The Maxine VM

Bernd Mathiske

Sun Microsystems Laboratories

Maxine Open Source Research VM

- GNU General Public License version 2
  - License-compatible with OpenJDK™

- Currently builds on JDK 1.6 release
- Not a fully compliant Java™ Virtual Machine (yet)

- Alpha release source code:

  `https://maxine.dev.java.net/`

Maxine Platforms

- *Supported:*
  - Solaris™ Operating System / SPARC® Technology
  - Solaris / x64
  - Mac OS X / x64
  - Xen / x64
- *Under development:*
  - Solaris OS / x32
  - Linux / x64
  - Linux / x32
  - Mac OS X / x32
- *Contemplated:*
  - Xen / x32
  - ARM
  - PowerPC
  - Windows

Pre-Alpha Release: Proof-Of-Concept

- Miscellaneous Meta-Circular Design Aspects
- Bootstrap
- Configurability
- Low-Level object interfaces and layering
- Unsafe features
- Portable JIT
- Safepoint mechanism
- ...

Sun microsystems

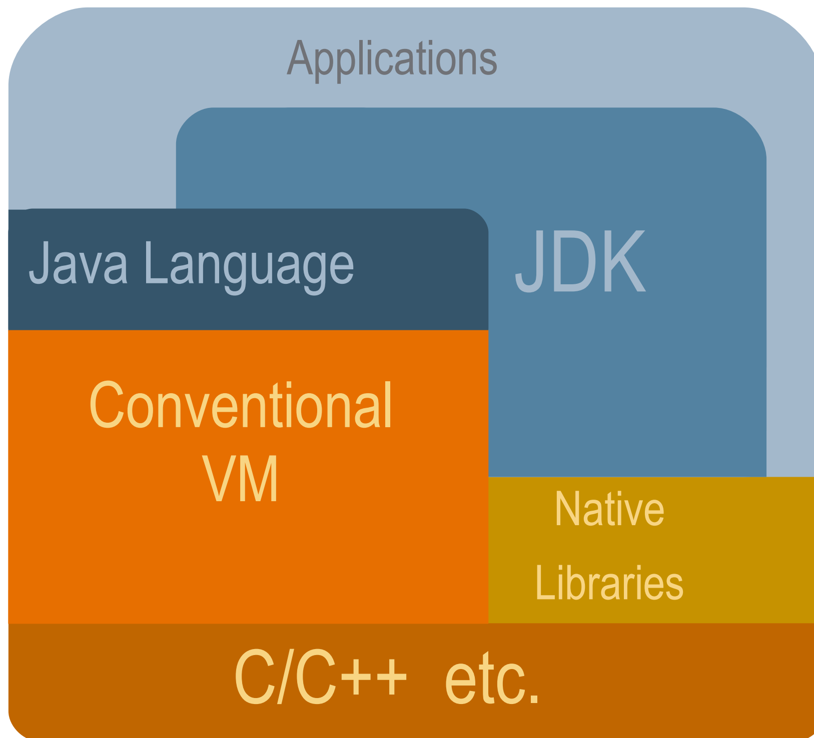
Configurable Components ("Schemes")

