Interpretation and Interpreter Optimization
Bytecode ISA

- **JVM**
  - Typed instructions
  - Opcode: 1-byte wide (253 are used)
  - Data: zero or more values to be operated on (operands)

- **MSIL**
  - Typed instructions
  - Opcode is 2-bytes (64K possible)

- **Python**
  - 113 opcodes (42 with arguments/operands and 71 without)

- All use operand stack for one or more of their operands
- Translator must translate this ISA to native code
Translation/Execution Options for Bytecode

• Transfer bytecode; interpret bytecode at the target
  ■ Verify bytecode at load time (for type safety if required)
  ■ Line by line execution, with some optimizations

• Transfer bytecode; compile bytecode at the target
  ■ Verify bytecode at load time
  ■ Translate multiple bytecode instructions to native code
    ▸ Method-level, path-level (trace compilation); JIT vs Dynamic

• Transfer native code: Ahead of Time (AoT) compilation
  ■ Requires whole program; compiles everything even stuff that doesn’t execute
  ■ Requires safety checks at target and greater trust; **hard** to sandbox
  ■ Greatly simplifies runtime and reduces runtime overhead
  ■ Used for smartphones (android/ios) and for some secure systems: many fear that dynamic code generation is dangerous
Translation/Execution Options for Bytecode

- Many 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
  - Ahead-of-time (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)
Translation/Execution Options for Bytecode

- Note that there is no such thing as a "interpreted language" or a "compiled language" – any language can be interpreted or compiled!

- Today: interpretation and optimizing the interpretation process
  - Additional reading if you are interested: http://craftinginterpreters.com/introduction.html
Process Address Space

Executable & Linkable Format (ELF): Linux/GNU
Running Interpreter (VM) Memory: Javascript V8

Also:
For the VM: global statics/symbol table, internal (code/data/obj) representations and maps, GC book keeping data structures, ...

For the program: objects allocated

Process Runtime Heap

Process Runtime Stack

stack pointer
program counter
accumulator
receiver
a0
a1
return address
closure
frame pointer
...
Translation

• **Interpretation**
  - **Line-by-line execution of a program**
    - If a statement is in a loop, the statement is processed repeatedly
  - For each instruction X, parse X and implement its semantics using another language
    - Instructions may be broken down into multiple operations
    - There is a *handler* for each operation

```java
static void foo() {
    C tmpA3 = new C();
    int k = tmpA3.mc();
    while (k > 0 && k > C.fielda) { //bytecodes 13-37
        tmpA3.fieldc += k--;
    }
}
```

```java
Code:
static void foo();
0: new           #7  // class C
3: dup
4: invokespecial #8  // Method "<init>":()V
7: astore_0
8: aload_0
9: invokevirtual #9  // Method mc():
12: istore_1
13: iload_1
14: ifle           40
17: iload_1
18: getstatic     #10 // Field fielda:
21: if_icmple     40
24: aload_0
25: dup
26: getfield      #5  // Field fieldc:
29: iload_1
30: iinc           1, -1
33: iadd
34: putfield      #5  // Field fieldc:
37: goto           13
40: return
```
Translation

- **Interpretation**
  - Line-by-line execution of a program
    - If a statement is in a loop, the statement is processed repeatedly
  - For each instruction X, parse X and implement its semantics using another language
    - Instructions may be broken down into multiple operations
    - There is a *handler* for each operation

```java
static void foo() {
    C tmpA3 = new C();
    int k = tmpA3.mc();
    while (k > 0 && k > C.fielda) { //bytecodes 13-37
        tmpA3.fieldc += k--;
    }
}
```
Translation

• Interpretation
  ■ Line-by-line execution of a program
    ▶ If a statement is in a loop, the statement is processed repeatedly
  ■ For each instruction X, parse X and implement its semantics using another language (e.g. C)
    ▶ Instructions may be broken down into multiple operations
    ▶ There is a handler for each operation – that implements the operation
      Read/parse next instruction (iadd), call handler
      iadd handler:
        pop tos into variable x (type int)
        pop tos into variable y (type int)
        int z = x+y
        push z on tos

Interpreter/runtime maintains an actual operand stack for each method in memory along with other data structures (statics table)
Translation

