Dynamic Compilation
Execution Options for Bytecode

• Transfer bytecode; **interpret** bytecode at the target
  ■ Line by line execution, with some optimizations to reduce the overhead of interpretation
    ▸ Indirect threading, direct threading, replication, superinstructions

• Transfer bytecode; **compile** bytecode at the target
  ■ Translate multiple bytecode instructions to native code
    ▸ Method-level, path-level (trace compilation)
    ▸ Just-in-time compilation: wait to compile upon first invocation
      ◆ Only compile what you will execute
    ▸ Dynamic compilation: JIT + recompile at any time
      ◆ Improve performance by waiting to or re-compiling when you know more about the behavior of the program
Execution Options for Bytecode

- Many runtime systems use a combination
  - **Interpretation**
    - Good if you only execute a path once
  - **Dynamic compilation**
    - Good if you can amortize the cost of compilation (time/iteration)
    - Complicates runtime, increases footprint
    - May have multiple compilers (e.g. 1 fast, 1 optimizing)
  - **AOT compilation (system libs)**
    - Good for some things, but not for all (some runtime-based feedback-directed optimization can improve performance significantly)
    - Increases footprint (native code is significantly bigger than bytecode)
- Goal: achieve the best performance (+battery life)
Jalapeno = JikesRVM – A Dynamic and Adaptive Optimizing Compiler for Java

Let’s consider just the optimizing compiler first...
JikesRVM Opt Compiler’s Intermediate Forms

- 3 different forms used for each method compilation
  - High-level intermediate representation (HIR)
  - Low-level intermediate representation (LIR)
  - Machine-level intermediate representations (MIR)

- N-tuples (1 typed operator & n-1 typed operands)
  - A generalization of 3 address code and quadruples

- Most operands represent symbolic (virtual) registers
  - Can also represent physical registers, memory locations, constants, branch targets and types.
  - Distinct operators for similar operations on different primitive types
  - Operands carry type information
JikesRVM Opt Compiler’s Intermediate Forms

- Instructions are grouped by **extended basic blocks**
  - Non-extended basic blocks have 1-entry & 1-exit
  - Extended: single entry, multiple exit
  - Exception throws and **method calls** do **not** end a basic block
    - Therefore, control may exit out the middle of a block
  - Better for optimizations (more instructions to work with)
  - Only a single entry however (except for returns)

- Cached information for each IR
  - Auxiliary information (optional) used for optimization
JikesRVM Opt Compiler’s Intermediate Forms

- Instructions are grouped by **extended basic blocks**
  - Non-extended basic blocks have 1-entry & 1-exit

**non-extended basic block leaders**

```
java.lang.Double OutputByBT();
Code:
  0: dconst_0
  1: invokevirtual #9              // Method
     java/lang/Double.valueOf:(D)Ljava/lang/Double;
  4: astore_1
  5:  iconst_0
  6:  istore_2
  7:  iload_2
  8:  bipush   32
  10: if_icmpge 104
13:  iconst_0
14:  istore_3
```

**Leaders:**
First instruction in a method
Instruction that follows a branch or jump
Instruction that is a **target** of a branch or jump
Instruction that follows a call/invoke

**extended basic block leaders**

```
java.lang.Double OutputByBT();
Code:
  0: dconst_0
  1: invokevirtual #9              // Method
     java/lang/Double.valueOf:(D)Ljava/lang/Double;
  4: astore_1
  5:  iconst_0
  6:  istore_2
  7:  iload_2
  8:  bipush   32
  10: if_icmpge 104
13:  iconst_0
14:  istore_3
```

**Leaders:**
First instruction in a method
Instruction that follows a branch or jump
Instruction that is a **target** of a branch or jump
HIR Generation (Abstract Interpretation)

- Implementation of translation from bytecode to HIR
  - Find extended basic block structure
  - Construct exception table for the method
  - Abstract interpretation (local var types + operand stack)

```java
worklist.add(entry_block, exception_handling_blocks[]);
while ((ele = worklist.remove()) != null) {
    /* stack is the operand stack for the method */
    stack = ele.getParentsStack();  /* values may differ but types must be the same. Multiple values make up a set. */
    insts = ele.getInstructions();
    interpret(insts, stack);  /* create a symbolic stack and walk the code updating the stack */
    if (changed())
        worklist.add(ele.getDests());  /* put all possible destination blocks on the worklist */
}
```
HIR Generation (Abstract Interpretation)

```
interpret(insts,stack)
VPC: virtual PC
process BB0
vpc = 0

0:ild R1,lv[0] vpc = 1
1:ist lv[1],R1 vpc = 2
2:ild R2,lv[0] vpc = 3
3:ilc R3,10 vpc = 5
5:ile 17,R3,R2 vpc = 8,17

store stack
add BB2,BB1 to worklist
```

