Dynamic Program Analysis in Jikes RVM

Harry Xu
May 2012
Complex, concurrent software

Precision (no false positives)
Find real bugs in real executions
Why Jikes RVM?

Need to modify JVM
(e.g., object layout, GC, or ISA-level code)

Need to demonstrate realism
(usually performance)
Otherwise use RoadRunner, BCEL, Pin, LLVM, ...
Keeping track of stuff as the program executes?

- Change application behavior (add instrumentation)
- Store per-object/per-field metadata
- Piggyback on GC
What is dynamic analysis?

Keeping track of stuff as the program executes?

- JVM written in Java?!
- Change application behavior (add instrumentation)
- Store per-object/per-field metadata
- Piggyback on GC
- Uninterruptible code
Resources (jikesrvm.org)

Jikes RVM

- Guide
- Research Archive
- Research mailing list
Resources (jikesrvm.org)

Jikes RVM

- Guide
- Research Archive
- Research mailing list
JVM written in Java?!

- Jikes RVM source code
  - Boot image writer
  - Dynamic compilers
JVM written in Java?!

Jikes RVM source code

- Boot image writer
- Dynamic compilers

Run with another JVM
JVM written in Java?!
JVM written in Java?!

- Jikes RVM source code
- Boot image writer
- Dynamic compilers
- Boot image (native code + initial heap space)

Build configurations:
- Base
- BaseAdaptive
- FullAdaptive
- FastAdaptive

Run with another JVM
JVM written in Java?!

Jikes RVM source code

Boot image writer

Dynamic compilers

Run with another JVM

Build configurations:
- BaseBase (prototype)
- BaseAdaptive (prototype-opt)
- FullAdaptive (development)
- FastAdaptive (production)

Boot image
(native code + initial heap space)
JVM written in Java?!

Jikes RVM source code

Boot image writer

Dynamic compilers

Run with another JVM

Build configurations:
- Base
- BaseAdaptive
- Full
- Fast
- Adaptive

Testing

Boot image (native code + initial heap space)
JVM written in Java?!

Build configurations:
- Base
- BaseAdaptive
- FullAdaptive
- FastAdaptive

Faster builds

Jikes RVM source code

Boot image writer

Run with another JVM

Dynamic compilers

Boot image (native code + initial heap space)
JVM written in Java?!

- Jikes RVM source code
- Boot image writer
- Dynamic compilers
- Run with another JVM

Build configurations:
- Base
- BaseAdaptive
- FullAdaptive
- FastAdaptive

Boot image (native code + initial heap space)

Faster runs
JVM written in Java?!

Jikes RVM source code

Build configurations:
- Base
- BaseAdaptive
- FullAdaptive
- FastAdaptive

Dynamic compilers

Run with another JVM

Boot image writer

Boot image (native code + initial heap space)

Performance
JVM written in Java?!

Jikes RVM source code

Boot image writer

Dynamic compilers

Run with another JVM

Initialize objects & classes

Boot image (native code + initial heap space)
JVM written in Java?!

Jikes RVM source code

- Boot image writer
- Dynamic compilers

Run with another JVM

Initialize objects & classes

Boot image
(native code + initial heap space)

Edit with Eclipse
(see Guide)
What is dynamic analysis?

Keeping track of stuff as the program executes?

- Change application behavior (add instrumentation)
- Store per-object/per-field metadata
- Piggyback on GC
Change application behavior (add instrumentation)

- Bytecode
  - Baseline compiler
  - Native code
Change application behavior (add instrumentation)