• Interpretation
  - Line-by-line execution of a program
    - If a statement is in a loop, the statement is processed repeatedly

• Benefits – this would be a great quiz question...
  - Great for fast prototyping of new languages/instructions
  - Can be used to define operational semantics of a language (e.g. Ruby)
  - **Portable** if written in a highlevel language -- simply **recompile** runtime
    - Compiler VM generates native (binary) code for a particular architecture
      - Requires porting ("retargeting") for each architecture
  - Much simpler, easier to debug, construct
  - Smaller footprint - memory, code -- commonly used for embedded devices
  - Interpreting code is much faster than dynamic/JIT compiling (the translation process)
  - Adding tools (profiling, optimizers, debuggers) is easy
Translation

- Interpretation
  - Line-by-line execution of a program
    - If a statement is in a loop, the statement is processed repeatedly
    - Fastest interpreters are 5-10x slower than executable native code
    - Could be 100x or more however for some programs
  - All bytecode languages (representations) can be executed this way

- Implementation
  - Decode and dispatch loop – AKA switch-dispatch interpretation
Interpretation (cPython): Switch-d dispatch

for(;;){
    //check for thread-switching/signals .. etc.

    ... 

    //read next VM instruction from bytecode file, extract opcode
    opcode = NEXTOP();
    // opcode has an arg ?
    if (HAS_ARG(opcode))
        oparg = NEXTARG();

    switch (opcode) {
    case NOP: break;
    case LOAD_FAST: loadfast(...); break;
    ...
    }
}
Bytecode ISA

- JVM
  - Typed instructions
  - Opcode: 1-byte wide (253 are used)
  - Data: zero or more values to be operated on (operands)

- MSIL
  - Typed instructions
  - Opcode is 2-bytes (64K possible)

- Python
  - 113 opcodes (42 with arguments and 71 without)

- All use operand stack for one or more of their operands
- Translator must translate this ISA to native code
for(;;){
    //check for thread-switching/signals .. etc.
...
    //read next VM instruction from bytecode file, extract opcode
    opcode = NEXTOP();
    // opcode has an arg ?
    if (HAS_ARG(opcode))
        oparg = NEXTARG();

    switch (opcode) {
        case NOP: break;
        case LOAD_FAST: ... break;
...
    }
}
Optimizing the Interpreter Loop: Do the obvious!

```c
for(;;){
    //check for thread-switching/signals .. etc.
    ...
    //read next VM instruction from bytecode file, extract opcode
    opcode = NEXTOP();
    // opcode has an arg ?
    if (HAS_ARG(opcode))
        oparg = NEXTARG();

    switch (opcode) {
        case NOP: break;
        case LOAD_FAST: ... break;
        ...
    }
}
```
Control and Data Flow Comparison

- Interpretation routines AKA handlers
- Dispatch loop
- Contains many branches (both direct and indirect)
  - Direct == target in instr
  - Indirect == target in register (need lookup)

Native execution        Decode & Dispatch

Control flow

Data flow (read source instructions)
Control and Data Flow Comparison

- **Native execution**
- **Decode & Dispatch**

- **Byte code**
- **Dispatch loop**
- **Interpreter routines AKA handlers**

- Contains many branches (both direct and indirect)

```plaintext
if ... then i_a //taken br
i_3 //fallthrough
```

- `i_2` is a conditional branch
- Hardware predicts its not taken, ie that the fallthrough instr `i_3` is next
- CPU computes branch target in EX
  - and finds out that its TAKEN!
  - `i_3` and `i_4` are mistakes! a MISS

Start correct instruction `i_a`
Flush `i_3` and `i_4` (bubble in pipeline)
(Un-)Conditional Branches

BB1: \[ p = 0 \]
    \[ i = 1 \]

BB2: \[ p = p + i \]
    if \( p \leq 60 \) goto BB4

BB3: ... goto BB4

BB4: ...

