High-Performance Object-Oriented Virtual Machines

Lars Bak

CEO

OOVM A/S

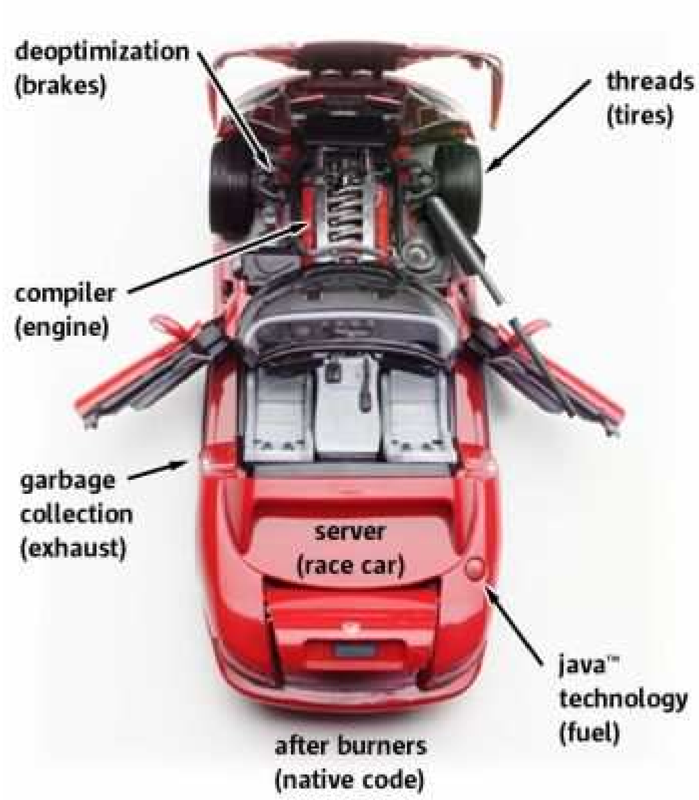
www.oovm.com

About Lars Bak

- ***Education***: Ms. degree in computer science at Aarhus University 1988
- ***Competence***: Project manager, virtual machine engineer
- ***Track record:***
  - 2002: CEO and founder of OOVM A/S
  - 2000: Designer of the CLDC HotSpot at Sun Microsystems Inc. CLDC Hotspot is a high performance Java virtual machine for mobile devices
  - 1997: Manager and technical lead for HotSpot at Sun Microsystems Inc. HotSpot is a high performance Java virtual machine for desktops and servers, USA
  - 1994: Technical lead on HotSpot at startup company Animorphic Systems, USA
  - 1990: Researcher in the Self group at Sun Microsystems Labs., USA
  - 1988: Founded Mjølner Informatics with others from the Mjølner research project
  - 1986: Joined the Nordic research project Mjølner

What Is A High-performance Virtual Machine?

It is a virtual machine that applies an array of optimizations while preserving the illusion that it executes bytecodes



What Is High-performance?

- ✓ Fast bytecode execution
- ✓ Fast startup time
- ✓ Fast warm-up time
- ✓ Small pauses
- ✓ Scalable
  - Heap size
  - CPUs

Early JVM Performance

- ✓ Poor performance results in bad programming style
  - Use of final methods
  - OO abstractions eliminated
  - Use of free lists

Then Let's Add A JIT

- ✓ Compile a method the first time it is called
- ✓ Compilation hurts:
  - Slow start-up time
  - Memory bloat
- ✓ Good code vs. fast compiler

Give Me A Benchmark And I'll Make It Fast

- Tuning for a small number of Benchmarks is easy
- Tuning so most programs run fast is very hard
- Most virtual machines balance
  - Fast bytecode performance
  - Startup performance
  - Interactive performance
  - Memory footprint
  - Scalability

Profiling A System

Where does time go?