Each bytecode $\rightarrow$ several x86 instructions

(BaselineCompilerImpl.java)
Change application behavior (add instrumentation)

```java
/**
 * Emit code to implement a getfield
 * @param fieldRef the referenced field
 */
@Override
protected final void emit_resolved_getfield(FieldReference fieldRef) {

  RVMField field = fieldRef.peekResolvedField();
  Offset fieldOffset = field.getOffset();

  TypeReference fieldType = fieldRef.getFieldContentsType();

  if (field.isReferenceType()) {
    // 32/64bit reference load
    if (NEEDS_OBJECT_GETFIELD_BARRIER && !field.isUntraced()) {
      Barriers.compileGetfieldBarrierImm(asm, fieldOffset, fieldRef.getId());
    } else {
      asm.emitPOP_Reg(T0); // T0 is object reference
      asm.emitPUSH_RegDisp(T0, fieldOffset); // place field value on stack
    }
  } else if (fieldType.isBooleanType()) {
    // 8bit unsigned load
    asm.emitPOP_Reg(S0); // S0 is object reference
  }
```
Change application behavior (add instrumentation)

```java
/**
 * Emit code to implement a getfield
 * @param fieldRef the referenced field
 */
@override
protected final void emit_resolved_getfield(FieldReference fieldRef) {
    RVMField field = fieldRef.peekResolvedField();
    Offset fieldOffset = field.getOffset();

    TypeReference fieldType = fieldRef.getFieldContentsType();

    if (!method.getDeclaringClass().get>TypeRef().getName().isBootstrapClassDescriptor()) {
        genParameterRegisterLoad(asm, 1); // T0 <- [SP]
        asm.emitPUSH_Reg(T0);
        asm.emitCALL_Abs(Magic.getTocPointer().plus(EntryPoints.fieldReadAnalysisMethod.getOffset()));
    }

    if (field.isReferenceType()) {
        // 32/64bit reference load
        if (NEEDS_OBJECT_GETFIELD_BARRIER && !field.isUntraced()) {
            Barriers.compileGetfieldBarrierImm(asm, fieldOffset, fieldRef.getId());
        } else {
            asm.emitPOP_Reg(T0); // T0 is object reference
            asm.emitPUSH_RegDisp(T0, fieldOffset); // place field value on stack
        }
    } else if (fieldType.isBooleanType()) {
        // 8bit unsigned load
       asm.emitPOP_Reg(C0); // C0 is object reference
    }
```
Change application behavior (add instrumentation)
Change application behavior (add instrumentation)

- Bytecode
  - Baseline compiler
  - Optimizing compiler
  - (Faster) native code
  - Native code

- Profiling
  - Adaptive optimization system
Change application behavior (add instrumentation)

[Bytecode] → [Baseline compiler] → [Native code]

[Bytecode] → [Optimizing compiler] → [(Faster) native code]

[Profiling] → [Adaptive optimization system]
Change application behavior (add instrumentation)

Bytecode

Optimizing compiler

(Faster) native code
Change application behavior (add instrumentation)

- Bytecode
- Resembles bytecode

- HIR
- Resembles typical compiler IR (3-address code)

- LIR
- Resembles assembly code

- MIR
- (Faster) native code

- (Faster) native code
Change application behavior (add instrumentation)

Opt levels: 0, 1, 2
Change application behavior (add instrumentation)

- Bytecode
  - ExpandRuntimeServices.java
    - Add instrumentation at reads, writes, allocation, synchronization
  - HIR
  - LIR
  - MIR
    - (Faster) native code
Change application behavior (add instrumentation)

case GETFIELD_opcode: {
  // MMMM: read/write instrumentation
  if (Mmmm.insertReadOfObjectHeaders()) {
    RVMMethod method = inst.position.getMethod();
    if (method != null) {
      if (VM.runningVM &&
          !method.getDeclaringClass().getDescriptor().isBootstrapClassDescriptor()) {
        Operand refOperand = inst.getOperand(1);
        RVMMethod target = Entrypoints.mmmmReadInstrumentationMethod;

        Instruction call = |
        Call.createI(CALL,
          null,
          IRTools.AC(target.getOffset()),
          MethodOperand.STATIC(target),
          refOperand.copy());

        call.bcIndex = RUNTIME_SERVICES_BCI;
        call.position = inst.position;
        inst.insertBefore(call);
        inline(call, ir);
      }
    }
  }
}

if (NEEDS_OBJECT_GETFIELD_BARRIER) {
  LocationOperand loc = GetField.getLocation(inst);
  FieldReference fieldRef = loc.getFieldRef();
  if (GetField.getResult(inst).getTyp() isReferenceType()) {

