Jikes Intermediate Code Representation

Presentation By: Shane A. Brewer
March 20, 2003
Outline

- Jikes RVM Overview
- Intermediate Representation Overview
- Implementation
- Examples
  - Bytecode to HIR
  - HIR to LIR
  - LIR to MIR
  - MIR to Machine Code
Jikes RVM Overview

- Jikes Research Virtual Machine (Jikes RVM)
- Not a full JVM as it is missing libraries (AWT, Swing, J2EE)
- Implemented in the Java Programming Language
- Contains 3 compilers
  - Baseline: Generates code that is “obviously correct”
  - Optimizing: Applies a set of optimizations to a class when it is loaded at runtime
  - Adaptive: Methods are compiled with a non-optimizing compiler first and then selects “hot” methods for recompilation based on run-time profiling information.
Intermediate Representation

Overview

- The Intermediate Representation (IR) used by Jikes is a register based IR
  - Allows more effective machine-specific optimizations
- An IR instruction is an N-tuple, consisting of an operator and some number of operands
  - An Operator is the instruction to perform
  - Operands are used to represent:
    - Symbolic Register
    - Physical Registers
    - Memory Locations
    - Constants
    - Branch targets
    - Method Signatures
    - Types
    - etc
IR Overview

- Java type information preserved
- Java specific operators and optimizations
- Also include space for caching of optional auxiliary information such as:
  - Reaching definition sets
  - Dependence Graphs
  - Encoding of loop-nesting structure.
Levels of IR

- 3 levels of IR
  - HIR (High Level IR)
    - Operators similar to Java bytecode
      - Example: ARRAYLENGTH, NEW, GETFIELD, BOUNDS_CHECK, NULL_CHECK
    - Operate on symbolic registers instead of an implicit stack
    - Contains separate operators to implement explicit checks for run-time exceptions (e.g., array-bounds checks)
  - LIR (Low Level IR)
    - Details of Jikes runtime and object layout
      - Example: GET_TIB (vtable), GET_JTOC (static), INT_LOAD (for getfield)
    - Expands complicated HIR structures such as TABLE_SWITCH
  - MIR (Machine Specific IR)
    - Similar to assembly code
    - Details of target architecture are introduced
    - Register Allocation is performed on MIR
IR Diagram

Translation From Bytecode to HIR

Optimization of HIR

Optimized HIR

Translation From HIR to LIR

Optimization of LIR

Optimized LIR

Translation From LIR To MIR

Optimization of MIR

Optimized MIR

Final Assembly

Binary Code
JTOC and TIB

- A TIB (Type Information Block) in Jikes included in an object header.
  - Is an array of Java object references
  - It’s first component describes the object’s class
  - The remaining components are compiled method bodies for the virtual methods of the class
  - Acts as the virtual method table

- The JTOC (Jalapeno Table of Contents) is an array that stores
  - all static fields
  - references to all static method bodies
To print out the Intermediate Representation produced by Jikes, the following command line options are available using the command:

-\texttt{X:opt: <option>=true}:
  - \texttt{print\_all\_ir}: Prints the IR after each compiler phase
  - \texttt{high}: Prints the IR after initial generation
  - \texttt{final\_hir}: Prints the IR just before conversion to LIR
  - \texttt{low}: Prints the IR after conversion to LIR
  - \texttt{final\_lir}: Prints the IR just before conversion to MIR
  - \texttt{mir}: Prints the IR after conversion to MIR
  - \texttt{final\_lir}: Prints the IR just before conversion to machine code
  - \texttt{mc}: Prints the final machine code
  - \texttt{cfg}: Prints the control flow graph
Extended Basic Blocks [3]

- IR instructions are grouped into “Extended Basic Blocks”
- Method calls and potential trap sites do not end basic blocks
- Thus extra care is needed when performing data flow analysis or code motion
- Are constructed during translation from Java Bytecode to HIR
Extended Basic Block Example

Java Source

```java
public Circle foo(Circle p, Circle[] a) {
    try {
        int n = Example.bar();
        p = Example.getNewCircle(a[n]);
    } catch (NullPointerException e) {
        ...
    } catch (Exception e) {
        ...
    } // End try-catch
    return p;
} // End method foo
```

Generic IR

```
1a: PEI call_static n = Example.bar()
2a: PEI null_check a
2b: PEI bounds_check a, n
2c: ref_aload t0 = @{a, n}
2d: PEI call_static p = Example.getNewCircle(t0)
     end_block B0

1a: PEI call_static n = Example.bar()
2a: PEI null_check a
2b: PEI bounds_check a, n
2c: ref_aload t0 = @{a, n}
2d: PEI call_static p = Example.getNewCircle(t0)
     end_block B0

5a: ref_return p
     end_block B1

(java.lang.NullPointerException handler)

label B2
3a: ...
3b: goto B1
     end_block B2

(java.lang.Exception handler)

label B3
4a: ...
4b: goto B1
     end_block B3
```
Extended Basic Block Example:
Traditional Control Flow Graph