- ✓ Byte code execution
- ✓ Runtime system
  - Garbage collection
  - Thread management
- ✓ Native libraries
- ✓ Operating system

Obstacles To Performance

- ✓ Platform independent byte codes
- ✓ Object-oriented
  - Virtual calls are the default
  - High call frequency
  - High allocation rate
  - Garbage collection
- ✓ Synchronization in language
- ✓ Dynamic class loading

The Agenda

- ✓ Fast virtual dispatch
- ✓ Runtime compilation
- ✓ Adaptive optimization
  - Aggressive inlining
  - Deoptimization

Please ask questions during the presentation

Dynamic Dispatch

- Core part of execution in most OO systems
- Target method at a call site dependens on receiver type
- Runtime lookup is needed to find the method

Lookup Function

- f(receiver type, name) -> method
- Implementation techniques for making the lookup function efficient
  - Virtual dispatch tables
  - Inline caches
  - Hash table

Virtual Dispatch Tables

- ✓ Indirect method table
- ✓ Often used in statically typed languages like Java and C++
- ✓ Technique
  - Object points to dispatch table
  - Method gets index in table of defining class
  - Call

```
vtbl   = obj->vtable()
target = vtbl[index]
call target
```

Problems With Vtable

- ✓ Indirect calls are very expensive on modern CPUs
- ✓ Can only handle single inheritance
- ✓ Does not work with interface calls
- ✓ Makes incremental execution very hard

Inline Caching

- Save the result from last lookup
- Deusch-Schiffman Smalltalk, 1984
- >85% of all calls are monomorphic in most object oriented systems
- Use self modifying code for implementation
- Where should the cache result reside?

Inline Caching

- ✓ Call site has several modes
  - Empty (no targets)
  - Monomorphic (one target)
  - Megamorphic
- ✓ Inline caching used in monomorphic case

Code For Inline Cache

- Caller

```
// receiver is in ecx
move eax, <address of receiver type>
call target
```

- Callee

```
cmp eax, ecx[class offset]
bne _inline_cache_miss
… code for callee method …
```

Inline Caching

- ✓ Fast on modern CPUs
- ✓ Used in most OO systems where self modifying code is permissible
- ✓ Side benefit: type information is collected

Hash Tables

- Often needed with dynamic typing
- Used in Smalltalk and Self
- Cache for f(receiver type, name) -> method
- Used as secondary lookup for inline cache

Hash Table Problems

- ✓ Many collisions result in slow down
- ✓ But, what about a 2-way associative cache
- ✓ Hard to update in multi threaded execution
- ✓ Use thread local cache

Combined Solutions

- Inline caching + hash table
  - Self + Smalltalk
- Inline caching + virtual table
  - Java

