Splicing Modules with a Metacircular Saw:

“Snippets” in the Maxine VM

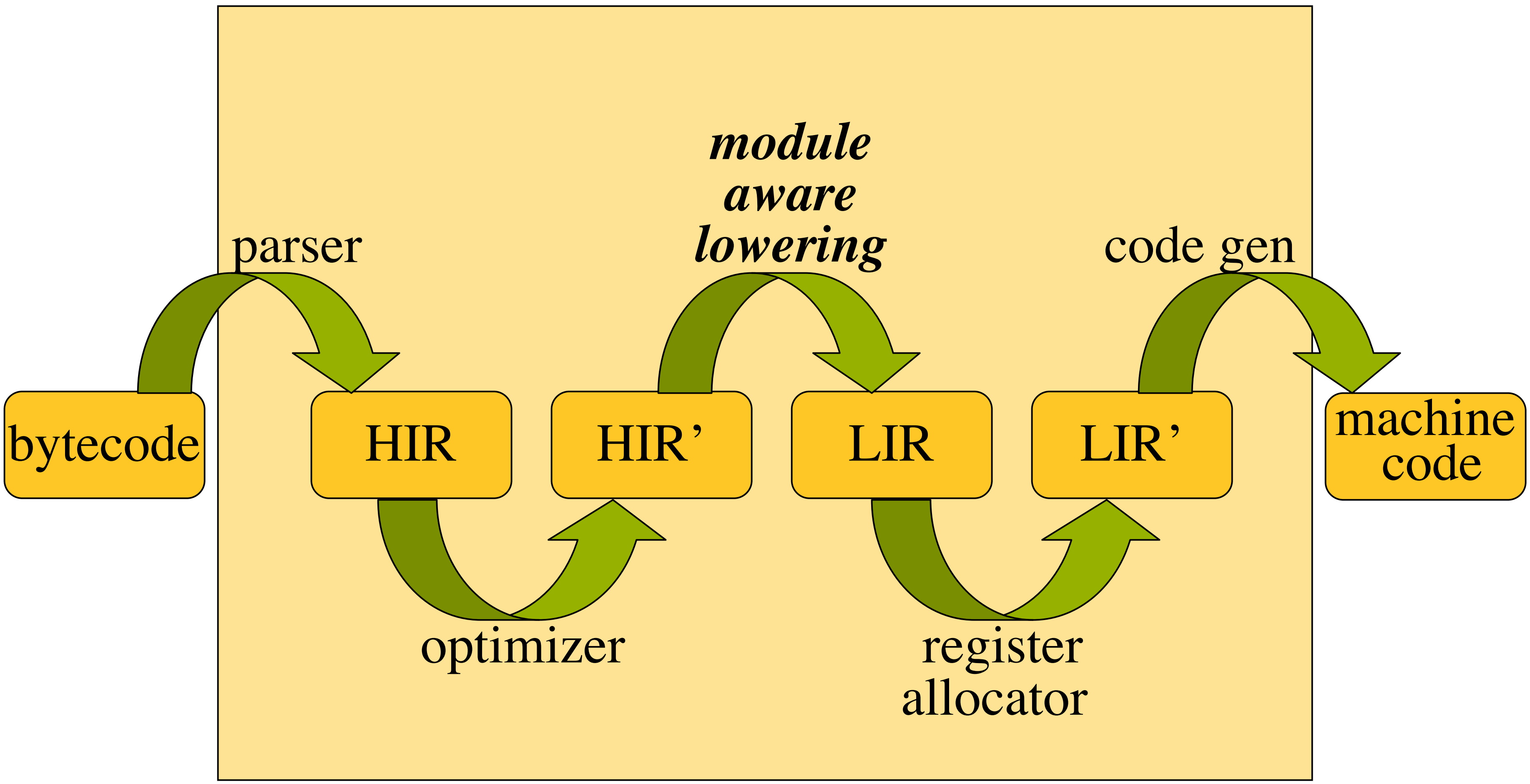
Doug Simon, Ben L. Titzer
Maxine Project, Sun Microsystems Laboratories

Virtual Machines and Intermediate Languages Workshop
Orlando, Florida, USA October 25, 2009

JVM Modules

- Modern, high performance JVMs are composed of complex modules
  - Read/write barriers, profile-driven optimizing compilers, fast synchronization, compressed oops, etc.
- Object based bytecode instruction
  - Abstract semantics in JVM Specification
  - Concrete implementation in terms of modules
- Binding to modules made by compiler
  - “Lowering phase”

Standard Compiler Anatomy



`putfield` in C1 (1 of 2)

```
void LIRGenerator::do_StoreField(StoreField* x) {
  LIRItem object(x->obj(), this);
  LIRItem value(x->value(),  this);
  object.load_item();
  value.load_for_store(field_type);
  set_no_result(x);
  LIR_Address* address = generate_address(object.result(),
                                          x->offset(),
                                          field_type);

  pre_barrier(LIR_OprFact::address(address));
  __ store(value.result(), address, info, patch_code);
  post_barrier(object.result(), value.result());
}
```

direct pointer/handle/compressed oop?

displacement over header?