  }
}
Change application behavior
(add instrumentation)

case GETFIELD_opcode: {
    // MMMM: read/write instrumentation
    if (Mmm.defInsertReadOfObjectHeaders()) {
        RVMMMethod method = inst.position.getMethod();
        if (method != null) {
            if (VM.runningVM && !method.getDeclaringClass().getDescriptor().isBootstrapClassDescriptor()) {
                Operand refOperand = inst.getOperand(1);
                RVMMMethod target = Entrypoints.mmmReadIntrumentationMethod;
                Instruction call = |
                    Call.createI(CALL,
                        null,
                        IRTools.AC(target.getOffset()),
                        MethodOperand.STATIC(target),
                        refOperand.copy());
                call.bcIndex = RUNTIME_SERVICES_BCI;
                call.position = inst.position;
                inst.insertBefore(call);
                inline(call, ir);
            }
        }
    }
}

if (NEEDS_OBJECT_GETFIELD_BARRIER) {
    LocationOperand loc = GetField.getLocation(inst);
    FieldReference fieldRef = loc.getFieldRef();
    if (GetField getResult(inst).getReturnType().isReferenceType()) {
```java
@Entrypoint
static final void readInstrumentation(Object o) {
    /* What do I put here? */
}
```
```java
if (Operand refOperand = inst.getOperand(1);
    RVMMethod target = Entrypoints.mmmmReadInstrumentationMethod;
    Instruction call = |
    Call.create1(CALL,
        null,
        IRTools.AC(target.getOffset()),
        MethodOperand.STATIC(target),
        refOperand.copy());
    call.bcIndex = RUNTIME_SERVICES_BCI;
    call.position = inst.position;
    inst.insertBefore(call);
    inline(call, ir);
}{
}
if (NEEDS_OBJECT_GETFIELD_BARRIER) {
    LocationOperand loc = getField getLocation(inst);
    FieldReference fieldRef = loc.getFieldRef();
    if (GetField getResult(inst) getType() isReferenceType()) {
```
What is dynamic analysis?

Keeping track of stuff as the program executes?

- Change application behavior (add instrumentation)
- Store per-object/per-field metadata
- Piggyback on GC
Object layout

Low address

High address

header field0 field1 field2
Object layout

- type info block
- locking & GC
- field0
- field1
- field2

Object reference
Object layout

- Type info block
- Locking & GC
- Field 0
- Field 1
- Field 2

- Type info block
- Locking & GC
- Array length
- Element 0
- Element 1
Extra header bits

- type info block
- locking & GC
- field0
- field1
- field2

Object reference

Steal bits
Extra header word

misc | type info block | locking & GC | field0 | field1 | field2

Object reference

MiscHeader.java
Extra header word

```java
@EntryPoint
static final void readInstrumentation(Object o) {
    int oldValue = ObjectReference.fromObject(o).toAddress().loadInt(MiscHeader.COUNTER_OFFSET);
    int newValue = oldValue + 1;
    ObjectReference.fromObject(o).toAddress().store(newValue, MiscHeader.COUNTER_OFFSET);
}
```
Extra header word

@EntryPoint
static final void readInstrumentation(Object o) {
    int oldValue = ObjectReference.fromObject(o).toAddress().loadInt(MiscHeader.COUNTER_OFFSET);
    int newValue = oldValue + 1;
    ObjectReference.fromObject(o).toAddress().store(newValue, MiscHeader.COUNTER_OFFSET);
}

Magic!

Compiles down to three x86 instructions
Extra header word

```
@EntryPoint
static final void readInstrumentation(Object o) {
    int oldValue = ObjectReference.fromObject(o).toAddress().loadInt(MiscHeader.COUNTER_OFFSET);
    int newValue = oldValue + 2;
    ObjectReference.fromObject(o).toAddress().store(newValue, MiscHeader.COUNTER_OFFSET);
}
```