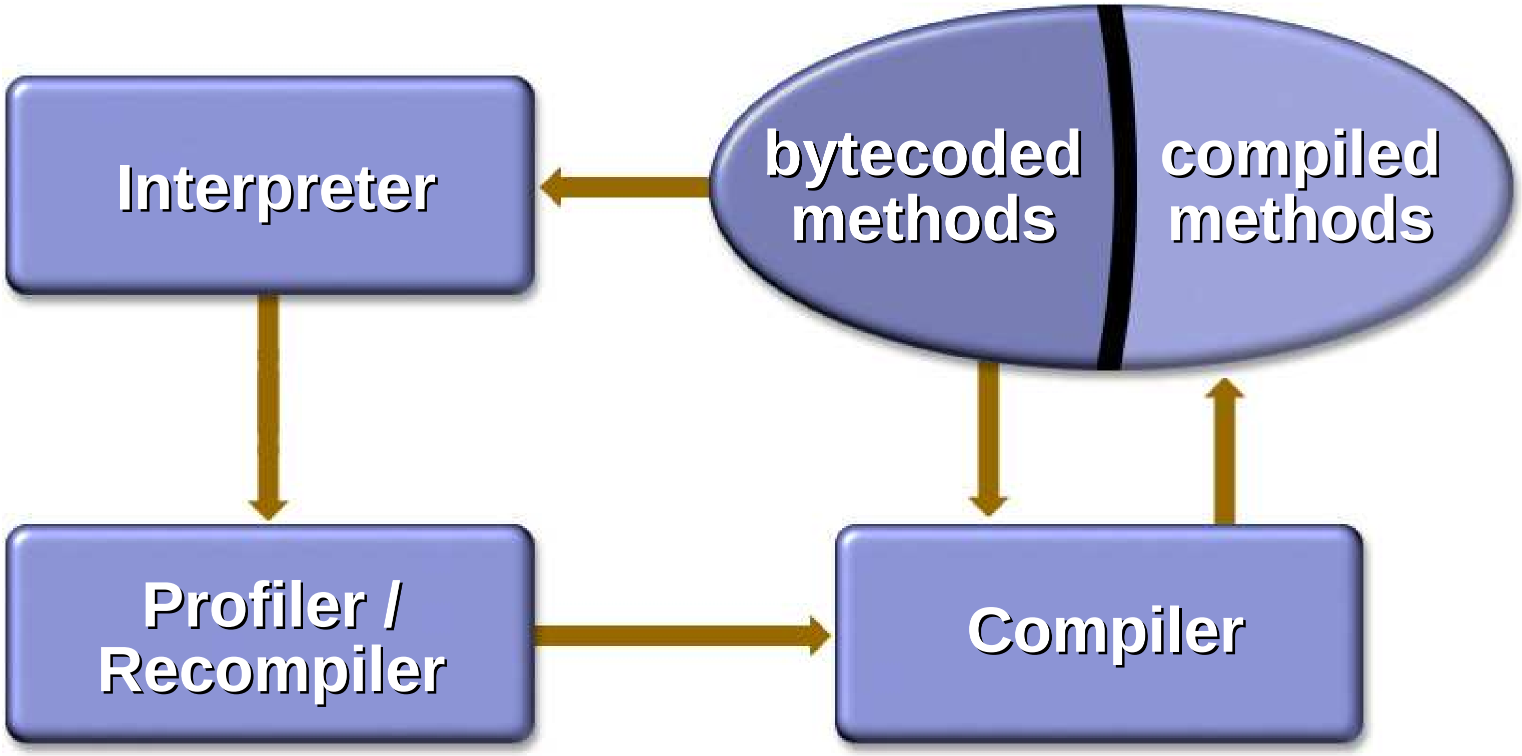
Dynamic Compilation

- The process of compiling at runtime
- Goals
  - Fast startup
  - Great performance
  - Good interactive performance
  - Minimize memory footprint

Execution Model

- ✓ Interpreter and compiled code combination
- ✓ Allows the systems to decide:
  - When to compile?
  - What to compile?

Adaptive Compilation



Should We Compile That Method?

- ✓ Compile the method if the total execution time is reduced
- ✓ How do we predict if it is beneficial?
- ✓ Use the past to predict the future

Learning From The Past

- ✓ Maintain a sliding window of the past to answer:
  - What to compile?
  - When to compile?
- ✓ Two approaches:
  - Invocation counters
  - Sample based profiling

Invocation Counters

- ✓ All methods have invocation counters
- ✓ Incremented on entry and backward branch
- ✓ Invocation overflow invokes compiler
- ✓ Counter decay to prevent compilation of infrequently executed methods

Invocation Counters

- Pros:
  - Simple solution
- Cons:
  - Hard to predict behavior
  - Does not control how much time is spent in compiler
  - Decay does not work! Why?

Sample Based Profiling

- ✓ Measure time spent:
  - In interpreter
  - In compiler
  - In compiled code
- ✓ Collect HotSpot list

Abstract Data Types and Performance

- ✓ Avoid premature optimizations
- ✓ Keep abstract data types clean
  - Use access methods
  - Use small readable methods
- ✓ Makes the program easier to understand and maintain

...and Abstract Data Types

- ✓ Leaves optimization options to the virtual machine
  - Inlining decisions based on behavior
  - Inlining decisions based on memory
  - Inlining decisions based on platform

Recap: Why We Need Inlining?

