# The Maxine Virtual Machine

Sun Microsystems Laboratories
Ben L. Titzer
ben.titzer@sun.com

April 30, 2009

# Outline

- Background
- Design Philosophy
- Runtime Overview
- Compilation Overview
- VM Status
- TM Potential

# Background

- Research VM written in Java™
- Started in 2005 by Bernd Mathiske
  - Original goal: malleable JVM to explore hardware support for objects, particularly GC
  - Evolved into a general purpose JVM effort
- Currently:
  - Doug Simon, PI (2006)
  - Laurent Daynes (2006)
  - Michael Van De Vanter (2007)
  - Ben L. Titzer (2007)

# Design Philosophy

- “Meta-circular”
  - No VM / application code distinction
  - Write as much as possible in Java
    - No special GC handles in source
  - Use the host VM's implementation
    - Reflective invocation (bootstrapping)
    - Enumeration of fields, methods (bootstrapping)
    - Processing method annotations (bootstrapping)
    - Reflection during Inspecting
  - Bootstrap
    - Custom classfile parser, bytecode verifier, compiler
    - Compiler compiles itself
    - Generates binary image with code + data

# Runtime Overview

- Everything is a Java object
- Internal representation of programs: Actors
  - ClassActor, FieldActor, MethodActor...
- Schemes encapsulate large modules, e.g. GC
- Currently compile-only execution approach
  - Fast JIT + optimizing compiler
- Simple semi-space GC
  - Beltway GC framework implemented, not integrated
- Use the standard JDK 1.6 runtime JAR
  - Substitution mechanism allows Maxine to implement many JDK native methods in Java
- Small C layer attaches to OS services, loads image

# Configurable “Schemes”

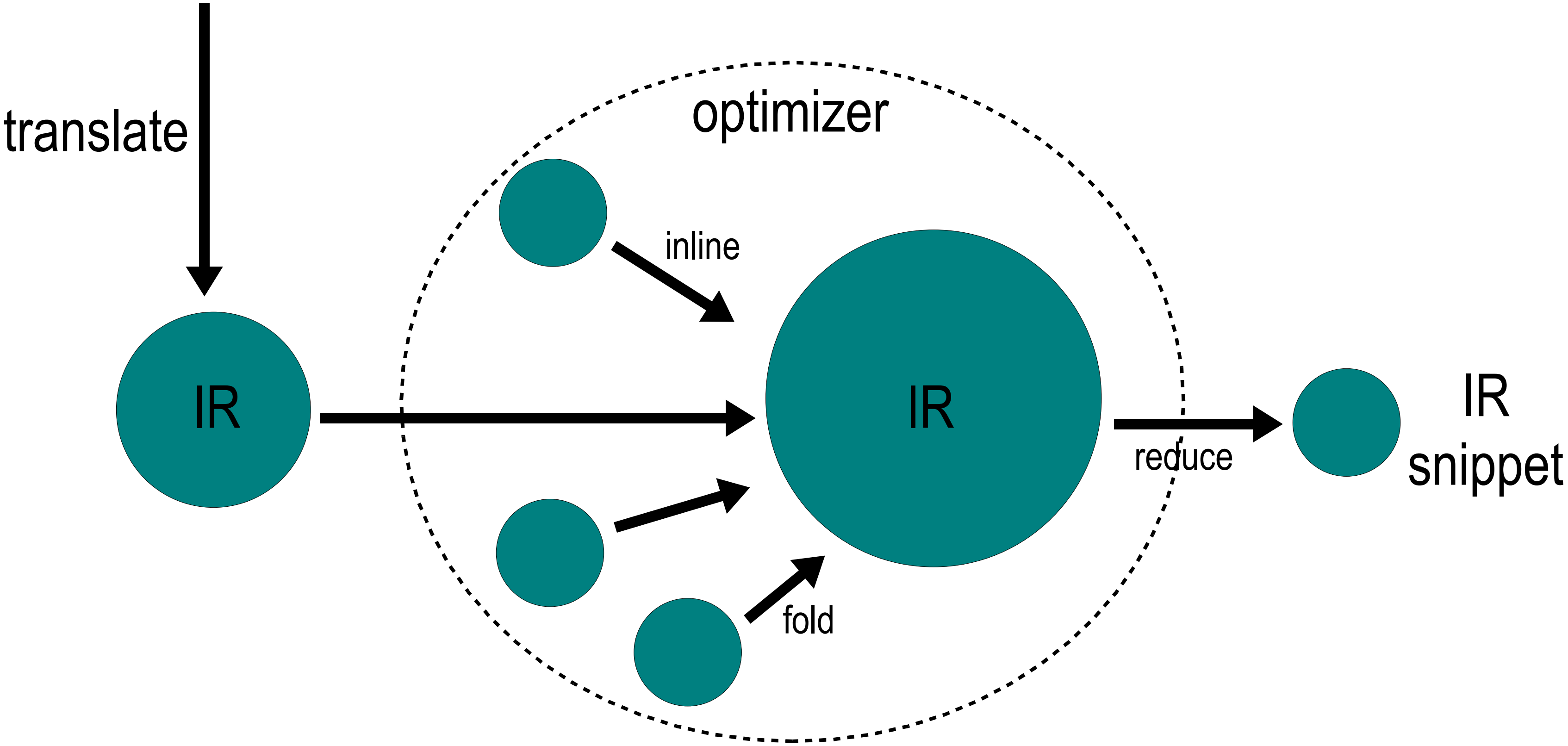
- `HeapScheme` - allocator, garbage collector, read and write barriers
- `LayoutScheme` - object layout
- `MonitorScheme` - synchronization operations
- `ReferenceScheme` - reference operations
- `DynamicCompilerScheme` - JIT compiler
- `CompilerScheme` - bootstrap compiler
- `TargetABIScheme` - ABI for compiled code
- `CompilationScheme` - (re)compilation policy
- `RunScheme` - VM startup