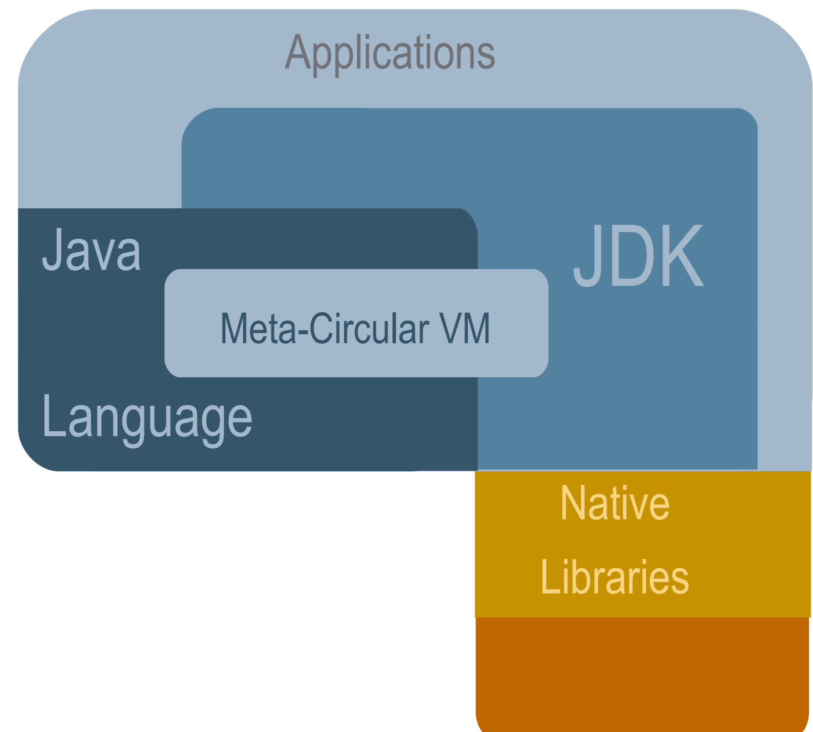
- Garbage collector
- Object layout
- Object reference representation
- Read and write barriers
- Fast JIT
- Optimizing compiler
- Optimizing compiler's ABI
- Dynamic re-compilation policy
- Method call trampolines (for dynamic compilation)
- Thread synchronization (object monitors)
- Startup sequence

Maxine VM Research Areas

- Addressed today:
  - Modularity
  - Tool support
  - Compilation
  - Garbage collection
  - Concurrency
  - Control flow
  - Data structures
  - Hypervisor
  - ...
- Anticipated:
  - Task Isolation
  - Resource control
  - Adaptive runtime behavior
  - Reliability
  - Real-time
  - Multi-language
  - Emulation
  - ...

Conventional vs. Meta-Circular





Meta-Circular VM Design

- The VM is written in the same language it executes
- The VM uses JDK packages
  and implements their downcalls into the VM
- Built around an optimizing compiler (not an interpreter)
- The optimizing compiler in the VM translates itself

Low Level Programming Support

- Direct calls to C without JNI
- Direct callbacks from C without JNI
- Assemblers, Disassemblers
- Unsafe type loophole
- Unboxed word types
  - 32 or 64 bits, depending on VM configuration
  - Unboxed representation like primitive types
  - Method call syntax like boxed object types

Assembler Use **outside** Compilers

- Safepoint stub
- Safepoint instruction
- Deoptimization trap instruction
- Breakpoint instruction

About a dozen assembler instructions total.

Assembler NOT used for:

- Most JIT bytecode implementations
- JNI stubs
- Native call stubs
- Write barriers
- OSR (TBD)
- Deoptimization
- Method dispatch
- ...

Fineprint: To be fair, some of this only holds because there are a few extra builtins implemented by the compiler

Layered Views on Objects

java.lang.Object — what the Java application program(mer) sees

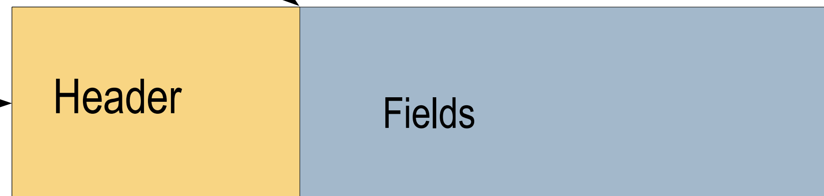
Reference — what the mutator primarily uses:
low-level operations with read/write barriers

Grip — what the garbage collector primarily uses:
low-level operations without read/write barriers

Pointer

origin

cell



just an example layout

Common Pointer/Object Interface

```
public interface Accessor { ...

    boolean isZero();

    byte readByte(Offset offset);

    int readInt(int offset);

    Word readWord(Offset offset);

    Reference readReference(int offset);

    void write Boolean(Offset offset, boolean value);

    void writeDouble(int offset, double value);

    void writeReference(int offset, Reference value);

    ...

}
```

implemented by these classes (and all their subclasses):