GC uses write barrier?

putfield in C1 (1 of 2)

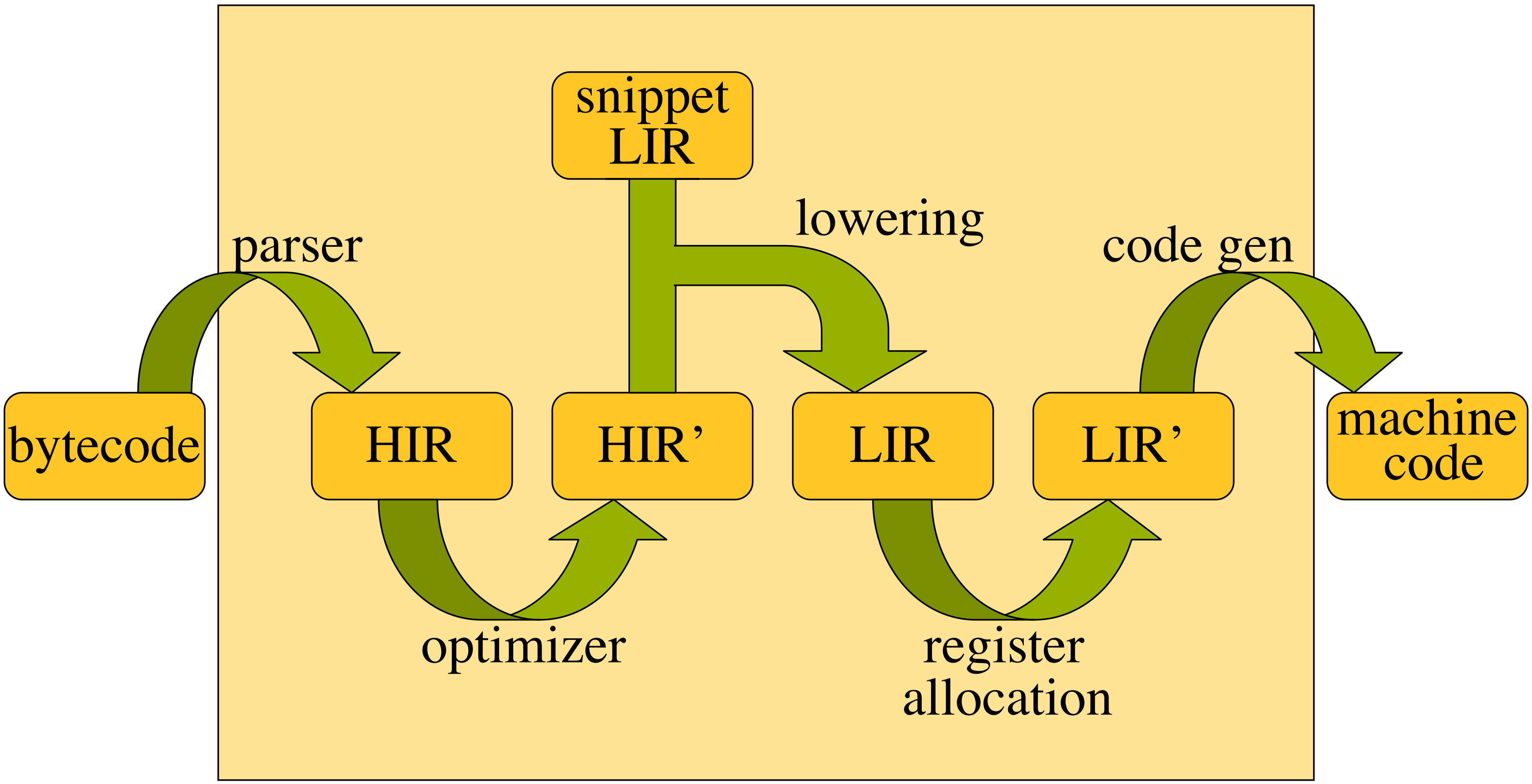
```
void LIRGenerator::post_barrier(LIR_OprDesc* addr,
                                LIR_OprDesc* new_val)
{
  __ move(addr, xor_res);
  __ logical_xor(xor_res, new_val, xor_res);
  __ move(xor_res, xor_shift_res);
  __ unsigned_shift_right(xor_shift_res, ...);
  __ cmp(lir_cond_notEqual, xor_shift_res, ...);
  CodeStub* slow = new G1PostBarrierStub(addr, new_val);
  __ branch(lir_cond_notEqual, T_INT, slow);
  __ branch_destination(slow->continuation());

}
```