- ✓ Too many dynamic dispatch
- ✓ Inlining will:
  - Create bigger basic blocks
  - Give optimizer more to work on
  - Might avoid allocation

Code For Point

```
class Point {
  virtual float x() = 0;
  virtual float y() = 0;
  float distance(Pointer other) {
   float dx = other.x() – x();
   float dy = other.y() – y();
   return sqrt(dx*dx + dy*dy);
  }
 }
```

Code For Cartesianpoint

```
class CartesianPoint {
   float _x, _y;
   virtual float x() { return _x; }
   virtual float y() { return _y; }
}
```

Code For Polarpoint

```
class PolarPoint {
   float _rho, _theta;
   virtual float x() {
     return _rho * cos(_theta);
   }
   virtual float y() {
     return _rho * sin(_theta);
   }
}
```

Customization Of Code

- ✓ Code for distance in hard to optimize
- ✓ Too many dynamic dispatch
- ✓ Customize the code for dispatch for
  - CartesianPoint
  - PolarPoint

Customization

- Pros
  - Dispatch to self/this is now bound
  - Opens up for better optimizations
- Cons
  - Code is replicated taking up more space

Self show why customization should be used with caution!

Over-customization In Self

- ✓ All compiled methods are customized
- ✓ Morph system had 40 different types of UI components
- ✓ Many methods only existed in top trait object
- ✓ Resulted in 40 copies of SAME compiled method

Cheap Inlining

- In languages with static typing not all virtual methods require dynamic dispatch
- Java
  - Virtual methods in final classes
    - String
  - Virtual methods in sealed packages

Cheap Inlining

- Pros
  - Simple performance gain
- Cons
  - Stack traversal is hard
  - Debugging is still complicated

Could this be done at the byte code level?

Type Feedback

- ✓ Profile unoptimized to collect receiver types
  - Inline caches
  - poly-morphic inline cached
- ✓ Feed the compiler with the profile data

Code With Type Feedback

```
Source:
  x = p->x();

Generated:
  if (p->class == CartesianPoint){
    x = p->_x;
  } else {
    x = p->x();
  }
```

Aggressive Inlining

- ✓ Use class hierarchy analysis if language has static types
- ✓ Inline based on the current state of the class hierarch
- ✓ Back out if class loading violates assumptions
  - Used deoptimization as described later