**BB0**

0 iload_0  
1 istore_1  
2 iload_0  
3 bipush 10  
5 if_icmple 17

**BB1**

8 iload_1  
9 iload_0  
10 iconst_4  
11 imul  
12 iadd  
13 istore_1  
14 goto 23

**BB2**

17 iload_1  
18 iload_0  
19 iconst_4  
20 iadd  
21 iadd  
22 istore_1

**BB3**

23 iload_1  
24 ireturn

```
0: lv0 (int)  
1:          
2: lv0 (int)  
3: lv0 (int)  
5: lv0 (int)  
8,17  

Inst stack VPC after
0: lv0 (int)  
1:          
2: lv0 (int)  
3: 10 (int)  
5:          
```
HIR Optimizations

- Goals
  - Reduce the size of the intermediate code
  - Remove redundancies

- Copy & constant propagation
- Dead code elimination
- Inline short methods that are static or final
  - Application methods as well as JVM methods!
- Redundant check & load elimination
- Common subexpression elimination
Basic Blocks and Control Flow

- Optimization and code generation can be performed on a small (easy) or large (hard) piece of the control flow graph

  - Local - within a basic block
  - Global - across basic blocks within one method (intraprocedural)
  - Inter-procedural - across methods, within one program

- Terminology used for reads/writes of variables/registers
  - Defines (def) - when a register is written to
    
    \[
    \text{Def } r2: \quad r2 = \ldots \quad \text{//write}
    \]
  - Use - when a register is read
    
    \[
    \text{Use of } r2: \quad \ldots = \ldots r2 \ldots \quad \text{//read}
    \]
Copy & Constant Propagation

- **Copy propagation**
  - If a variable value is assigned into a second variable (register) and that second variable is used in subsequent instructions, use the first variable and eliminate the copy.

- **Constant propagation**
  - Same as above only for constant values.

```plaintext
r1 = a
r2 = r1
r3 = 10
if (r3 <= r2) goto 17

//r2 & r3 are not used again
```

```plaintext
r1 = a
if (10 <= r1) goto 17
//copy prop, remove r2
//constant prop, remove r3
//propagate 10
```
Local Variable Register Renaming & DCE

- Dead code elimination (DCE)
  - Remove instructions that have no affect

```plaintext
r1 = r2 + 1
r2 = r1
r3 = r2 * 2
// no more
// r2 uses
// in method
```

Copy propagation

```plaintext
r1 = r2 + 1
r1 = r1
r3 = r1 * 2
// no more
// r2 uses
// in method
```

DCE

```plaintext
r1 = r2 + 1
r3 = r1 * 2
```

Def (define) of r2:  
Use of r2:
```plaintext
... = ... r2 ...
```
Inlining

- CFG merge

prologue: ...

BB1: p = 0
     i = 1

BB2: p = p + i
     param p; foo(1);

BB3: p = 0
     i = 5

BB4: t1 = i * 2
     i = t1 + 1
     if i <= 20 goto BB2

BB5: k = p*3

BB1: p = 0
     i = 1

BB2: p = p + i
     param p; foo(1);

BB3: p = 0
     i = 5

BB4: t1 = i * 2
     i = t1 + 1
     if i <= 20 goto BB2

BB5: k = p*3

prologue/epilogue: code inserted by compiler to setup runtime stack frame for function call (map arguments to parameters) & return, saving off & restoring registers, send back return value, + setup for GC/threading/locks

**basic blocks here not extended basic blocks (calls end a bb for simplicity)

static foo(int j)

prologue: ...

BB1: ...

