loop

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

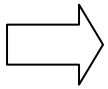


```
if (j < A.length)
then for (int i=0;i<j;i++) {
  sum += A[i];
}
else for (int i=0;i<j;i++) {
  sum += A[i];       <range check A>
}
```

Optimization
Low-Level Optimizations (cont'd)

- Loop Peeling
  - the null check is not needed for the remaining loop iterations.

```
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;
}
```

Optimization
Optimizing Garbage Collection

- Compiler support
  - Compiler provide the garbage collector with "yield point" at regular intervals in the code. At these points a thread can guarantee a consistent heap state so that control can be yielded to the garbage collector
  - Called GC-point in Sun's CDC VM