Reference **Grip** **Pointer**

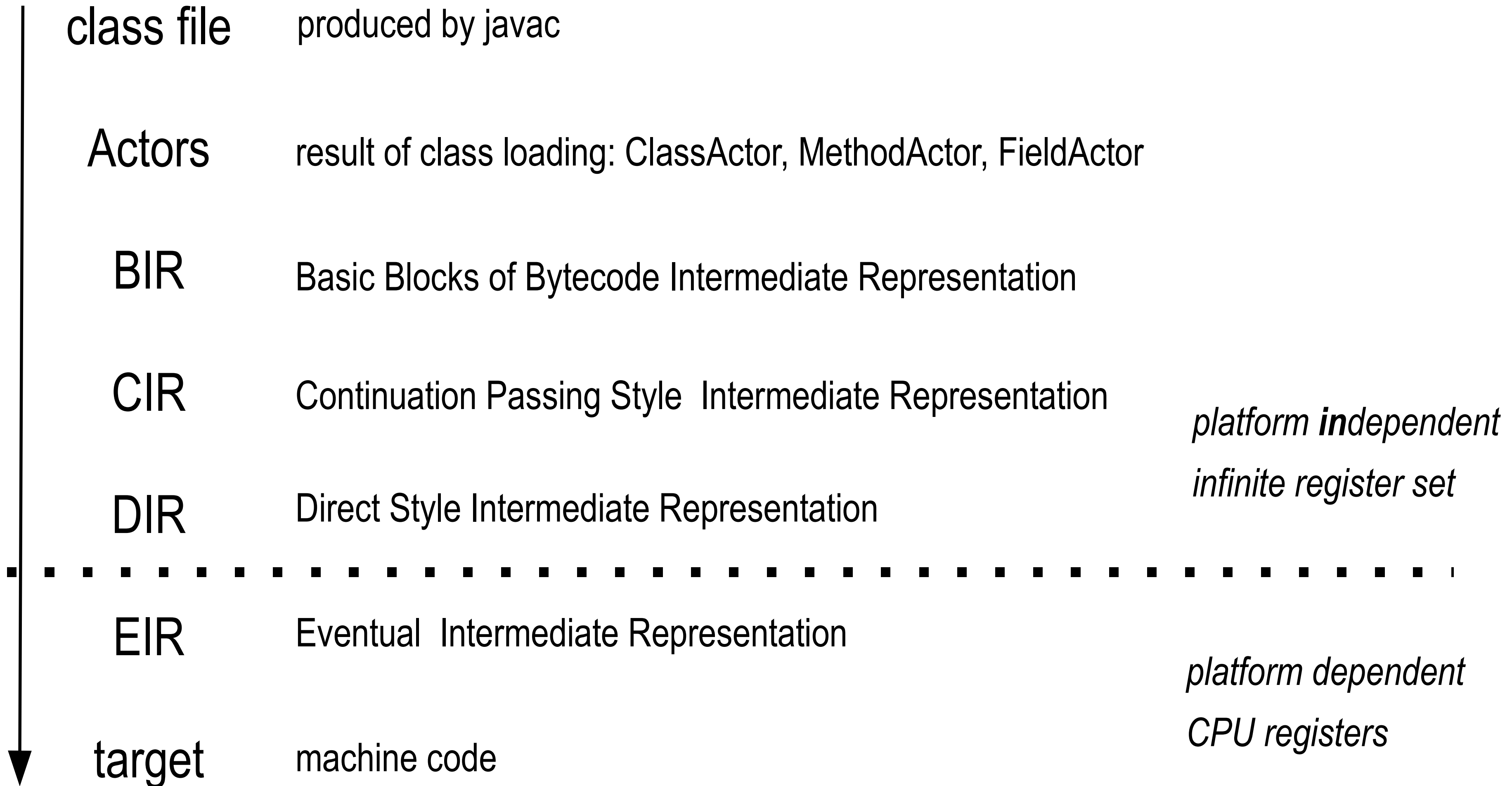
No Compiler Work: Write Barrier

```
public class CardReferenceScheme { ...
    @INLINE public void performWriteBarrier(Reference r) {
        cardTableBase().writeByte(
             r.toOrigin().unsignedShiftedRight(CARD_SHIFT).asOffset(), ZERO);
    }

    @INLINE public void writeReference(Reference R, Offset offset, Reference v) {
        performWriteBarrier(r);
        gripScheme().fromReference(r).writeGrip(offset, v.toGrip());
    }

public class DirectGripScheme { ...
    @INLINE public void writeGrip(Grip grip, Offset offset, Grip value) {
        toOrigin(grip).writeGrip(offset, value);
    }

public class Pointer { ...
    @INLINE public final void writeReference(Offset offset, Reference v) {
        if (Word.width() == WordWidth.BITS_64) {
            writeReferenceAtLongOffset(offset.toLong(), v);
        } else {
            writeReferenceAtIntOffset(offset.toInt(), v);
        }
    }

    @BUILTIN(PointerStoreBuiltin.WriteReferenceAtLongOffset.class)
    protected native void writeReferenceAtLongOffset(long offset, Reference v);

    @BUILTIN(PointerStoreBuiltin.WriteReferenceAtIntOffset.class)
    protected native void writeReferenceAtIntOffset(int offset, Reference v);
```