\( i_2 \) is a conditional branch

Hardware predicts its not taken, ie that the fallthrough instr \( i_3 \) is next

CPU computes branch target in EX
- and finds out that its TAKEN!
- \( i_3 \) and \( i_4 \) are mistakes! a MISS

Start correct instruction \( i_a \)

Flush \( i_3 \) and \( i_4 \) (bubble in pipeline)
Predicting Indirect Branches

- Branch Target Buffer (BTB) – in CPU/chip hardware
  - An idealized BTB contains one entry for each branch and predicts that the branch jumps to the same target as the last time it was executed.
  - The size of real BTBs is limited, resulting in capacity and conflict misses.
Interpreter: Switch-dispatch

```c
//interpreter loop
for(;;){
    //checks...
    //read/parse next bytecode instr
    opcode = NEXTOP();
    switch (opcode) {
        case NOP: break;
        case IADD: iadd_handler(); break;
    }
}
```

- Contains many branches (both direct and indirect)
- Typically difficult to predict:
  - Switch-case (register indirect)
  - Call to interp routine
  - Return from interp return (indirect branch)
  - Loop end test/branch

![Diagram](image-url)
Interpreters: Switch-case dispatch

//interpreter loop
for(;;){
    //checks
    ...
    //read/parse next bytecode instr
    opcode = NEXTOP();
    switch (opcode) {
    case NOP: break;
    case IADD: iadd_handler(); break;
    }
}

- Contains many branches (both direct and indirect)
- Typically difficult to predict:
  - Switch-case (register indirect)
  - Call to interp routine
  - Return from interp return (indirect branch)
  - Loop end test/branch

- Optimizations are needed to speed up the process
  - Reduce number of dispatches
  - Reduce the overhead of a single dispatch
  - the interpreter loop
    - fewer branches
    - more predictable branches

//interpreter loop
for(;;){
    //checks
    ...
    //read/parse next bytecode instr
    opcode = NEXTOP();
    switch (opcode) {
    case NOP: break;
    case IADD: iadd_handler(); break;
    }
}
Switch-Case:

```java
    inst = getFirstInst();
    while((inst!=null)
    {
        opcode = getOpcode(inst);
        switch (opcode){
            case opA:
                opA_handler(inst);
                break;
            case opB:
                opB_handler(inst);
                break;
            ...
        }
        inst = getNextInst(inst);
    }
    finish();
```

Optimization 1:

- get rid of the outer loop (test/branch per each instruction interpreted) and switch

- get rid of the function calls (and their returns) for each opcode
  - 1-call, 1-return per instruction interpreted
  - Returns are typically indirect jumps

  Get rid of the return
  Replace the call

To enable this: Put all of the handler code at specific/known locations in memory, and put their addresses in a lookup table (indexed by opcode)

- inline (aka "thread") the handlers into one long interpreter code body
Indirect Threading (ITI)

Switch-Case:
inst = getFirstInst();
while((inst!=null)
{
  opcode = getOpcode(inst);
  switch (opcode){
    case opA:
      opA_handler(inst);
      break;
    case opB:
      opB_handler(inst);
      break;
    ...
  }
  inst = getNextInst(inst);
}
finish();

ITI:
inst = getFirstInst();
if (inst==null) finish();
opcode = bit_shift_inst;
handler = handlers[opcode];
goto *handler;
...

OPA_LABEL:
... /* implement opcode A */
inst = getNextInst(inst);
if (inst==null) finish();
opcode = bit_shift_inst;
handler = handlers[opcode];
goto *handler

OPB_LABEL:
...

Eliminates: switch-case (register indirect) & loop
Improves: prediction for handler target (if opcodes occur in the same sequences – which they do)
Adds: Lookup table for handler address
**Direct Threading (DTI)**

### Switch-Case:
```java
inst = getFirstInst();
while((inst!=null)
{
    opcode = getOpcode(inst);
    switch (opcode){
        case opA:
            opA_handler(inst);
            break;
        case opB:
            opB_handler(inst);
            break;
        ...
    }
    inst = getNextInst(inst);
}
finish();
```

---

**Direct Threading (DTI):**
```java
inst = getFirstInst();
if (inst==null) finish();
handler = bit_shift_inst;
goto *handler;
...
```

**OPA_LABEL:**
```java
    ... /* implement opcode A */
    inst = getNextInst(inst);
    if (inst==null) finish();
    handler = bit_shift_inst;
goto *handler;
```