# Bootstrap/Optimizing Compiler

- Uses a variant of CPS as its main IR for optimization
- Layered compiler with several IRs
  - BIR → CIR → DIR → EIR → Target
  - Porting requires only new EIR and Target translations
- Meta-circular snippets:
  - Instead of writing the IR to implement a particular runtime feature, one writes Java “snippets”
  - Compiler bootstrap phase reduces snippets to pieces of IR
  - Translation of bytecodes weaves IR
    - But runtime implementors typically only write Java code, e.g. write barrier

# Snippet Example

```
@INLINE
public static Word selectVirtualMethod(Object obj, VirtualMethodActor declaredMethod) {
    final Hub hub = ObjectAccess.readHub(obj);
    return hub.getWord(declaredMethod.vtableIndex());
}
```

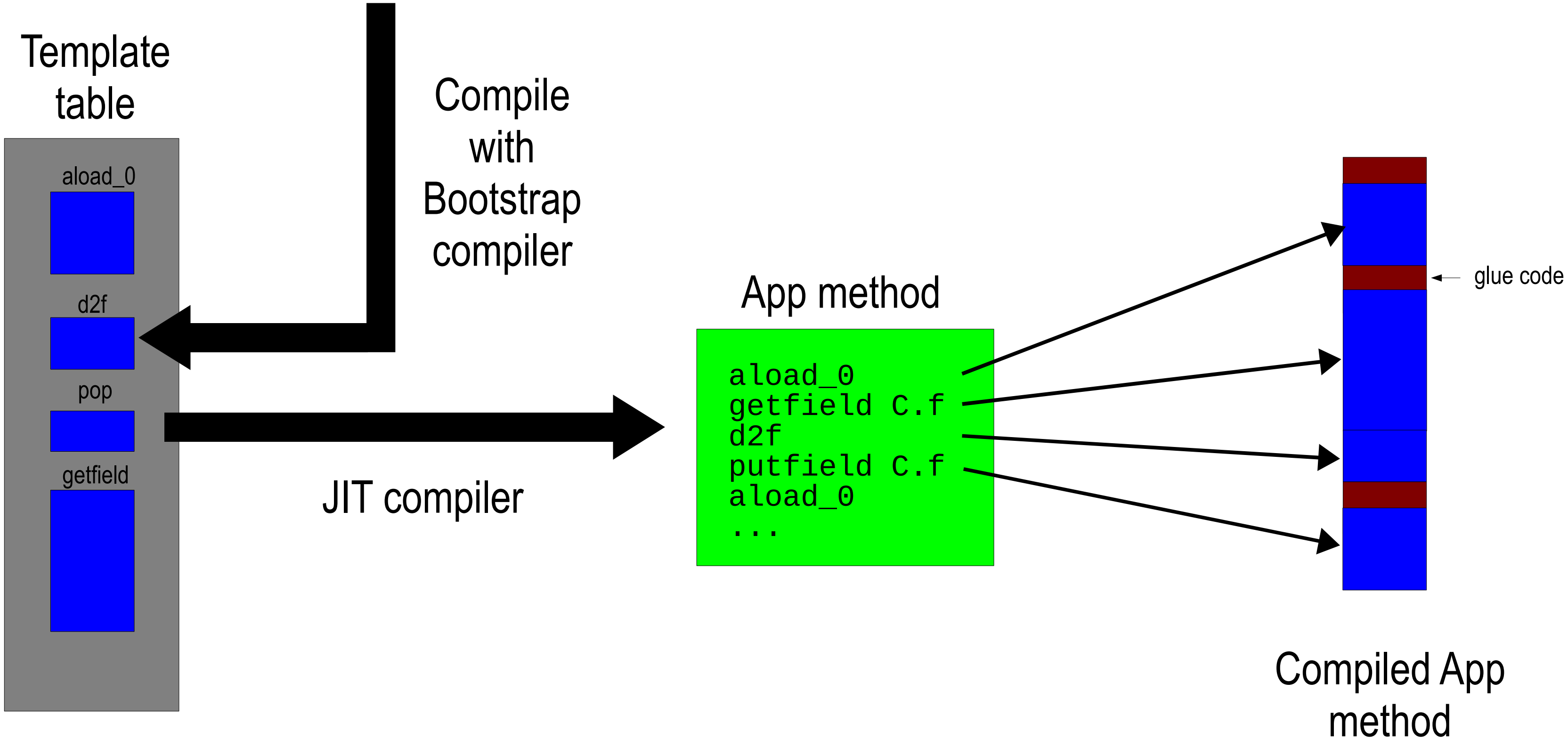


# Template-based JIT

- Compile-only strategy requires a fast first compiler
- Template-based JIT has a table of machine code sequences for each bytecode
  - > Resolved, unresolved, instrumented versions
- Single-pass over the bytecode to emit machine code
  - > Plus a pass over forward branch sites to patch
  - > Complex methods may require GC map computation
- Bootstrap compiler generates machine code for templates
  - > Requires glue code between templates
  - > Restrictions on templates
  - > Some tradeoff on code quality

# JIT Template Example

```
public static void d2f() {
     final double value = JitStackFrameOperation.peekDouble();
     JitStackFrameOperation.removeSlots(1);
     JitStackFrameOperation.pokeFloat(0, (float) value);
}
```



```
aload_0
getfield C.f
d2f
putfield C.f
aload_0
...
```

# JDK Substitution Example

```
@SUBSTITUTE(java.lang.Object.class)
public class JDK_java_lang_Object {

    @SUBSTITUTE
    public int hashCode() {
         return ObjectAccess.makeHashCode(this);
    }

    @SUBSTITUTE
    public Object clone() throws CloneNotSupportedException {
        if (Cloneable.class.isInstance(this)) {
            return Heap.clone(this);
        }
        throw new CloneNotSupportedException();
    }

    . . .
}
```

# VM Status

- JCK tests = 88% pass
  - Corner cases, malformed input, JDK issues
- SPECjvm98 = all pass
  - 1.6-5x slower than production HotSpot
- DaCapo = 5 of 12 pass