fine grain control over code placement

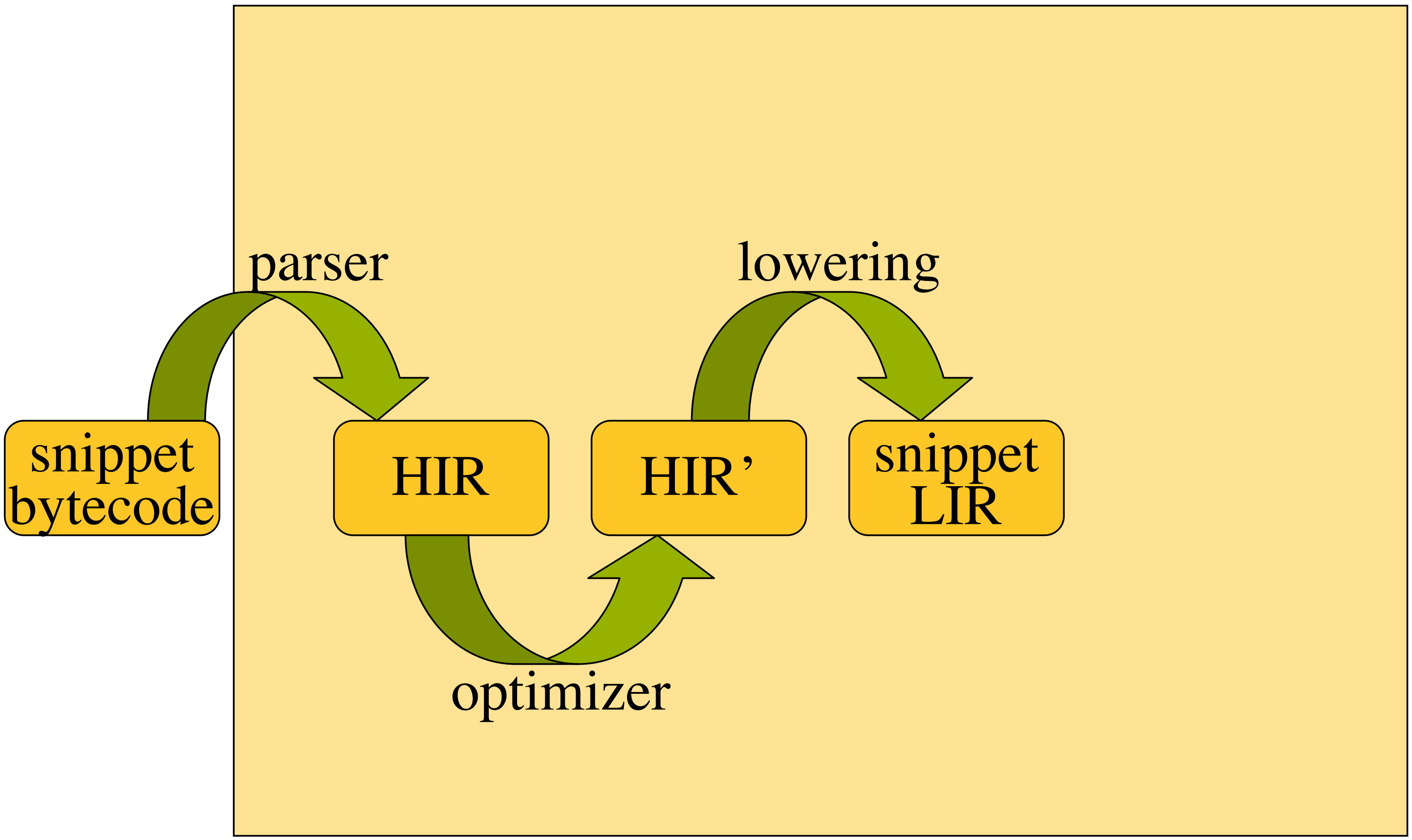
Isn't there a better way?

- Assume intrinsic compiler support for:
  - Integer, float, long, double arithmetic
  - Control flow
  - Calls to machine addresses with signature
  - Pointer operations
    - Exposed via Java API (e.g. `org.vmmagic.*`)
- Express object bytecode lowering in domain specific language (DSL)
  - Can access and manipulate VM data types and values
  - Reduced to compiler's LIR
    - Used to lower object bytecode HIR (for which no intrinsic LIR exists)
- VM written in Java $\Rightarrow$ DSL is Java!

Snippet-based Compiler Anatomy (normal)



Snippet-based Compiler Anatomy (snippet bootstrapping)



`putfield` with Snippets

How do we turn this…

```
@INLINE
static void writeReference(Object tuple,
                                          int fieldOffset, // resolved
                                          Object value)
{
   TupleAccess.writeObject(tuple, fieldOffset, value);
}
```

… into LIR?

Bootstrapping a Snippet (1 of 5)

```
TupleAccess.writeObject(tuple, fieldOffset, value);
```

TupleAccess.java

```
@INLINE
static void writeObject(Object tuple, int offset, Object value) {
    Reference tupleRef = Reference.fromJava(tuple);
    Reference valueRef = Reference.fromJava(value)
    tupleRef.writeReference(offset, valueRef);
}
```

```
Reference tupleRef = Reference.fromJava(tuple);
Reference valueRef = Reference.fromJava(value)
tupleRef.writeReference(fieldOffset, valueRef);
```

Bootstrapping a Snippet (2 of 5)

```
Reference tupleRef = Reference.fromJava(tuple);
Reference valueRef = Reference.fromJava(value)
tupleRef.writeReference(fieldOffset, valueRef);
```

Reference.java

```
@INLINE
static Reference fromJava(Object object) {
    return referenceScheme().fromJava(object);
}

@FOLD
static ReferenceScheme referenceScheme() {
    return VMConfiguration.referenceScheme();
}
```

HeapReferenceScheme

(reference module)

```
Reference tupleRef = heapReferenceScheme.fromJava(tuple);
Reference valueRef = heapReferenceScheme.fromJava(value)
tupleRef.writeReference(fieldOffset, valueRef);
```

Bootstrapping a Snippet (3 of 5)

```
Reference tupleRef = heapReferenceScheme.fromJava(tuple);
Reference valueRef = heapReferenceScheme.fromJava(value)
tupleRef.writeReference(fieldOffset, valueRef);
```

HeapReferenceScheme.java

```
@INLINE
static Reference fromJava(Object object) {
    return toReference(object);
}

@UNSAFE_CAST
static native Reference toReference(Object object);
```

```
Reference tupleRef = tuple;
Reference valueRef = value;
tupleRef.writeReference(fieldOffset, valueRef);
```