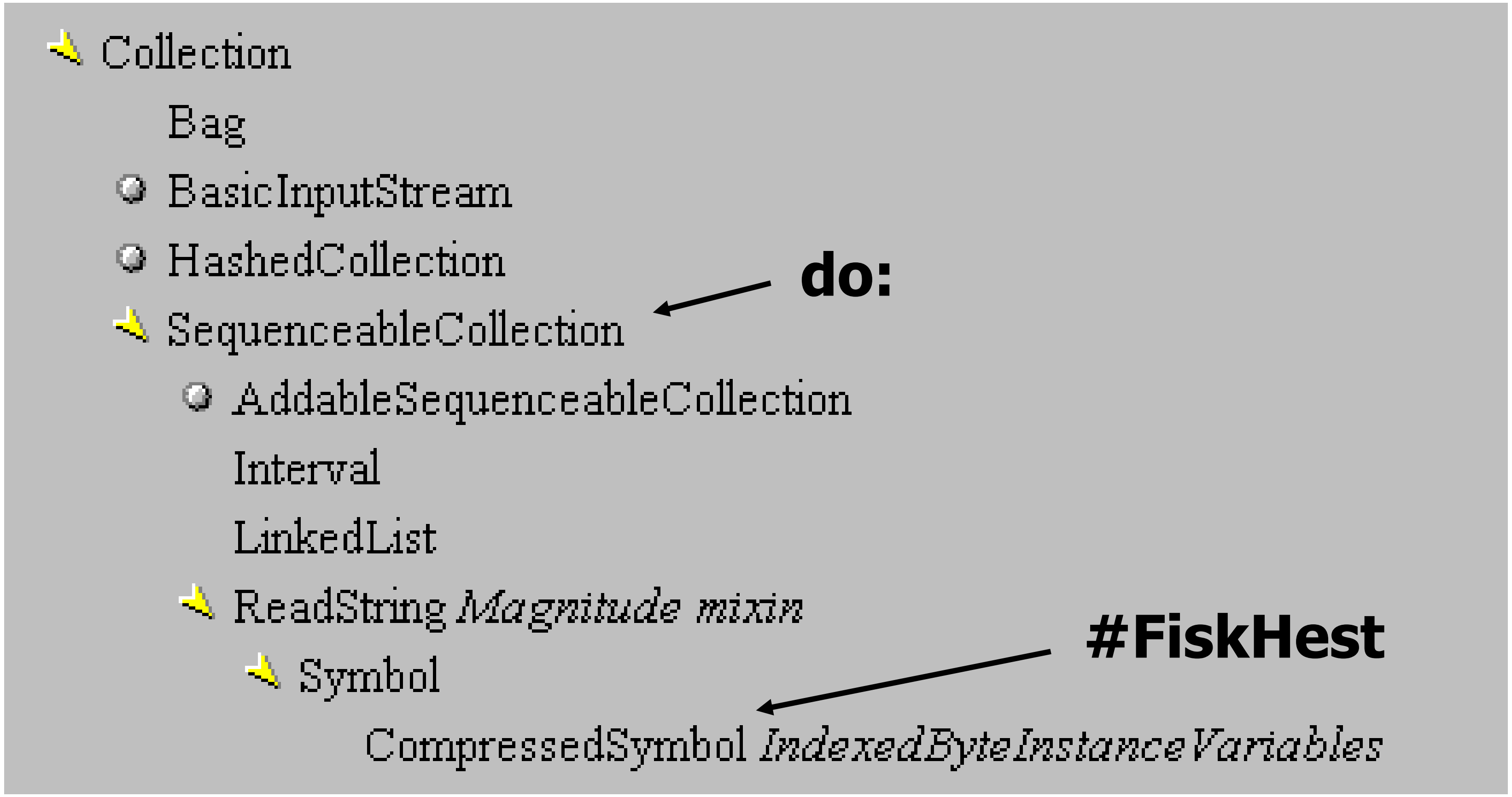
Context Elimination

✓ What happens when you execute in Strongtalk?

```
#FiskHest do: [:char | … ]
```

- Block context is allocated
- do: method is invoked for instance of CompressedSymbol

Collection Hierarchy In Strongtalk™



SequenceableCollection

```
do: f
  1 to: self size do:
     [ :i | f value: (self at: i)]
```

Context Elimination

- ✓ Customization is needed to bind self
- ✓ Inline enough to avoid allocation of contexts on common execution paths
- ✓ Use uncommon traps where contexts can escape

Uncommon Traps

- ✓ How to avoid compiling for uncommon situations
  - If code has never been executed
  - If types are unlikely to appear
- ✓ If an uncommon trap is executed the compiled activation is converted to interpreter activations

Uncommon Traps Example

```
x = p->x();

if (p->class !=CartesionPoint) {
   uncommon_trap();
}
x = p->_x; // fast access
```

Debugging Of Optimized Code

- Deoptimization support for debugging of optimized code with dynamic deoptimization

Problem With The Old Way

- ✓ Special compile flag to generate debug executable
  - Very slow execution
  - Program behaves differently
  - Not usable for production systems
  - Only used for inspection/single stepping during development

What We Really Want

- ✓ Preserve the illusion of bytecode interpretation
- ✓ Important for platform independent debugging
- ✓ Provides serviceability for systems in production systems

Deoptimization

- ✓ The act of converting a compiled activation into a set of interpreter activations

What Can Deoptimization Be Used For?