1a: PEI call_static n = Example.bar()
2a: PEI null_check a
2b: PEI bounds_check a, n
2c: ref_aload t0 = @{a, n}
2d: PEI call_static p = Example.getNewCircle(t0)

3a: ...
3b: goto B1

4a: ...
4b: goto B1

8 basic blocks
13 edges
Extended Basic Block Example:
Factored Control Flow Graph

label B0
1a: PEI call_static n = Example.bar()
2a: PEI null_check a
2b: PEI bounds_check a, n
2c: ref_aload t0 = @{a, n}
2d: PEI call_static p = Example.getNewCircle(t0)
   end_block B0

label B1
5a: ref_return p
   end_block B1

(java.lang.NullPointerException handler)
label B2
3a: ...
3b: goto B1
   end_block B2

(java.lang.Exeption handler)
label B3
4a: ...
4b: goto B1
   end_block B3

March 20, 2003
Shane A. Brewer
Implementation

- Each IR operator is defined in the class OPT_Operators
  - OPT_Operators is automatically generated from a template by a driver
- Driver takes 2 input files both named OperatorList.dat
  - /rvm/src/vm/compilers/optimizing/ir/instruction defines machine independent operators
  - /rvm/src/vm/arch/{arch}/compilers/optimizing/ir/instruction defines machine-dependent operators where {arch} specifies which architecture
OperatorList.dat File

- Each operator in OperatorList.dat is defined by a five-line record, consisting of:
  - SYMBOL: a static symbol to identify the operator
  - INSTRUCTION_FORMAT: The instruction format class that accepts this operator.
    - Every instance of OPT_Operator is assigned to exactly one Instruction Format class
    - Intended to represent the “syntactic form” of an instruction
  - TRAITS: A set of characteristics of the operator, composed with a bit-wise or (|) operator
  - IMPLDEFS: A set of registers implicitly defined by this operator; usually only for machine specific operators
  - IMPLUSES: A set of registers implicitly used by this operator; usually only for machine specific operators.
OperatorList.dat File Example

**INT_ADD**
- Integer Addition Operation
- Binary Instruction Format: 2 values and a return value
- None
- No Traits
- No implicit uses or definitions

**REF_IFCOMP**
- Conditional branch operator based on values of 2 references
- IfCmp Instruction Format: 2 Values, Condition, Target, Branch Profile, Guard Result
- Branch | conditional
- Branch and Conditional Traits
- No implicit uses or definitions

March 20, 2003 Shane A. Brewer
Addition Method Test Example: From Java Source To Java Bytecode

Java Source Code

class AdditionMethodTest {
   public static void main(String args[]) {
      int a = 3;
      int b = 4;
      int c = a + b;
      int d = getNewValue(c);
      return;
   } // End method main

   public static int getNewValue(int var) {
      return var * var;
   } // End method getNewValue
}

Java Bytecode

Method AdditionMethodTest()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return

The aload_<n> loads the object stored in the local variable array at index n. The objectref is pushed on to the stack. In this case, the “this” object is pushed on to the stack.
Addition Method Test Example: From Java Source To Java Bytecode

class AdditionMethodTest {
    public static void main(String args[]) {
        int a = 3;
        int b = 4;
        int c = a + b;
        int d = getNewValue(c);
        return;
    } // End method main

    public static int getNewValue(int var) {
        return var * var;
    } // End method getNewValue
}

Method AdditionMethodTest()
0 aload_0
1 invokespecial #1 <Method java.lang.Object()>
4 return

The invokespecial bytecode calls the constructor of the “this” class’s superclass. In this case, the java.lang.Object is the superclass. No other bytecodes are present because
Addition Method Test Example: From Java Source To Java Bytecode

Java Source Code

```java
class AdditionMethodTest {
    public static void main(String args[]) {
        int a = 3;
        int b = 4;
        int c = a + b;
        int d = getNewValue(c);
        return;
    } // End method main

    public static int getNewValue(int var) {
        return var * var;
    } // End method getNewValue
}
```

Java Bytecode

```
Method AdditionMethodTest()  
0 aload_0
1 invokespecial #1 <Method java.lang.Object()>
4 return
```

The return bytecode simple returns void
Addition Method Test Example: From Java Source To Java Bytecode

Java Source Code

class AdditionMethodTest {
    public static void main(String args[]) {
        int a = 3;
        int b = 4;
        int c = a + b;
        int d = getNewValue(c);
        return;
    } // End method main

    public static int getNewValue(int var) {
        return var * var;
    } // End method getNewValue
}