BBn: ...

epilogue: ...

return
Inlining

- CFG merge

```c
inline
p = 0
i = 1
BB1:
p = p + i
param p;
BB2:
p = 0
i = 5
BB3:
p = 0
i = 5
BB4:
t1 = i * 2
i = t1 + 1
if i <= 20 goto BB2
BB5:
k = p*3
```

**basic blocks here not extended basic blocks (calls end a bb for simplicity)**

Was: static foo(int j)
param (j=1) stored appropriately

prologue:  
BB1:  
P = 0
i = 1
BB2:  
P = p + i
param p;
BB3:  
P = 0
i = 5
BB4:  
t1 = i * 2
i = t1 + 1
if i <= 20 goto BB2
BB5:  
k = p*3
BBn:  
...
Redundant Check and Load Elimination

- Redundant load elimination
  - Store values in registers to avoid unnecessary loads
  - A load of a local variable is a memory load

```
0 iload_0            r1 = lv[0]          r1 = lv[0]
1 istore_1           lv[1] = r1          lv[1] = r1
2 iload_0            r2 = lv[0]          r2 = r1
3 bipush 10          r3 = 10              r3 = 10
5 if_icmple 17       if r2<=r3 goto 17    if r2<=r3 goto 17
```

**copy propagation + DCE**

- **assumes r2 is unused below**
Redundant Check Elimination

- Redundant check elimination
  - If it can be proven (statically) that one check is sufficient to ensure that subsequent instructions will not violate language rules, subsequent checks can be removed
    - Array bounds checks
      
      \[
      \begin{aligned}
      \text{check}(0\leq i<10) \\
      a[i] = r1 \\
      i = i + 1 \\
      \text{check}(0\leq i<10) \\
      a[i] = r1
      \end{aligned}
      \]

      \[
      \begin{aligned}
      \text{check}(0\leq i<9) \\
      a[i] = r1 \\
      i = i + 1 \\
      a[i] = r1
      \end{aligned}
      \]

    - Null checks (here foo has type C and is an object reference variable)
      
      \[
      \begin{aligned}
      \text{if} \ foo==\text{null} \ \text{goto} \ 17 \\
      \text{foo.fld} = r1 \ //\text{putfield} \ C.\text{fld.I} \\
      \text{if} \ foo==\text{null} \ \text{goto} \ 17 \\
      \text{foo.meth()} \ //\text{invokevirtual} \ C.\text{meth}()V
      \end{aligned}
      \]

      \[
      \begin{aligned}
      \text{if} \ foo==\text{null} \ \text{goto} \ 17 \\
      \text{foo.fld} = r1 \ //\text{putfield} \\
      \text{foo.meth()} \ //\text{invokevirtual-}\to \ \text{call}
      \end{aligned}
      \]
Low-level & Machine Intermediate Representation

- Translate HIR of a method to LIR
  - Expands operations like calls/dispatches
  - Optimize
    - Available field and method offsets (constants)
    - **Local** common subexpression elimination
  - Reorder instructions in each basic block
    - Optimize for co-location in pipeline, and avoid memory stalls

- Convert LIR to machine-specific MIR
  - Assembly with infinite number of registers
  - Code generation (via Bottom-up Rewriting System (BIRS))
    - Map MIR (grammar) to native (grammar)
    - Efficiency (cycles) computed using dynamic programming

- Convert infinite symbolic registers to physical registers
  - Register Allocation: Linear Scan greedy algorithm
Register Allocation: Starting place

- Symbolic registers are mapped to physical registers
  - Find the **live ranges** (aka live variable analysis)
    - **The range of instructions a register is used**
    - From the first assignment/write into the register (**def**)
    - To the next **def** of that register
      - Or to the last **use** (read) of the register if there is no next **def**

```
1)  R3 = &a  //param
2)  R2 = c  //param
3)  R1 = R3[R2]
4)  R4 = R2 + 1
5)  R2 = R4 * R1
6)  return R2
```
Variable/Register Live Ranges

- Symbolic registers are mapped to physical registers
  - Find the live ranges (aka live variable analysis)
    - The range of instructions a register is used
    - All compilers do some form of this
    - It's how they assign registers given this information that varies widely
  - Draw them and look for maximal overlap
    - = number physical registers needed to put all in registers (the goal)
    - Captures max interference – any that don’t fit must be spilled to memory (costly=memory access + more instructions)

```
R3 = &a  //param
R2 = c  //param
R1 = R3[R2]
R4 = R2 + 1
R2 = R4 * R1
return R2
```
JikesRVM Linear Scan Register Allocator

- Assign physical registers to symbolic registers - JikesRVM
  - Linear scan over instructions (top to bottom)
    - Greedily allocate physical registers in a single linear time scan of the symbolic registers’ live ranges
  - Its fast and does a decent job at allocating