Gotcha: can’t actually use LSB of leftmost word
What's the problem with this code?
```java
@EntryPoint
class final void readInstrumentation(Object o) {
    int oldValue;
    int newValue;
    do {
        oldValue = ObjectReference.fromObject(o).toAddress().prepareInt(MiscHeader.COUNTER_OFFSET);
        newValue = oldValue + 2;
    } while (!ObjectReference.fromObject(o).toAddress().attempt(oldValue, newValue, MiscHeader.COUNTER_OFFSET));
}
Thread-local data

@Entrypoint
static final void readInstrumentation(Object o) {
    RVMThread.getCurrentThread().perThreadReadCounter++;
}
Thread-local data

@Entrypoint
static final void readInstrumentation(Object o) {
    RVMThread.currentThread().perThreadReadCounter++;
}

Compiles down to three x86 instructions
Extra header word

Object reference

misc  type info block  locking & GC  field0  field1  field2
Extra header word

- misc
- type info block
- locking & GC
- field0
- field1
- field2

What if GC moves object?
What if GC collects object?
What is dynamic analysis?

Keeping track of stuff as the program executes?

- Change application behavior (add instrumentation)
- Store per-object/per-field metadata
- Piggyback on GC
// Initially worklist populated with roots
while worklist has elements
    Object obj = worklist.pop()
    foreach reference field obj.f
        obj.f = markAndPossiblyCopy(obj.f)
    worklist.push(obj.f)
Tracing existing pointers

// Initially worklist populated with roots
while worklist has elements
    Object obj = worklist.pop()
    foreach reference field obj.f
        obj.f = markAndPossiblyCopy(obj.f)
    worklist.push(obj.f)
// Initially worklist populated with roots
while worklist has elements
  Object obj = worklist.pop()
  foreach reference field obj.f
    obj.f = markAndPossiblyCopy(obj.f)
  worklist.push(obj.f)
// Initially worklist populated with roots
while worklist has elements
    Object obj = worklist.pop()
    foreach reference field obj.f
        obj.f = markAndPossiblyCopy(obj.f)
        worklist.push(obj.f)
    obj.misc = markAndPossiblyCopy(obj.f)
    worklist.push(obj.misc)
// Initially worklist populated with roots
while worklist has elements
    Object obj = worklist.pop()
    foreach reference field obj.f
        obj.f = markAndPossiblyCopy(obj.f)
        worklist.push(obj.f)
    obj.misc = markAndPossiblyCopy(obj.f)
    worklist.push(obj.misc)
// Initially worklist populated with roots
while worklist has elements
    Object obj = worklist.pop()
    foreach reference field obj.f
        obj.f = markAndPossiblyCopy(obj.f)
        worklist.push(obj.f)
    obj.misc = markAndPossiblyCopy(obj.f)
    worklist.push(obj.misc)

TraceLocal.scanObject()
// Initially worklist populated with roots
while worklist has elements

    Object obj = worklist.pop()
    foreach reference field obj.f
        obj.f = markAndPossiblyCopy(obj.f)
    worklist.push(obj.f)
// Initially worklist populated with roots
while worklist has elements
    Object obj = worklist.pop()
    foreach reference field obj.f
        obj.f = markAndPossiblyCopy(obj.f)
    worklist.push(obj.f)

TraceLocal.processNode()
What is dynamic analysis?

Keeping track of stuff as the program executes?

- Change application behavior (add instrumentation)
- Store per-object/per-field metadata
- Piggyback on GC
- Uninterruptible code
Uninterruptible code

- Normal application code can be interrupted
  - Allocation → GC
  - Synchronization & yield points → join a GC
- Some VM code shouldn’t be interrupted
  - Heap etc. in inconsistent state
- Most instrumentation can’t be interrupted
  - Reads & writes aren’t GC-safe points
@Uninterruptible
class MyClass {

    @Uninterruptible
    static void myMethod(Object o) {

        // No allocation or synchronization

        // No calls to interruptible methods

    }

}
@Uninterruptible
class Uninterruptible {
    @Uninterruptible
    static void myMethod(Object o) {
        currentThread.deferGC = true;
        Metadata m = new Metadata();
        currentThread.deferGC = false;
        setMiscHeader(o, offset, m);
    }
}
Need to modify JVM internals
Need to demonstrate realism

Jikes RVM

- Guide
- Research Archive
- Research mailing list

Overview of other tasks & components
Dynamic analysis examples
Help (especially for novices)
Notes

- Object layout
  - Extra bits or words in header
  - Stealing bits from references
  - Discuss magic here
- Adding instrumentation
  - Baseline & optimizing compilers
  - Allocation sites; reads & writes
  - Inlining instrumentation
- Garbage collection
  - Piggybacking on GC
  - New spaces
- Low-level stuff
  - Uninterruptible code
  - Walking the stack
- Concurrency
  - Atomic stores
  - Thread-local data