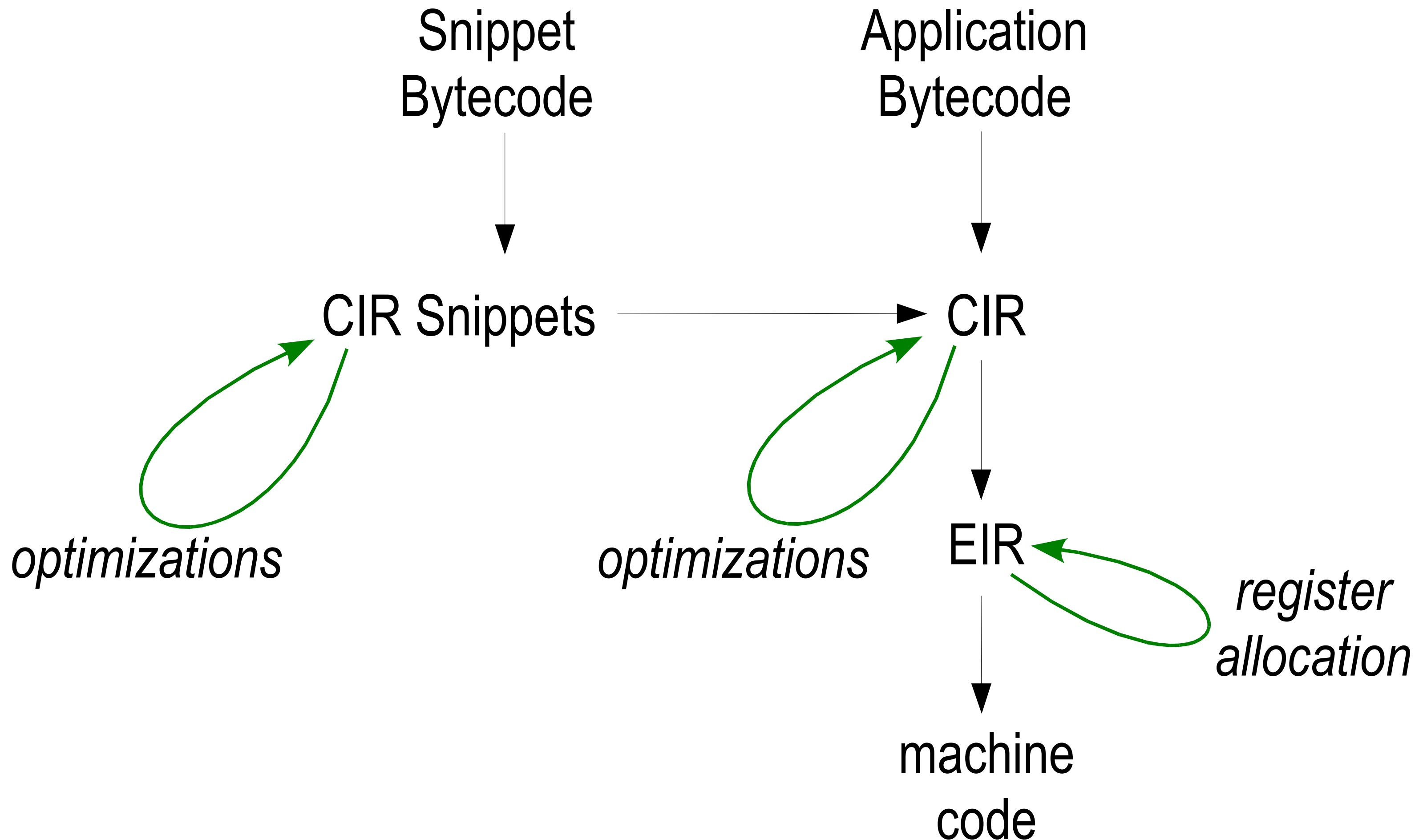
Optimizing Compiler Highlights

- Layered architecture
- Reduced compiler interface
- Meta-evaluation by reflective invocation
- Annotation-driven optimizations
- Interpreters for intermediate representations
- Continuation passing style
- No intermediate representation manipulation outside the translator and optimizer
- Almost no hand-written assembly code in the runtime
- Portable register allocator framework

Program Representations



Optimizing Compiler Overview



Snippets

- “IR macros” written in **source** code
- Compiled to IR and then stored
- Inlined into generated code
- Express anything larger than one machine instruction

- Decouple runtime features
  and compilers from each other

- => no IR construction other than byte code translation

Snippet Example: Invocation

```
@SNIPPET
@INLINE
public static Word selectVirtualMethod(Object receiver,
                              VirtualMethodActor declaredMethod) {
    if (declaredMethod.isPrivate()) {
        return makeCompiled(declaredMethod, ...);
    }
    final Hub hub = ObjectAccess.readHub(receiver);
    return hub.getWord(declaredMethod.vTableIndex());
}
```

Bytecode Generation

- Exception dispatch
- Synchronized methods
- Reflective invocation
- Native method invocation

=> streamlined compiler,
translating bytecodes without incongruous diversions