Bootstrapping a Snippet (4 of 5)

```
Reference tupleRef = tuple;
Reference valueRef = value;
tupleRef.writeReference(fieldOffset, valueRef);
```

Reference.java

```
@INLINE
void writeReference(int offset, Reference value) {
    referenceScheme().writeReference(offset, value);
}
```

HeapReferenceScheme.java

```
@INLINE
void writeReference(Reference reference, int offset, Reference value) {
    heapScheme().writeBarrier(reference, value);
    gripScheme().fromReference(reference).writeGrip(offset, value.toGrip());
}
```

`HeapScheme`

(GC module)

`GripScheme`

(object pointer module)

Bootstrapping a Snippet (5 of 5)

more inlining, folding steps…

```
Pointer tuplePtr = tuple;
Reference valueRef = value;
tuplePtr.writeReferenceAtIntOffset(fieldOffset, valueRef);
```

Pointer.java

```
@BUILTIN
native void writeReferenceAtIntOffset(int offset, Reference value);
```

LIR operation

What did we just see?

- *Not one line of compiler code*
- Compilation pragmas (annotations)
  - **@FOLD**, **@INLINE**, **@UNSAFE_CAST**, **@CONSTANT**, **@BUILTIN** ,
- Missing/undecided:
  - Fine grained code layout control
    - Refine **@INLINE** with a parameter?

```
enum {FAST_PATH, SLOW_PATH, STUB}
```

  - Performance, performance, performance!
    - Solution: C1X…

C1X Motivation

- Need better compiler
  - Design from scratch?
  - Adapt existing compiler?
- Preserve Maxine Design Focus
  - Compiler / runtime separation
  - Flexibility
  - Reduce runtime implementer burden

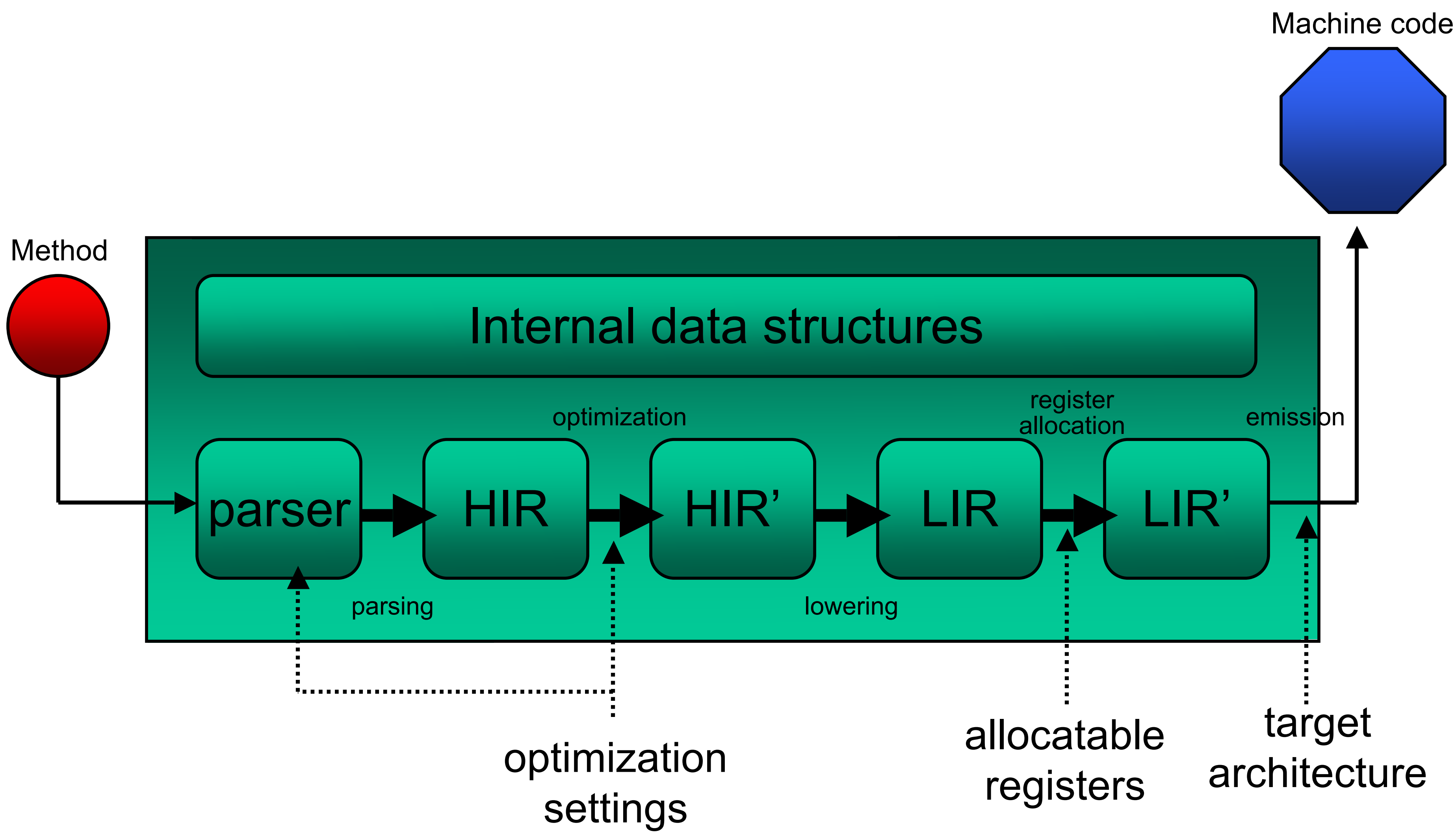
C1: HotSpot™ Client Compiler

- As opposed to "-server"
  - Faster compilation times, better startup
  - Default on smaller machines, Java 5
  - Simpler IR, architecture
- Supports
  - Inlining, speculative optimization, deoptimization
  - Precise GC, including all HotSpot collectors
  - Compressed Oops
- Proven research vehicle
  - Object co-location and inlining, bounds check elimination, hot swapping, multi-tasking