Java Bytecode

Method void main(java.lang.String[])
0 iconst_3
1 istore_1
2 iconst_4
3 istore_2
4 iload_1
5 iload_2
6 iadd
7 istore_3
8 iload_3
9 invokestatic #2 <Method int getNewValue(int)>
12 istore 4
14 return

The iconst_<n> bytecode loads the int constant onto the operand stack.
Addition Method Test Example: From Java Source To Java Bytecode

class AdditionMethodTest {
    public static void main(String args[]) {
        int a = 3;
        int b = 4;
        int c = a + b;
        int d = getNewValue(c);
        return;
    } // End method main

    public static int getNewValue(int var) {
        return var * var;
    } // End method getNewValue
}

Method void main(java.lang.String[])
0 iconst_3
1 istore_1
2 iconst_4
3 istore_2
4 iload_1
5 iload_2
6 iadd
7 istore_3
8 iload_3
9 invokestatic #2 <Method int getNewValue(int)>
12 istore 4
14 return

The istore command stores the value on the top of the operand stack to the location in the local variable array location indicated by index.
Addition Method Test Example: From Java Source To Java Bytecode

```java
class AdditionMethodTest {
    public static void main(String args[]) {
        int a = 3;
        int b = 4;
        int c = a + b;
        int d = getNewValue(c);
        return;
    } // End method main

    public static int getNewValue(int var) {
        return var * var;
    } // End method getNewValue
}
```

Java Source Code

Java Bytecode

```java
Method void main(java.lang.String[])
0 iconst_3
1 istore_1
2 iconst_4
3 istore_2
4 iload_1
5 iload_2
6 iadd
7 istore_3
8 iload_3
9 invokestatic #2 <Method int getNewValue(int)>
12 istore 4
14 return
```

The iload command loads the int value stored in the location indicated by index in the local variable array.
Addition Method Test Example: From Java Source To Java Bytecode

Java Source Code

class AdditionMethodTest {
    public static void main(String args[]) {
        int a = 3;
        int b = 4;
        int c = a + b;
        int d = getNewValue(c);
        return;
    } // End method main

    public static int getNewValue(int var) {
        return var * var;
    } // End method getNewValue
}

Java Bytecode

Method void main(java.lang.String[])
   0 iconst_3
   1 istore_1
   2 iconst_4
   3 istore_2
   4 iload_1
   5 iload_2
   6 iadd
   7 istore_3
   8 iload_3
   9 invokestatic #2 <Method int getNewValue(int)>
  12 istore 4
  14 return

The iadd bytecode pops the top 2 int values off of the stack, performs the additions, and puts the result back on to the stack.
Addition Method Test Example: From Java Source To Java Bytecode

Java Source Code

class AdditionMethodTest {
    public static void main(String args[]) {
        int a = 3;
        int b = 4;
        int c = a + b;
        int d = getNewValue(c);
        return;
    } // End method main

    public static int getNewValue(int var) {
        return var * var;
    } // End method getNewValue
}

Java Bytecode

Method void main(java.lang.String[])
    0 iconst_3
    1 istore_1
    2 iconst_4
    3 istore_2
    4 iload_1
    5 iload_2
    6 iadd
    7 istore_3
    8 iload_3
    9 invokestatic #2 <Method int getNewValue(int)>
   12 istore 4
   14 return

The invokestatic bytecode invokes the static method getNewValue(). The method takes a single parameter which is pushed on the operand stack in the bytecode directly before.
Addition Method Test Example: From Java Source To Java Bytecode

class AdditionMethodTest {  
    public static void main(String args[]) {  
        int a = 3;  
        int b = 4;  
        int c = a + b;  
        int d = getNewValue(c);  
        return;  
    } // End method main

    public static int getNewValue(int var) {  
        return var * var;  
    } // End method getNewValue
}  

Method int getNewValue(int)  
0 iload_0  
1 iload_0  
2 imul  
3 ireturn

The imul bytecode is similar to the iadd bytecode except the product of the top 2 int values is pushed on to the operand stack.
Addition Method Test Example: From Java Source To Java Bytecode

Java Source Code

```java
class AdditionMethodTest {  
    public static void main(String args[]) {    
        int a = 3;    
        int b = 4;    
        int c = a + b;    
        int d = getNewValue(c);    
        return;  
    } // End method main

    public static int getNewValue(int var) {    
        return var * var;    
    } // End method getNewValue
}
```

Java Bytecode

```
Method int getNewValue(int)  
0 iload_0  
1 iload_0  
2 imul  
3 ireturn
```

The ireturn bytecode returns the int value that is stored on the top of the operand stack.
Conversion From Java Bytecode to HIR:

- Conversion from bytecode to HIR is performed by the compiler front-end.
- The front-end contains 2 parts:
  - The BC2IR algorithm that translates bytecodes to HIR and performs on-the-fly optimizations during translation.
  - Additional optimizations perform on the HIR after translation.
The BC2IR translation

- Discovers extended-basic-blocks
- Constructs an exception-table for the method
- Creates HIR instructions for bytecodes
- Performs On-the-fly optimizations
  - Copy propagation
  - Constant propagation
  - Register renaming for local variables
  - Dead-Code elimination
  - Short final or static methods are inlined
- Even though these optimizations are performed in later phases, doing so here reduces the size of the HIR generated and thus compile time.
Example of On-the-Fly Analyses and Optimizations

- Consider Copy Propagation as an example:

<table>
<thead>
<tr>
<th>Java Bytecode</th>
<th>Generated IR (optimization off)</th>
<th>Generated IR (optimization on)</th>
</tr>
</thead>
<tbody>
<tr>
<td>iload x</td>
<td>INT_ADD tint, xint 5</td>
<td>INT_ADD yint, xint, 5</td>
</tr>
<tr>
<td>iconst 5</td>
<td>INT_MOVE yint, tint</td>
<td></td>
</tr>
<tr>
<td>iadd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>istore y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
AdditionTest Example: From Bytecode to HIR

Java Bytecode

Method AdditionMethodTest()
0 aload_0
1 invokespecial #1 <Method java.lang.Object()>
4 return

Method void main(java.lang.String[])
0 iconst_3
1 istore_1
2 iconst_4
3 istore_2
4 iload_1
5 iload_2
6 iadd
7 istore_3
8 iload_3
9 invokestatic #2 <Method int getNewValue(int)>
12 istore 4
14 return

Method int getNewValue(int)
0 iload_0
1 iload_0
2 imul
3 ireturn

NOTE: This example was run with RVM command line option -X:opt:inline=false to prevent inlining
AdditionTest Example: From Bytecode to HIR

********** START OF IR DUMP Initial HIR FOR AdditionMethodTest.main ([Ljava/lang/String;)V
-13  LABEL0 Frequency: 0.0
-2  EG  ir_prologue  l0i([Ljava/lang/String;,d) =
   1     int_move  l1i(B) = 3
   3     int_move  l2i(B) = 4
   7     int_move  l3i(B) = 7
9    EG  call  l5i(I) AF CF OF PF SF ZF = 66668, static"AdditionMethodTest.getNewValue (I)I", <unused>, 7
-3    return <unused>
-1    bbend  BB0 (ENTRY)
********** END OF IR DUMP Initial HIR FOR AdditionMethodTest.main ([Ljava/lang/String;)V

********** START OF IR DUMP Initial HIR FOR AdditionMethodTest.getNewValue (I)I
-13  LABEL0 Frequency: 0.0
-2  EG  ir_prologue  l0i(I,d) =
   2     int_mul  t2i(I) = l0i(I,d), l0i(I,d)
   3     int_move  t1i(I) = t2i(I)
-3    return  t1i(I)
-1    bbend  BB0 (ENTRY)
********** END OF IR DUMP Initial HIR FOR AdditionMethodTest.getNewValue (I)I
Breakdown of the HIR

Java Bytecode

Method void main(java.lang.String[])
0 iconst_3
1 istore_1
2 iconst_4
3 istore_2
4 iload_1
5 iload_2
6 iadd
7 istore_3
8 iload_3
9 invokestatic #2 <Method int getValue(int)>
12 istore 4
14 return

HIR

-13 LABEL0 Frequency: 0.0

Each label is the beginning of a new Extended Basic Block

Is the corresponding line number in the source code, similar to what javap -c would print out. Some values have a negative number because they were inserted at a lower level as thus do not have an equivalent in the bytecode.

The Frequency is used by Jikes to predict control flow.
Java Bytecode

Method void main(java.lang.String[])
0 iconst_3
1 istore_1
2 iconst_4
3 istore_2
4 iload_1
5 iload_2
6 iadd
7 istore_3
8 iload_3
9 invokestatic #2 <Method int getNewValue(int)>
12 istore 4
14 return

HI R

-2 EG ir_prologue
10i([Ljava/lang/String;,d) =

Operator that have been defined in OperatorList.dat
ir_prologue is a pseudo instruction to represent the prologue

The E refers to that this line can produce an Exception
The G denotes that this line may yield to the Garbage Collector

March 20, 2003 Shane A. Brewer
March 20, 2003

Shane A. Brewer

Breakdown of the HIR

Java Bytecode

Method void main(java.lang.String[])
0  iconst_3
1  istore_1
2  iconst_4
3  istore_2
4  iload_1
5  iload_2
6  iadd
7  istore_3
8  iload_3
9  invokestatic #2 <Method int getValue(int)>
12  istore 4
14  return