**OPB_LABEL:**
```
...
```

---

Eliminates: lookup table for handler address

Gets: same benefits as ITI

Adds: Translation of each instruction executed (once): opcode_operands -> handlerAddr_operands
- necessarily increases the instruction size from 1 byte to 4 bytes

```
iadd -> 0x60 -> 0x8852771A
```
Direct Threading (DTI)

Switch-Case:
inst = getFirstInst();
while((inst!=null)
{
    opcode = getOpcode(inst);
    switch (opcode){
        case opA:
            opA_handler(inst);
            break;
        case opB:
            opB_handler(inst);
            break;
        ...
    }
    inst = getNextInst(inst);
}
finish();

Direct Threading (DTI):
inst = getFirstInst();
if (inst==null) finish();
handler = bit_shift_inst;
goto *handler;
...

OPA_LABEL:
... /* implement opcode A */
inst = getNextInst(inst);
if (inst==null) finish();
handler= bit_shift_inst;
goto *handler;
OPB_LABEL:
...

Eliminates: lookup table for handler address
Gets: same benefits as ITI
Adds: Translation of each instruction executed (once): opcode_operands -> handlerAddr_operands
-- necessarily increases the instruction size from 1 byte to 4 bytes

Requires GNU C and lables-as-values (not supported by ANSI C)
Direct Threading (DTI)

```
Direct Threading (DTI):
inst = getFirstInst();
if (inst==null) finish();
handler = getOpcode(inst);
goto *handler;
...
OPA_LABEL:
... /* implement opcode A */
inst = getNextInst(inst);
if (inst==null) finish();
handler = bit_shift_inst;
goto *handler;
OPB_LABEL:
...
```

iadd -> 0x60 -> 0x8852771A

Eliminates: lookup table for handler address
Gets: same benefits as ITI
    More predictable branching

- Improves branch prediction performance over decode & dispatch
- when the next instruction is the same as last time
Direct Threading (DTI):

```
inst = getFirstInst();
if (inst==null) finish();
handler = bit_shift_inst;
goto *handler;
```

```
OPA_LABEL:
... /* implement opcode A */
inst = getNextInst(inst);
if (inst==null) finish();
handler = bit_shift_inst;
goto *handler;
```

```
OPB_LABEL:
...
```

```
iadd -> 0x60 -> 0x8852771A
```

Eliminates: lookup table for handler address

Gets: same benefits as ITI

More predictable branching

- Improves branch prediction performance over decode & dispatch
- But pattern repetition is limited
  - iload->iadd
  - iload->iload

data segment VM Code VM routines (handlers)

- iload
- iadd
- iload
- iload

Machine code (handler) for iload; dispatch next
Machine code for iadd; dispatch next
Control and Data Flow Comparison

Native execution

Decoded & Dispatch

Direct Threaded Interpretation

Byte code

Interpreter routines

Dispatch loop

Control flow

Data flow (read source instructions)
Interesting Interpreter Measurements: % Time Spent

Java

Python

Java has “thin” bytecodes
- less work done in handler of each opcode

Python has “fat” bytecodes
- more work done in handler of each opcode
• More cycles per dispatch for Python bytecodes – “fat” bytecodes
  • Type-generic instructions (lots of work needed from interpreter)
    • EX: BINARY_ADD – add’s two objects, different semantics depending on object types
  • Built-in semantics: EX: print for lists, tuples, strings
    • Java breaks this up into individual bytecodes/calls libs
Interesting Interpreter Measurements: Dispatch Rate

- **Java:** High dispatch-to-work ratio
  = thin bytecodes

- **Python:** Lower dispatch-to-work ratio
  = fat bytecodes

Anomoly:
POWER f/n in loop
- built in for Python
- call to Math lib for Java

... another good quiz question: what does “fat” or “thin” mean in terms of bytecode instructions?
Interpreter Optimization
(Un-)Conditional Branches

BB1: \[ p = 0 \]
\[ i = 1 \]

BB2: \[ p = p + i \]
if \( p \leq 60 \) goto BB4

BB3: ... goto BB4

BB4: ...