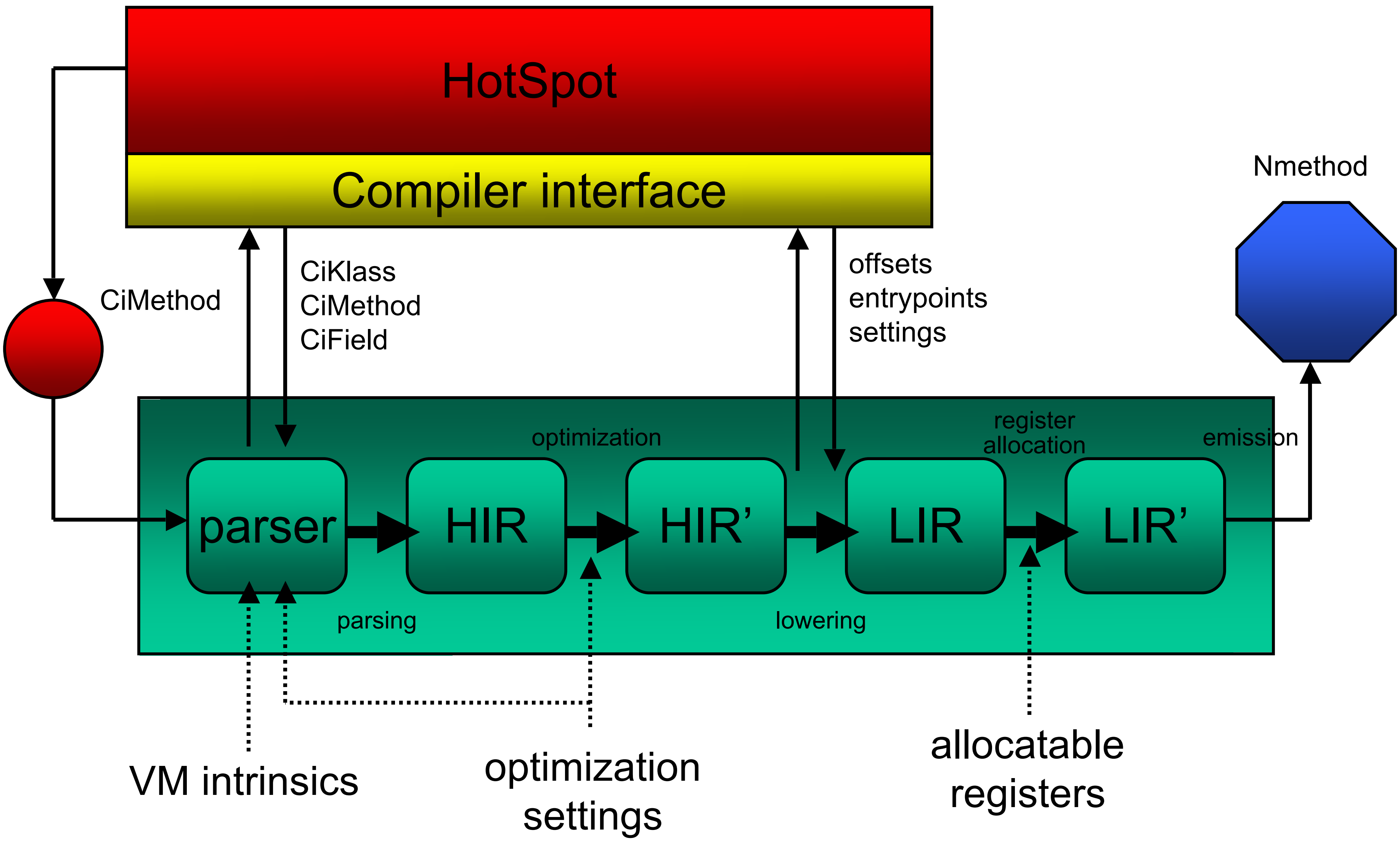
C1X

- Rewrote C1 in Java (for Maxine)
  - Improved compiler / runtime separation
- "Less wrinkly" front end
  - Major benefits from Java language
  - Better documentation
- Redesigned backend around XIR
  - Reduced backend complexity
  - VM-independent, programmable lowering