  - 5-10x slower than HotSpot
- Issues:
  - Stale state from JDK bootstrapping
  - GC bugs, race conditions
  - Many unimplemented JVM_* C functions

# Maxine TM Potential

- Transactional Lock Elision
  - Replace `synchronized() { … }` with HTM on Rock
  - More work needed on Maxine SPARC port
  - Visibility and compilation policy
- STM
  - Virendra Marathe is building an STM prototype
  - Explored STM instrumentation in Maxine
- New Compiler
  - Eye toward STM instrumentation and optimization

# Maxine STM Ideas

- Approaches to denoting transactions
  - Special method annotation `@TRANSACTION`
  - `try {...} catch (TransactionExit e) {}`
  - Block statement `atomic { }`
  - None implemented - Tell us what you want
- Instrumenting called methods
  - On-demand (re)compilation of methods
  - VM must manage multiple method versions
- Transactional word in object header
  - Easily done in Maxine with `LayoutScheme`

# Maxine STM Prototype

- API between compiler/VM and STM
  - `beginTransaction()`
  - `commitTransaction()`
  - `readFieldX(Object, Field)`
  - `writeFieldX(Object, Field)`
  - `openForRead(Object)`
  - `openForWrite(Object)`
  - ...
  - How are TM logs exposed to GC?
- Can test in isolation with hand-written transactions
  - Clunky interface
  - Maybe bytecode rewriting?

# Open Source

- GPL Version 2
  - License-compatible with OpenJDK
- Builds with JDK 1.6 and OpenJDK 1.6
  - No binary changes required (up to 6u12)
  - But new updates often break Maxine
- Website
  - http://research.sun.com/projects/maxine
- Mercurial repository
  - https://kenai.com/hg/maxine~maxine

# Platforms

- Primary:
  - Solaris™ x86-64
  - Xen x86-64
    - See Sun Labs Guest VM Project
    - http://research.sun.com/projects/guestvm
- Secondary:
  - Mac OS X x86-64
  - Linux x86-64
  - Solaris SPARCv9
- No other ports currently
  - It hurts just thinking about it

# Upcoming Major Items

- HotSpot Client compiler port (Oct)
  - Performance, architecture validation
  - A chance to explore STM / Compiler interface
- Generational GCs
  - Performance
  - Beltway framework waiting to be integrated
- Potentially integrate Virendra's STM
  - If we get there (and he gets there)
- Revisit JIT/interpreter question (?)

Sun microsystems

# Thank you!