Corresponds to line 1 in the main method source code

HIR

1     int_move                l1i(B) = 3

Operator to move an integer into a register
Breakdown of the HIR

Java Bytecode

Method void main(java.lang.String[])
0 iconst_3
1 istore_1
2 iconst_4
3 istore_2
4 iload_1
5 iload_2
6 iadd
7 istore_3
8 iload_3
9 invokestatic #2 <Method int newNameValue(int)>  
10 istore 4
11 return

HIR

1 int_move
2 l1i(B) = 3

The prefix indicates the locality of the variable while the number indicates the register name:

- l = local
- t = temporary

The suffix indicates the type of the register:

- i = Integer
- c = Condition
- d = Double
- f = Float
- l = Long
- v = Validation

The parameter indicates the type of the variable (taken from the JVM Specification):

- B = Byte
- C = Char
- D = Double
- F = Float
- I = Int
- J = Long
- L<classname> = Reference
- S = Short
- Z = Boolean

[ = Reference (One Array Dimension)
Breakdown of the HIR

Java Bytecode

Method void main(java.lang.String[])
0 iconst_3
1 istore_1
2 iconst_4
3 istore_2
4 iload_1
5 iload_2
6 iadd
7 istore_3
8 iload_3
9 invokestatic #2 <Method int getValue(int)>
12 istore 4
14 return

HIR

3 int_move
l2i(B) = 4

Value we are assigning to the register

Same idea as the last operator
Breakdown of the HIR

Java Bytecode

Method void main(java.lang.String[])
0 iconst_3   
1 istore_1   
2 iconst_4   
3 istore_2   
4 iload_1    
5 iload_2    
6 iadd       
7 istore_3   
8 iload_3    
9 invokestatic #2 <Method int getNewValue(int)>  
12 istore 4  
14 return

HIR

7 int_move    
13i(B) = 7

Same idea as the last operator
Breakdown of the HIR

Java Bytecode
Method void main(java.lang.String[])
0 iconst_3
1 istore_1
2 iconst_4
3 istore_2
4 iload_1
5 iload_2
6 iadd
7 istore_3
8 iload_3
9 invokestatic #2 <Method int getNewValue(int)>
12 istore 4
14 return

A call instruction
Indicates that the call can set the following EFLAGS on the IA32 architecture
A = Alignment
C = Carry
O = Overflow
P = Parity
S = Sign
Z = Zero

March 20, 2003
Shane A. Brewer

HIR
9 EG call
15i(I) AF CF OF PF SF ZF = 66668,
static"AdditionMethodTest.getNewValue (I)I", <unused>, 7

Indicates that we have one parameter to the method, which is an integer (Again taken from the JVM Specification)
The name of the method we are calling. In this case it is to a static method called “AdditionMethodTest.getNewValue()”
The offset into the JTOC which contains the address of the method
Java Bytecode

Method void main(java.lang.String[])

0 iconst_3
1 istore_1
2 iconst_4
3 istore_2
4 iload_1
5 iload_2
6 iadd
7 istore_3
8 iload_3
9 invokestatic #2 <Method int getNewValue(int)>
12 istore 4
14 return

HIR

9  EG  call  15i(1) AF CF OF PF SF ZF = 66668,
static"AdditionMethodTest.getNewValue (I)I", <unused>, 7

Indicates the return type of the method.
In this case, the method returns an int
(Taken from the JVM Specification)

The parameter that we are passing to the method. In this case, the int 7.

Used if we are passing a guard to a method. In this case, no guard is passed.
Breakdown of the HIR

Java Bytecode

Method void main(java.lang.String[])

0 iconst_3
1 istore_1
2 iconst_4
3 istore_2
4 iload_1
5 iload_2
6 iadd
7 istore_3
8 iload_3
9 invokestatic #2 <Method int getNewValue(int)>
12 istore 4
14 return

HIR

-3 return <unused>

Return instruction. No value is returned in this case.
Breakdown of the HIR

Java Source Code

public static int getNewValue(int var) {
    return var * var;
} // End method getNewValue

Java Bytecode

Method int getNewValue(int)
  0 iload_0
  1 iload_0
  2 imul
  3 ireturn

HIR

-13  LABEL0  Frequency:  0.0
-2   EG   ir_prologue           l0i(l,d) =
  2   int_mul              t2i(l) = l0i(l,d), l0i(l,d)
  3   int_move            t1i(l) = t2i(l)
-3   return              t1i(l)
-1   bbend                BB0 (ENTRY)

Same as described before except for one attribute. In the use of l0i(l,d), the additional attribute describes the variable as one of the following:

x - Is Extant
d - Is Declared Type
p - Is Precise Type
+ - Is Positive Int
### Breakdown of the HIR

**Java Source Code**

```java
public static int getNewValue(int var) {
    return var * var;
} // End method getNewValue
```

**Java Bytecode**

```
Method int getNewValue(int)
0 iload_0
1 iload_0
2 imul
3 ireturn
```

Multiplies the value stored in l0i (the first parameter passed to the method) to itself and stores the result in a temporary register t2i. The int_move instruction moves the calculated value from t2i to t1i.