Standard Compiler Anatomy

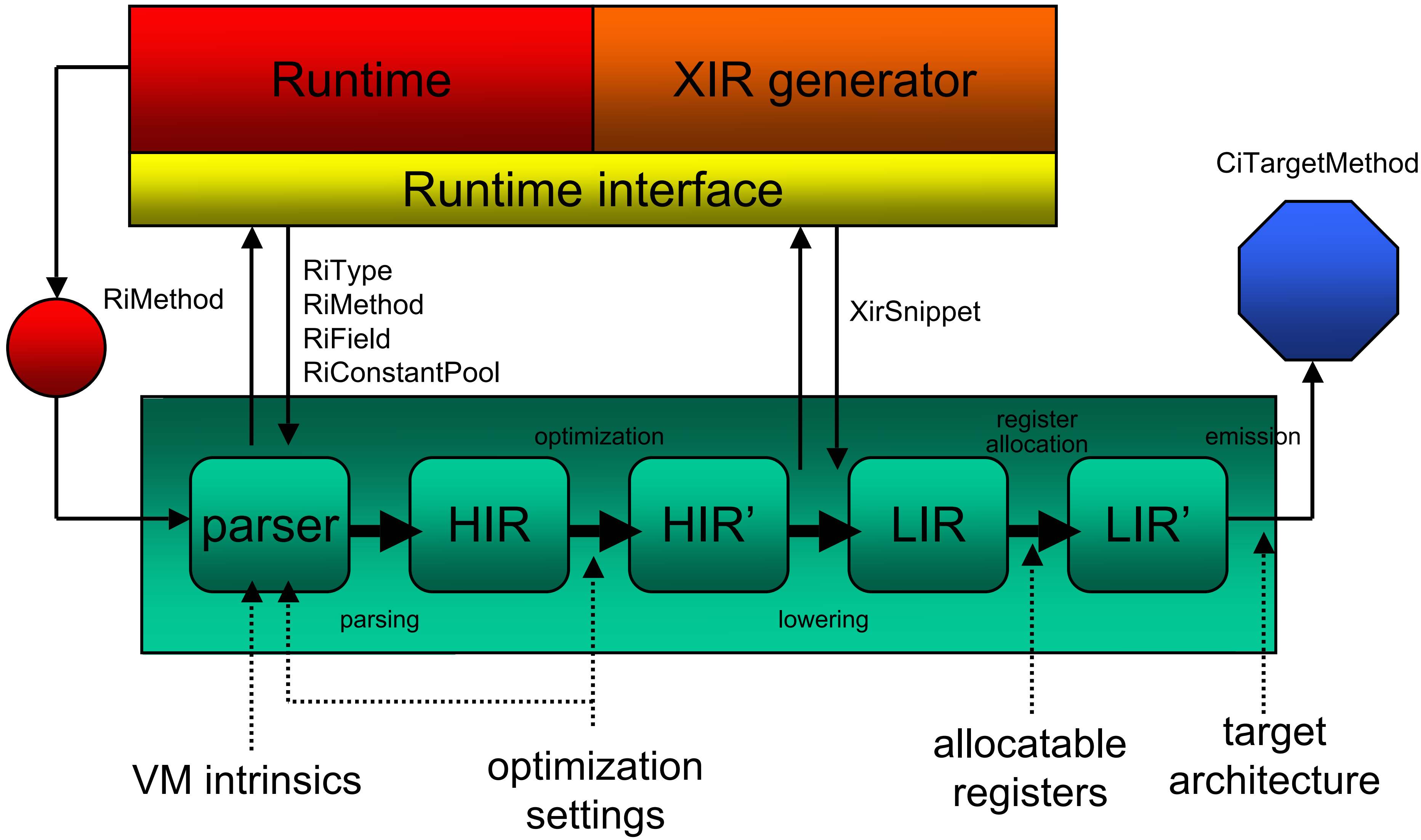


C1 Anatomy

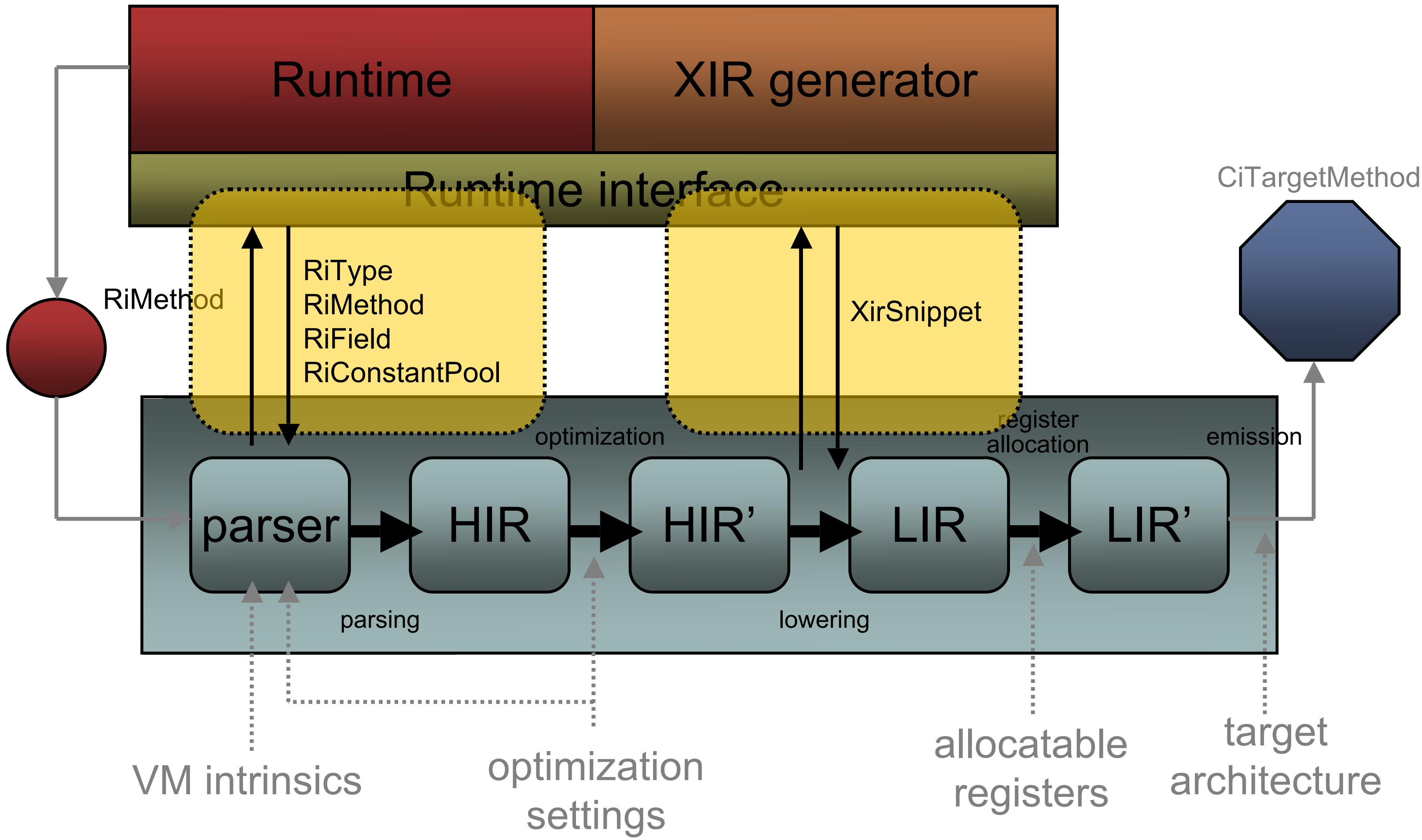


HotSpot
Compiler interface
CiMethod
CiKlass
CiMethod
CiField
offsets
entrypoints
settings
Nmethod
optimization
register allocation
emission
parser
HIR
HIR'
LIR
LIR'
parsing
lowering
VM intrinsics
optimization settings
allocatable registers

C1X Anatomy


Runtime
XIR generator
Runtime interface
CiTargetMethod
RiMethod
RiType
RiMethod
RiField
RiConstantPool
XirSnippet
optimization
register
allocation
emission
parser
HIR
HIR’
LIR
LIR’
parsing
lowering
VM intrinsics
optimization
settings
allocatable
registers
target
architecture

Compiler Runtime Interaction



Front end

- Parses bytecodes to create SSA-form HIR
  - Abstract interpreter models Java frame state
    - Produces embedded value dependence graph
  - Eager local optimizations
    - Devirtualization, inlining, constant folding, strength reduction, local value numbering, load elimination
- HIR operations are close to Java
  - Invokevirtual, getfield, monitorenter
  - Java type information preserved
    - RiField, RiMethod, RiType

Global Optimization

- Null check elimination
  - Iterative, flow-sensitive
- Block merging
  - Straighten control flow
- Global value numbering
  - Lift common subexpressions
- Dead code elimination

Backend

- Compute good block order
- Convert out of SSA form
- Lower object operations
  - VM-configurable with XIR
- Linear Scan Register Allocation
  - Architectural constraints (e.g. x86)
  - Calling convention