BB1: \[ p = 0 \]
\[ i = 1 \]

If ... then \( i_a \) //taken/jump br
\( i_3 \) //fallthrough

- Contains many branches (both direct and indirect)

CPU

\[
\begin{array}{cccc}
\text{Time} & \text{IF} & \text{DEC} & \text{EX} & \text{WB} \\
1 & i_1 &  &  &  \\
2 & i_2 & i_1 &  &  \\
3 & i_3 & i_2 & i_1 &  \\
4 &  & i_3 &  & i_2 \\
5 &  &  &  & i_1 \\
6 & i_b &  &  & i_2 \\
7 & i_b & i_b &  &  \\
\end{array}
\]

- \( i_2 \) is a conditional branch
- Hardware predicts its not taken, ie that the fallthrough instr \( i_3 \) is next
- CPU computes branch target in EX
  - and finds out that its TAKEN!
  - \( i_3 \) and \( i_4 \) are mistakes! a MISS
- Start correct instruction \( i_a \)
- Flush \( i_3 \) and \( i_4 \) (bubble in pipeline)
Predicting Indirect Branches

- Branch Target Buffer (BTB) – in CPU/chip hardware
  - An idealized BTB contains one entry for each branch and predicts that the branch jumps to the same target as the last time it was executed
  - The size of real BTBs is limited, resulting in capacity and conflict misses
Control and Data Flow Comparison

Native execution  Decode & Dispatch

- Control flow
- Data flow (read source instructions)

Interpreter routines AKA handlers

• Contains many branches (both direct and indirect)

\[ \text{if ... then } i_a \text{ //taken br } \]
\[ i_3 \text{ //fallthrough} \]

\begin{tabular}{|c|c|c|c|}
\hline
Time & IF & DEC & EX & WB \\
\hline
1 & \(i_1\) & & & \\
2 & \(i_2\) & \(i_1\) & & \\
3 & \(i_3\) & \(i_2\) & \(i_1\) & \\
4 & \(\times\) & \(\times\) & \(i_2\) & \(i_1\) \\
5 & & & & \\
6 & \(i_8\) & & \(i_2\) & \\
7 & \(i_6\) & \(i_8\) & & \\
\hline
\end{tabular}

- \(i_2\) is a conditional branch
- Hardware predicts its not taken, ie that the fallthrough instr \(i_3\) is next
- CPU computes branch target in EX
- and finds out that its TAKEN!
- \(i_3\) and \(i_4\) are mistakes! a MISS
- Start correct instruction \(i_a\)
- Flush \(i_3\) and \(i_4\) (bubble in pipeline)
Interpretation: Decode & Dispatch

**Switch-Case:**
inst = getFirstInst();
while((inst!=null)
{
    opcode = getOpcode(inst);
    switch (opcode){
        case opA:
            opA_handler(inst);
            break;
        case opB:
            opB_handler(inst);
            break;
        ...
    }
    inst = getNextInst(inst);
}
finish();

**Optimizations:**
- Indirect Threading
- Direct Threading
Indirect Threading (ITI)

**Switch-Case:**

```java
inst = getFirstInst();
while((inst!=null)
{
    opcode = getOpcode(inst);
    switch (opcode){
    case opA:
        opA_handler(inst);
        break;
    case opB:
        opB_handler(inst);
        break;
    ...
}
inst = getNextInst(inst);
}
finish();
```

**ITI:**

```java
inst = getFirstInst();
if (inst==null) finish();
opcode = bit_shift_inst;
handler = handlers[opcode];
goto *handler;
...
```

**OPA_LABEL:**

```java
... /* implement opcode A */
inst = getNextInst(inst);
if (inst==null) finish();
opcode = bit_shift_inst;
handler = handlers[opcode];
goto *handler
```

**OPB_LABEL:**

```java
...
```

**Eliminates:** switch-case (register indirect) & loop

**Improves:** prediction for handler target (if opcodes occur in the **same sequences** – which they do)

**Adds:** Lookup table for handler address
**Direct Threading (DTI)**

Switch-Case:
```java
inst = getFirstInst();
while((inst!=null)
{
    opcode = getOpcode(inst);
    switch (opcode){
    case opA:
        opA_handler(inst);
        break;
    case opB:
        opB_handler(inst);
        break;
    ...
}     
    inst = getNextInst(inst);
}
finish();
```