**HIR**

```
-13 LABEL0 Frequency: 0.0
-2 EG ir_prologue
   l0i(I,d) =
2     int_mul
   t2i(I) = l0i(I,d), l0i(I,d)
3     int_move
   t1i(I) = t2i(I)
-3     return
   t1i(I)
-1     bbend
   BB0 (ENTRY)
```
Breakdown of the HIR

Java Source Code

```java
public static int getNewValue(int var) {
    return var * var;
} // End method getNewValue
```

Java Bytecode

```
Method int getNewValue(int)
    0 iload_0
    1 iload_0
    2 imul
    3 ireturn

Returns the value stored in temporary register t1i.
bbend indicates the end of Extended Basic Block BB0
```

HIR

-13 LABEL0 Frequency: 0.0
-2 EG ir_prologue l0i(1,d) =
 2 int_mul t2i(I) = l0i(1,d), l0i(1,d)
 3 int_move t1i(I) = t2i(I)
-3 return t1i(I)
-1 bbend BB0 (ENTRY)
Optimization of HIR

- Simple optimization algorithms with modest compile-time overheads are performed.

- Generally fall into 3 classes:
  - Local Optimizations
    - Common Sub-expression elimination
    - Removal of redundant exception checks
    - Redundant load elimination
  - Flow-insensitive Optimizations
    - Copy Propagation
    - Dead code elimination
    - Conservative Escape Analysis
  - In-line Expansion of Method Calls
    - Guarded Receiver type prediction
Conversion From HIR to LIR

- The LIR expands the HIR instructions into operations that are specific to the Jikes RVM
  - Object Layout
  - Parameter-passing mechanisms
- Instructions like a single HIR “invokevirtual” instruction are expanded to 3 LIR instructions that:
  - Obtain the TIB pointer from an object
  - Obtain the address of the appropriate method body from the TIB
  - Transfer control to the method body
- A Dependence Graph is constructed for each extended basic block and includes the representation of:
  - True/Anti/Output Dependences for both Registers and Memory
  - Control/Synchronization/Exception Dependences
- LIR can be 2 to 3 times larger than corresponding HIR
AdditionTest Example: From HIR to LIR

HIR

*********** START OF IR DUMP Initial HIR FOR AdditionMethodTest.main ([Ljava/lang/String;)V
-13  LABEL0 Frequency: 0.0
-2  EG ir_prologue  l0i([Ljava/lang/String;,d) =
  1  int_move  l1i(B) = 3
  3  int_move  l2i(B) = 4
  7  int_move  l3i(B) = 7
  9  EG call  l5i(I) AF CF OF PF SF ZF = 66668, static"AdditionMethodTest.getNewValue (I)I", <unused>, 7
-3  return  <unused>
-1  bbend  BB0 (ENTRY)
*********** END OF IR DUMP Initial HIR FOR AdditionMethodTest.main ([Ljava/lang/String;)V

LIR

*********** START OF IR DUMP Initial LIR FOR AdditionMethodTest.main ([Ljava/lang/String;)V
-13  LABEL0 Frequency: 1.0
-2  EG ir_prologue  l0si([Ljava/lang/String;,d) =
  0  G yieldpoint_prologue
  9  ref_load  t6i([B) = 1124073932, 66668, <unused>, <unused>
  9  EG call  l5si(I) AF CF OF PF SF ZF = t6i([B), static"AdditionMethodTest.getNewValue (I)I", <unused>, 7
-3  return  <unused>
-1  bbend  BB0 (ENTRY)
*********** END OF IR DUMP Initial LIR FOR AdditionMethodTest.main ([Ljava/lang/String;)V

March 20, 2003 Shane A. Brewer 46
Breakdown of the LIR

**HIR**

-13  LABEL0  Frequency:  0.0

-2   EG  ir_prologue   l0i([Ljava/lang/String;,d) =

**LIR**

-13  LABEL0  Frequency:  1.0

-2   EG  ir_prologue   l0si([Ljava/lang/String;,d) =

l0i has changed to losi to indicate that the register is being used by SSA.