Code Generation

- Additional optimizations, then emit code
- Ultimate result: CiTargetMethod
  - machine code as byte array
  - Locations of:
    - calls, safepoints, implicit exception points, scalar and reference constants
  - Debug (deoptimization) information
  - Reference maps
  - Compilation statistics

Feature Dependency

VM / Compiler interactions aren't easy to factor into interfaces due to feature interdependence

- c Class Layout
- o Object Layout
- g Garbage Collection
- s Stack Model
- i Instrumentation
- x Synchronization

c	o	g	s	i	x	
			s	i		Entrypoint
		g		i		Safepoint
		g	s	i		Return
c				i		ResolveClass
c	o	g		i		GetField, PutField
	o			i		ArrayLength
	o	g		i		ArrayLoad, ArrayStore
c	o		s	i		Invoke
	o			i		Intrinsic
c	o	g	s	i		NewInstance, NewArray
c	o			i		CheckCast, InstanceOf
	o		s	i	x	MonitorEnter / Exit
			s	i		ExceptionObject

C1X Solution - XIR

- During lowering, C1X requests code from VM
  - VM returns XIR code “snippet”
  - Primitive operations left to compiler
- Two-step process
  - Before compilation: VM creates XIR template
  - During compilation: VM instantiates XIR template as XIR snippet

XIR Language

- Low-level, machine-independent
- RISC-like operations
  - Three-address add, sub, mul, div, shift, etc
  - Pointer load, store, compare-and-swap
  - CallJava, CallStub, CallRuntime
- Virtual and physical registers
  - Allows access to VM-specific thread locals
- Distinguishes primitives from objects
  - Allows for precise GC maps
  - Unsafe values cannot be live across safepoints