- Backing out of aggressive inlining
- Source level inspection
- Support for uncommon traps
- Support for source code level debugging
- Support for incremental execution

Displaying The Stack

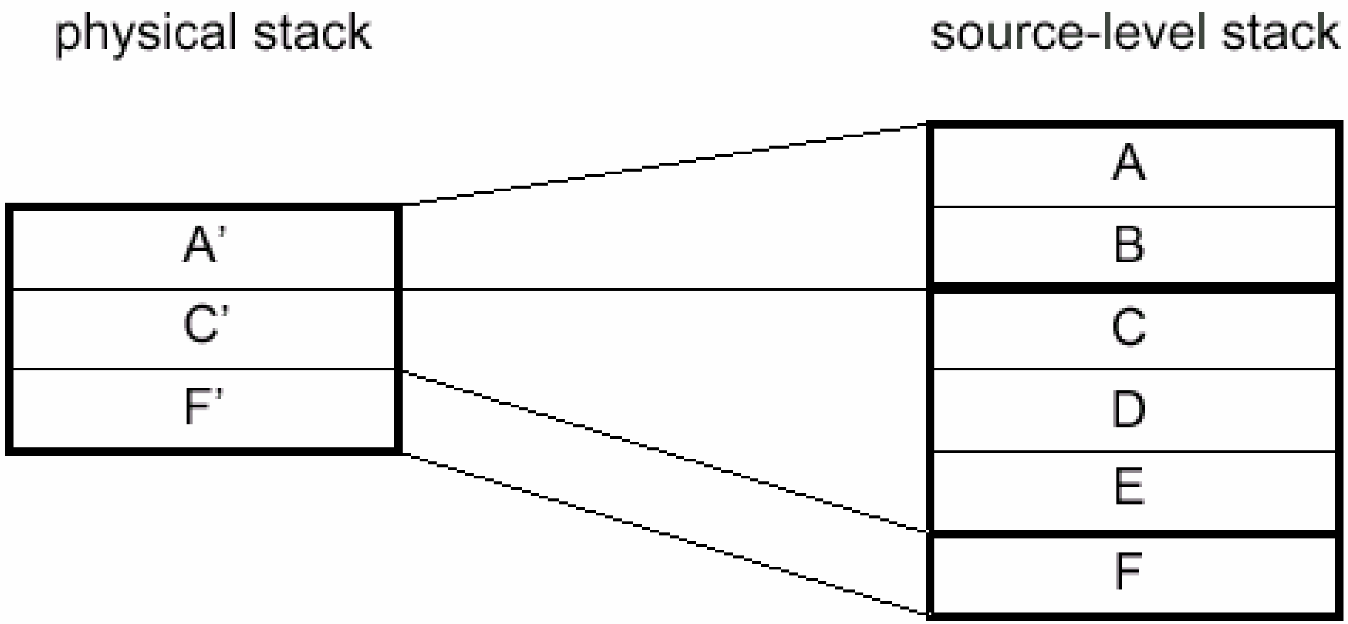


Figure 1. Displaying the stack

Recovering Interpreter State

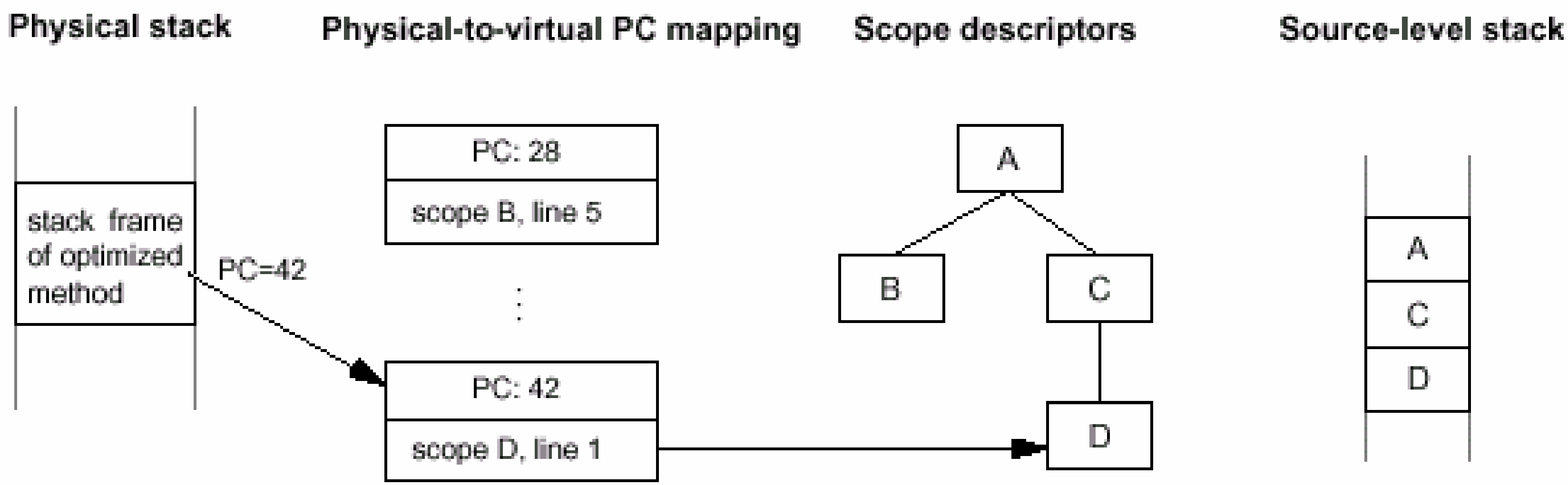


Figure 3. Recovering the source-level state

What Is Needed?

- Debugging formation stored with compiled code
  - Enough information to reconstruct the interpreter state at each interrupt points.
- Access to compiled frame

Abstraction For Debugging Information

```
class Activation {
   Method get_method();
   int get_bytecode_index();
   Value* get_locals();
   Value* get_expression_stack();
}
```

Value Information

```
class Value;
  class ConstantValue;
  class OnStackValue;
  class InRegisterValue;
  class EscapedObject;
  class …;
```

The Process Of Deoptimization

1) Identify activations to deoptimize
2) Transform the compiled activations into an off-stack interpreter based representation
3) Insert trap and remove compiled method
4) When returning to activation unpack the off-stack representation to the stack
5) Continue execution in interpreter

Eager/Lazy Deoptimization

- Lazy
  - Compiled code is kept around until all frames have returned
- Eager
  - Compiled frame are converted to off-stack representation eagerly

Deoptimization Conclusion

- Pros:
  - Makes aggressive inlining possible
  - Makes debugging possible in production mode
- Cons:
  - Inhibits some optimizations