JNI Stubs

- Generated as bytecode
  - > Portable across compiler schemes
  - > JNI stub inlining is trivial
  - > Tested with all IR interpreters
- Reusing the bytecode assembler
  (also used for JNI function wrapping and exception dispatching)
- Special bytecode: `invokenative`
- Extra compiler builtin: `CreateStackHandle`
- Snippet: `ResolveNativeMethod`

JNI Stub Example

```
package com.sun;
public class Foo {
    static native void register(String name, int value);
}
```

save current thread	0: invokestatic JvmThread.current() 3: astore_2
save current JNI frame	4: aload_2 5: invokevirtual JvmThread.jniHandles() 8: astore_3 9: aload_3 10: invokevirtual JniHandles.top() 13: istore 4
push JNIEnv	15: invokestatic JvmThread.currentJniEnvironmentPointer()
push class reference handle	18: ldc com.sun.Foo 20: invokestatic JniHandles.createStackHandle(Object)
name == null?	23: aload_0 24: ifnull 34
false: push *name* handle	27: aload_0 28: invokestatic JniHandles.createStackHandle(Object) 31: goto 37
true: push null	34: invokestatic JniHandle.zero()
push *value*	37: iload_1
invoke native function	38: invokenative "Java_com_sun_Foo_register"
restore JNI frame	41: aload_3 42: iload 4 44: invokevirtual JniHandles.resetTop(int)
throw pending exception (if any)	47: aload_2 48: invokevirtual JvmThread.throwPendingException()
return	51: vreturn