Frequency has changed from 0.0 to 1.0, indicating that this Extended Basic Block has a high probability of being executed.
Breakdown of the LIR

**HIR**

1. int_move  \( l1i(B) = 3 \)
2. int_move  \( l2i(B) = 4 \)
3. int_move  \( l3i(B) = 7 \)
9. EG call  \( l5i(I) \) AF CF OF PF SF ZF = 66668, static"AdditionMethodTest.getNewValue (I)I", <unused>, 7

**LIR**

9. ref_load  \( t6i([B) = 1124073932, 66668, <unused>, <unused> \)
9. EG call  \( l5si(I) \) AF CF OF PF SF ZF = \( t6i([B), static"AdditionMethodTest.getNewValue (I)I", <unused>, 7 \)

All 3 int_move instructions have been replaced by a ref_load instruction, which obtains an array of bytes that refer to the address of the method.

The call instruction now takes the temporary register \( t6i \) instead of the offset into the JTOC.

1124073932 refers to the address of the JTOC

66668 is the offset into the JTOC that contains the address of the method
Breakdown of the LIR

HIR

-3  return  <unused>
-1  bbend     BB0 (ENTRY)

LIR

-3  return  <unused>
-1  bbend     BB0 (ENTRY)

No Change
AdditionTest Example: From HIR to LIR

HIR

********** START OF IR DUMP Initial HIR FOR AdditionMethodTest.getNewValue (I)I
-13     LABEL0   Frequency:  0.0
-2      EG  ir_prologue             l0i(I,d) =
 2      int_mul                 t2i(I) = l0i(I,d), l0i(I,d)
 3      int_move               t1i(I) = t2i(I)
-3      return                 t1i(I)
-1      bbend                   BB0 (ENTRY)
********** END OF IR DUMP Initial HIR FOR AdditionMethodTest.getNewValue (I)I

LIR

********** START OF IR DUMP Initial LIR FOR AdditionMethodTest.getNewValue (I)I
-13     LABEL0   Frequency:  1.0
-2      EG  ir_prologue             l0si(I,d) =
 0      G  yieldpoint_prologue
 2      int_mul                 t2si(I) = l0si(I,d), l0si(I,d)
-3      return                 t2si(I)
-1      bbend                   BB0 (ENTRY)
********** END OF IR DUMP Initial LIR FOR AdditionMethodTest.getNewValue (I)I

March 20, 2003 Shane A. Brewer
Breakdown of the LIR

HIR

-13  LABEL0  Frequency:  0.0
-2   EG  ir_prologue  \text{l0i}(l,d) =

LIR

-13  LABEL0  Frequency:  1.0
-2   EG  ir_prologue  \text{l0si}(l,d) =

Same as described before
Breakdown of the LIR

Here we see that we have optimized out the int_move instruction and just return the value in temporary register t2si.
Optimizations of LIR

- Local common Subexpression elimination is the only optimization performed on LIR.
- In principle, any optimization performed on HIR could also be performed on LIR, but the size of LIR makes it not as attractive as HIR.
Conversion From LIR to MIR

- Dependence Graphs for the extended basic blocks of a method partitioned into trees.
- These trees are fed into the Bottom-Up Rewriting System (BURS) which produces the MIR
- Symbolic registers are mapped to physical registers
- A **Prolog** is added at the beginning and an **Epilog** is added at the end of each method [2]
Method Prolog and Epilog [2]

- The method Prolog:
  - Saves any nonvolatile registers needed by the method
  - Checks to see if a yield has been requested
  - Locks the indicated object if the method is synchronized
  - Acts as a yield point

- The method Epilog:
  - Restores any saved registers
  - Deallocate the stack frame.
  - Unlocks the indicated object if the method is synchronized
AdditionTest Example: From LIR to MIR

LIR

******* START OF IR DUMP Initial LIR FOR AdditionMethodTest.main ([Ljava/lang/String;)V
-13    LABEL0  Frequency:  1.0
-2     EG  ir_prologue  l0si([Ljava/lang/String;,) =
0      G  yieldpoint_prologue
9      ref_load    t6i([B) = 1124073932, 66668, <unused>, <unused>
9      EG  call     l5si(I) AF CF OF PF SF ZF = t6i([B), static"AdditionMethodTest.getNewValue (I)"", <unused>, 7
-3     return     <unused>
-1     bbend     BB0 (ENTRY)
******* END OF IR DUMP Initial LIR FOR AdditionMethodTest.main ([Ljava/lang/String;)V

MIR

******* START OF IR DUMP Initial MIR FOR AdditionMethodTest.main ([Ljava/lang/String;)V
-13    LABEL0  Frequency:  1.0
-2     EG  ir_prologue  l0si([Ljava/lang/String;,) =
0      G  yieldpoint_prologue
9      ia32_call     l5si(I) AF CF OF PF SF ZF = <0+1124140600>DW, static"AdditionMethodTest.getNewValue (I)"", 7
-3     ia32_ret     <unused>, <unused>, <unused>
-1     bbend     BB0 (ENTRY)
******* END OF IR DUMP Initial MIR FOR AdditionMethodTest.main ([Ljava/lang/String;)V