    - Tail recursion elimination
    - Dead store elimination

High-performance Virtual Machine Conclusion

- ✓ Object oriented optimizations
  - Agressive inlining
  - Subtype check (instanceof & cast)
- ✓ Efficient memory management system
  - Fast allocation
  - Efficient garbage collection
- ✓ Ultra fast synchronization

... and a good compiler

Research Papers

- ✓ **Adaptive optimization for Self: Reconciling High Performance with Exploratory Programming**, *by Urs Hölzle*

  Ph.D. thesis, Computer Science Department, Stanford University
- ✓ **Debugging Optimized Code with Dynamic Deoptimization**, *by Urs Hölzle, Craig Chambers, and David Ungar*

  Proceedings of the ACM SIGPLAN `92 Conference on Programming Language Design and Implementation, 32-43, San Francisco, June, 1992
- ✓ **Tenuring Policies for Generation-Based Storage Reclamation**, *by David Ungar and Frank Jackson*

  Proceedings of OOPSLA 1988: San Diego, California, 1-17
- ✓ **Incremental Collection of Mature Objects**, *by Richard L. Hudson and J. Eliot B. Moss*

  IWMM 1992: 388-403
- ✓ **A Fast Write Barrier for Generational Garbage Collectors**, *by Urs Hölzle*

  OOPSLA 1993 Workshop on Garbage Collection, Washington, D.C., October 1993
- ✓ **A Simple Graph-Based Intermediate Representation**, *by Cliff Click*.

  Proceedings of the Intermediate Representations' 95 Workshop, pages 35-49.
- ✓ **Global Code Motion, Global Value Numbering**, *by Cliff Click*.

  Proceedings of the ACM SIGPLAN `92 Conference on Programming Language Design and Implementation '95.