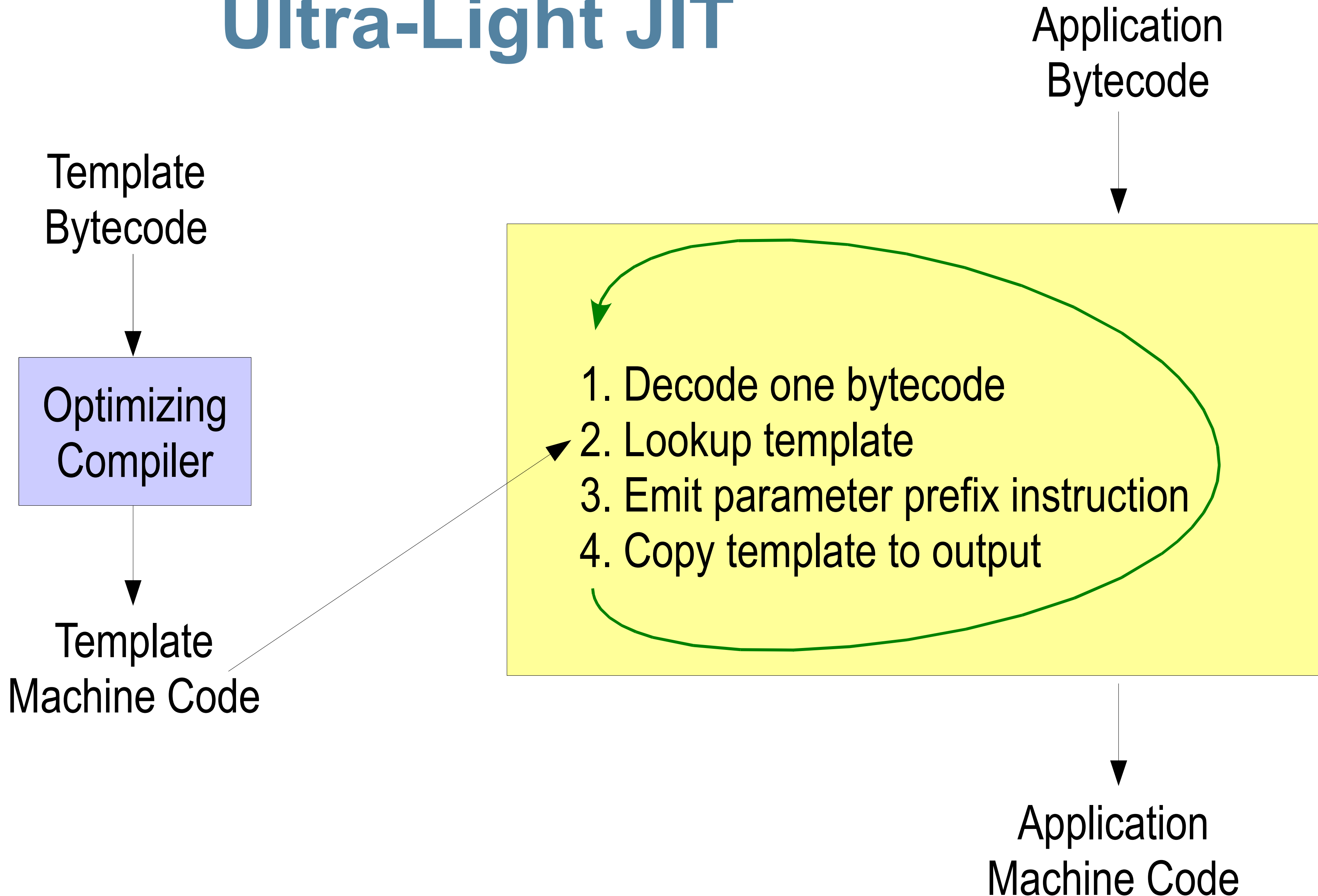
Ultra-Light JIT

- Goal: produce code as quickly as possible, hopefully preventing the need for an interpreter
- Single pass!
- Code quality is of secondary concern
- Closely matches the JVM spec's execution model
- No stack map production while generating code
  - > Stack maps can be filled in at GC safepoints as needed, by abstract interpretation without allocating

Portable JIT Implementation

- Copy machine code snippets from a template table
- Templates generated by the optimizing compiler at VM build time from Java source code
- One-instruction template prefix arguments specify local variable indices, constant pool indices
- Adapter frames mediate between stack-oriented JIT code and register-oriented optimized code and vice-versa

Ultra-Light JIT



Summary

- Compilers and other runtime features decoupled
- Assembler and IR wrangling avoidable
- When source code approach not possible, generate bytecode

Consequently...

- *Usually*, runtime feature implementation or modification does **NOT** involve (adding to or changing) any of the compilers.
- Experimental runtime system extensions are expedited by being shielded from compiler work.
- All the above should hold for alternative implementations of the compilers or layers thereof.
- The decoupling will also be maintained when adding a third kind of compiler to the game, a trace compiler, in collaboration with UCI.

Multi-Language Potential

- Fortress
- JRuby
- Jython
- JavaScript
- Java FX

- All of the above already have Java implementations
- Let's reuse scalable managed runtime investment
- No need for Java bytecode (or extensions thereof)

THANK YOU

research.sun.com/projects/maxine

maxine.dev.java.net