March 20, 2003 Shane A. Brewer
Breakdown of the MIR

LI R

-13  LABEL0  Frequency:  1.0
-2   EG  ir_prologue  l0si([Ljava/lang/String;,d) =
  0   G  yieldpoint_prologue

MIR

-13  LABEL0  Frequency:  1.0
-2   EG  ir_prologue  l0si([Ljava/lang/String;,d) =
  0   G  yieldpoint_prologue

No change
Breakdown of the MIR

The IR instructions “call” and “return” have been replaced by the IA32 specific “ia32_call” and “ia32_ret” instructions.

The ref_load instruction has been replaced with a literal address (Not sure what DW means or why it is 0+).
AdditionTest Example: From LIR to MIR

LIR

********** START OF IR DUMP Initial LIR FOR AdditionMethodTest.getNewValue (I)I
-13 LABEL0 Frequency: 1.0
-2 EG ir_prologue l0si(l,d) =
0 G yieldpoint_prologue
2 int_mul t2si(l) = l0si(l,d), l0si(l,d)
-3 return t2si(l)
-1 bbend BB0 (ENTRY)
********** END OF IR DUMP Initial LIR FOR AdditionMethodTest.getNewValue (I)I

MIR

********** START OF IR DUMP Initial MIR FOR AdditionMethodTest.getNewValue (I)I
-13 LABEL0 Frequency: 1.0
-2 EG ir_prologue l0i(l,d) =
0 G yieldpoint_prologue
2 ia32_imul2 l0i(l,d) AF CF OF PF SF ZF <-- l0i(l,d)
-3 ia32_ret <unused>, l0i(1), <unused>
-1 bbend BB0 (ENTRY)
********** END OF IR DUMP Initial MIR FOR AdditionMethodTest.getNewValue (I)I

March 20, 2003 Shane A. Brewer
Breakdown of the MIR

LI R

-13 LABEL0 Frequency: 1.0
-2 EG ir_prologue \ l0si(l,d) =
0 G yieldpoint_prologue

MIR

-13 LABEL0 Frequency: 1.0
-2 EG ir_prologue \ l0i(l,d) =
0 G yieldpoint_prologue

No changes
Breakdown of the MIR

LI R

2    int_mul                 t2si(I) = l0si(I,d), l0si(I,d)
-3    return                  t2si(I)
-1    bbend                   BB0 (ENTRY)

MI R

-2    ia32_imul2              l0i(I,d) AF CF OF PF SF ZF <-- l0i(I,d)
-3    ia32_ret                <unused>, l0i(I), <unused>
-1    bbend                   BB0 (ENTRY)

The int_mul instruction is replaced with an ia32_imul2 instruction as the compiler knows that it is multiplying the same values together.

The temporary register t2si is also eliminated.
MIR Optimizations

- Live variable analysis is performed
- A Linear-scan global register-allocation algorithm assigns physical machine registers to symbolic MIR registers
  - Based on a greedy algorithm
Conversion From MIR to Machine Code

- The binary executable code is emitted into an array of ints that is the method body
- Finalizes the exception table
- Converts intermediate-instruction offsets into machine-code offsets.
AdditionTest Example: From MIR to IA32 Machine Code

-2 ia32_imul2    l0i(I,d) AF CF OF PF SF ZF <-- l0i(I,d)     MIR
-3 ia32_ret     <unused>, l0i(I), <unused>
-1 bbend        BB0 (ENTRY)

Machine Code

Numbers on the right indicate IA32 opcodes
Numbers of the left indicate offsets in the method???
AdditionTest Example: From MIR to IA32 Machine Code

********** START OF IR DUMP Initial MIR FOR AdditionMethodTest.getNewValue (I)I
-13       LABEL0 Frequency: 1.0
-2        EG ir_prologue    l0i(I,d) =
0         G yieldpoint_prologue
2        ia32_imul2    l0i(I,d) AF CF OF PF SF ZF <-- l0i(I,d)
-3        ia32_ret     <unused>, l0i(I), <unused>
-1        bbend        BB0 (ENTRY)
********** END OF IR DUMP Initial MIR FOR AdditionMethodTest.getNewValue (I)I

Machine Code

000000|    IMUL    eax    eax   | 0FAFC0
000003|    RET     4       | C20400
********** END OF: Final machine code FOR AdditionMethodTest.getNewValue (I)I
References


J. Choi et al., “Efficient and Precise Modeling of Exceptions for the Analysis of Java Programs”, 1999