- Live interval \([i, j]\) for variable \(v\)
  - There is no instruction \(i' < i\) for which \(v\) is live
  - There is no instruction \(j' \geq j\) for which \(v\) is live
  - There may be intervals in \([i, j]\) for which \(v\) is not live
    - These are disregarded
    - \([i, j]\) is \textbf{maximal} live interval for \(v\)
  - Interference between variables is caught by interval overlap
Linear Scan Register Allocator

Algorithm

- Store intervals in array in increasing interval_start order

  A: [1,4]
  B: [2,5]
  C: [3,10]
  D: [4,8]
  E: [5,7]

- Keep “active” list at each step through the array
  - List of intervals that overlap the current point that have been placed in registers
  - In order of increasing interval_end

- Active list

<A,B> means A & B have overlapping/interfering intervals are placed in registers and are currently active. In addition, A’s end is smaller than B’s
Linear Scan Register Allocator

- **Algorithm**
  - Store intervals in array in increasing interval_start order
  - Keep “active” list at each step through the array
  - Spill if number in active list > number of available registers
    - Make one of the elements go to memory to get the data
    - Choose the one with the largest interval_end
      - Heuristic that says, if its end is way out there its going to overlap with other intervals so spill it to allow the other intervals say in registers
      - Just an educated guess, so this in some cases will not be optimal
        - You’ll spill an important, heavily used variable or you will have registers sitting around not doing anything
        - ... but its fast (much faster than graph coloring used in static compilers)
  - Allocate physical registers as intervals become “active”
Linear Scan Register Allocator

- **Algorithm**
  - Store intervals in array in increasing interval_start order
  - Keep “active” list at each step through the array
  - Spill if number overlapped > number of available registers
  - Allocate as intervals become “active”

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>[1,4]</td>
</tr>
<tr>
<td>B</td>
<td>[2,5]</td>
</tr>
<tr>
<td>C</td>
<td>[3,10]</td>
</tr>
<tr>
<td>D</td>
<td>[4,8]</td>
</tr>
<tr>
<td>E</td>
<td>[5,7]</td>
</tr>
</tbody>
</table>

```
Available registers: 2
```

```
<A>   

<A,B> 

<A,B,C> spill C: interval with largest interval_end

<B,D> 

<E,D>
```
Linear Scan Register Allocator

Algorithm

- Store intervals in array in increasing interval_start order
- Keep “active” list at each step through the array
- Spill if number overlapped > number of available registers
- Allocate as intervals become “active”

A: [1,4]
B: [2,4]
C: [3,10]
D: [4,8]
E: [5,7]

Available registers: 2

- <A> <A,B> <A,B,C> spill C: interval with largest interval_end
- <D> here, allocation is non-optimal, spill + available register (=cost of heuristic)
Bytecode translation via compilation

- Bytecode → HIR (abstract interp) + basic optimizations
- HIR → LIR (expand invokes/calls) + CSE + reorder instructions
- LIR → MIR (instr. scheduling) + Register Allocation
- Prologue/epilogue added to method
  - Prologue
  - Epilogue
Bytecode translation via compilation

- Bytecode → HIR (abstract interp) + basic optimizations
- LIR → LIR (expand calls) + CSE + data dependencies
- LIR → MIR (instr. scheduling) + Register Allocation
- Prologue/epilogue added to method
  - **Prologue**
    - Allocate runtime stack frame
    - Save any nonvolatile registers
    - Check whether a thread yield has been requested
    - Lock if the method is synchronized
  - **Epilogue**
    - Restore any nonvolatile registers
    - Store return value
    - Unlock if the method is synchronized
    - Deallocate the runtime stack frame
    - Branch to return address
Bytecode translation via compilation

- Bytecode $\rightarrow$ HIR (abstract interp) + basic optimizations
- LIR $\rightarrow$ LIR (expand calls) + CSE + data dependencies
- LIR $\rightarrow$ MIR (instr. scheduling) + Register Allocation
- Prologue/epilogue added to method
  - Prologue
  - Epilogue

- **Store compiled code block in memory at address**
  - Convert intermediate-instruction offsets to machine code offsets
    - For exception handling
    - For garbage collection (reference maps)
  - Update VM tables (statics or VMTs) with address
  - Jump and execute (JIT-compiled methods), fixing up the stack to return to the caller of the JIT'd method