XIR Assembler Interface

```
public interface CiXirAssembler {
     XirVariable createInputParameter(CiKind kind);
     XirVariable createConstantInputParameter(CiKind kind);
     XirVariable createTemporary(CiKind kind);
     XirVariable createConstant(CiConstant constant);
     XirLabel createInlineLabel();
     XirLabel createOutOfLineLabel();
     void add(XirVariable result, XirVariable a, XirVariable b);
     void sub(XirVariable result, XirVariable a, XirVariable b);
     void mul(XirVariable result, XirVariable a, XirVariable b);
     . . .
     void move(XirVariable dest, XirVariable a);
     void pload(CiKind kind, XirVariable dest, XirVariable ptr);
     void pstore(CiKind kind, XirVariable ptr, XirVariable a);
     void bind(XirLabel label);
     void jump(XirLabel label);
     void jeq(XirLabel label);
     . . .
     void callJava(XirVariable addr);
     void callStub(XirTemplate template, XirVariable result, . . .);
     void callRuntime(XirTemplate template, XirVariable result, . . .);
     XirTemplate finishStub(String name);
     XirTemplate finishTemplate(String name);
}
```

XIR Template Example

```java
public XirTemplate buildResolvedGetField(XirAssembler asm, CiKind kind) {
    // resolved case
    asm.start(kind);
    XirParameter object = asm.createInputParameter("object", CiKind.Object);
    XirParameter fieldOffset = asm.createConstantInputParameter("fieldOffset",
            CiKind.Int);
    XirVariable resultOperand = asm.getResultOperand();
    asm.pload(kind, resultOperand, object, fieldOffset, true);
    return asm.finishTemplate("getfield<" + kind + ">");
}
```

XIR Generator Interface (1 of 2)

```
public interface RiXirGenerator {
    XirSnippet genEntrypoint(. . .);
    XirSnippet genSafepoint(. . .);
    XirSnippet genReturn(. . .);
    XirSnippet genResolveClassObject(. . .);
    XirSnippet genIntrinsic(. . .);
    XirSnippet genGetField(. . .);
    XirSnippet genPutField(. . .);
    XirSnippet genGetStatic(. . .);
    XirSnippet genPutStatic(. . .);
    XirSnippet genMonitorEnter(. . .);
    XirSnippet genMonitorExit(. . .);
    XirSnippet genNewInstance(. . .);
    XirSnippet genNewArray(. . .);
    XirSnippet genNewMultiArray(. . .);
    XirSnippet genCheckCast(. . .);
    XirSnippet genInstanceOf(. . .);
    XirSnippet genInvokeInterface(. . .);
    XirSnippet genInvokeVirtual(. . .);
    XirSnippet genInvokeSpecial(. . .);
    XirSnippet genInvokeStatic(. . .);
    XirSnippet genArrayLoad(. . .);
    XirSnippet genArrayStore(. . .);
    XirSnippet genArrayLength(. . .);
}
```

XIR Generator Interface (1 of 2)

```
public interface RiXirGenerator {
    . . .
    XirSnippet genGetField(XirArgument object,
                           RiField field);
    XirSnippet genPutField(XirArgument object,
                           XirArgument value,
                           RiField value);
    XirSnippet genNewInstance(RiType type);
    XirSnippet genNewArray(XirArgument length,
                           RiType arrayType);
    XirSnippet genInvokeVirtual(XirArgument receiver,
                                RiMethod method);
    XirSnippet genInvokeSpecial(XirArgument receiver,
                                RiMethod method);
    . . .
}
```

XIR Snippet Example

```
public MaxXirGenerator implements RiXirGenerator {

@Override
public XirSnippet genGetField(XirArgument object, RiField field) {
    XirPair pair = getFieldTemplates[field.kind().ordinal()];
    if (field.isLoaded()) {
        XirArgument offset = XirArgument.forInt(field.offset());
        return new XirSnippet(pair.resolved, receiver, offset);
    }
    XirArgument guard = XirArgument.forObject(guardFor(field));
    return new XirSnippet(pair.unresolved, receiver, guard);
}

}
```

Complications

- Fast / slow paths
  - Inline, out-of-line, fast-path global stub, slow-path global stub, runtime call
- XIR preprocessing
  - Gather architectural register constraints
  - Recognize addressing modes
- Attaching debug information
- Code patching, guards

C1X Status

- Runtime interface for Maxine
  - 13 classes, 5300 lines of Java code
    - 1400 lines in MaxXirGenerator
- C1X passes all our regression suite
  - Some debugging needed with inlining
- XIR fully implemented
  - More tuning needed
  - Measuring compile time / quality now
- More in VEE 2010 submission

C1X Story Continues…

- Separated C1X from VM details
  - More firm runtime interface
  - XIR achieves VM-configurable lowering
  - Still heavily tied to Java language
- Planning back-port to HotSpot
  - Runtime interface + XIR
- Snippets!

Acknowledgements

Mario Wolczko – manager

Team:

- Paul Caprioli
- Laurent Daynès
- Michael Van De Vanter

Former:

- Bernd Mathiske
- Greg Wright

Interns:

Athul Acharya
Aritra Bandyopadhyay
Michael Bebenita
Marcelo Cintra
Abdulaziz Ghuloum
Yi Guo
Christos Kotselidis
David Liu
Karhik Manivannan
Sumeet Panchal
Hannes Payer
Sunil Soman
Simon Wilkinson
Thomas Würthinger
Hiroshi Yamauchi

Others: David Ungar, Tom Rodriguez, Christian Wimmer, Robert Griesemer, Srdjan Mitrovic, Ken Russell

The Maxine Project

research.sun.com/projects/maxine
maxine.kenai.com

Doug.Simon@sun.com
Ben.Titzer@sun.com