---

**Direct Threading (DTI):**
```java
inst = getFirstInst();
if (inst==null) finish();
handler = bit_shift_inst;
goto *handler;
...

OPA_LABEL:
... /* implement opcode A */
inst = getNextInst(inst);
if (inst==null) finish();
handler= bit_shift_inst;
goto *handler;

OPB_LABEL:
...
```

iadd -> 0x60 -> 0x8852771A

**Eliminates:** lookup table for handler address

**Gets:** same benefits as ITI

**Adds:** Translation of each instruction executed (once): opcode_operands -> handlerAddr_operands  
-- necessarily increases the instruction size from 1 byte to 4 bytes
Direct Threading (DTI):

```java
inst = getFirstInst();
if (inst==null) finish();
handler = bit_shift_inst;
goto *handler;
```

... 

```
OPA_LABEL:
... /* implement opcode A */
inst = getNextInst(inst);
if (inst==null) finish();
handler= bit_shift_inst;
goto *handler;
```

```
OPB_LABEL:
...
```

iadd -> 0x60 -> 0x8852771A

Eliminates: lookup table for handler address
Gets: same benefits as ITI
     More predictable branching

- Improves branch prediction performance over decode & dispatch
- when the next instruction is the same as last time
Direct Threading (DTI):

```
inst = getFirstInst();
if (inst==null) finish();
handler = bit_shift_inst;
goto *handler;
...

OPA_LABEL:
... /* implement opcode A */
inst = getNextInst(inst);
if (inst==null) finish();
handler = bit_shift_inst;
goto *handler;
OPB_LABEL:
...
```

- Improves branch prediction performance over decode & dispatch
- But pattern repetition is limited
  - iload->iadd
  - iload->iadd

Eliminates: lookup table for handler address

Gets: same benefits as ITI
  - More predictable branching
Interpretation – Interesting points made in the paper

- 81-98% of indirect branches in switch-dispatch mispredicted by BTBs
- 57-63% of indirect branches in direct-threaded mispredicted by BTBs

- Why is there this difference?
Interpretation – Interesting points made in the paper

- 81-98% of indirect branches in switch-dispatch mispredicted by BTBs
- 57-63% of indirect branches in direct-threaded mispredicted by BTBs

Why is there this difference?

- BTB = hardware branch predictor (aka branch target buffer)
- BTB predicts that this jump target is last jump target
  - prediction is correct only if we end up jumping to the same place as last time

- Switch-dispatch: single indirect branch for all opcodes
- Direct-threaded: A copy of the dispatch sequence is appended to the native code for each VM instruction
  - Each VM (bytecode) instruction has its own indirect branch
Interpretation – Interesting points made in the paper (Continued)

• The biggest problem with interpretation on performance
  ■ Branch mispredictions
  ■ The deeper the pipeline the worse the cost
  ■ Again for bytecodes with high dispatch rates

■ And the overhead of the dispatch loop
  ‣ **Two sources of overhead:** Number of dispatches, cost per dispatch

• Solutions: replication, superinstructions
Each instruction has its own dispatch body (handler)

- Dynamic – make a copy for each instruction, flush icache *dynamically*
  - Concatenate dispatch bodies
  - Requires that code be relocatable
  - Note that this is one dispatch body for each unique instruction in a program
    - Repeated execution of the same instruction will use the same dispatch routine – thus is more predictable

```
Decode & Dispatch
data segment
VM Code       VM routines (handlers)

iload           Machine code (handler) for iload; dispatch next
iadd
iload
iload
iload
```
Interpreter Optimization: Dynamic Replication

- Each instruction has its own dispatch body (handler)
  - Dynamic – make a copy for each instruction, flush icache dynamically
    - Concatenate dispatch bodies
    - Requires that code be relocatable
    - Note that this is one dispatch body for each unique instruction in a program
      - Repeated execution of the same instruction will use the same dispatch routine – thus is more predictable

![Dynamic Replication Diagram]

Diagram:
1. Machine code for iload; dispatch next
2. Machine code for iadd; dispatch next
3. Machine code for iload; dispatch next
Interpreter Optimization: Static Replication

- Each instruction has its own dispatch body
  - Static – make multiple copies for each operation, reroute execution of instructions to different copies --- use a greedy algorithm for rerouting
    - Note that this has no notion of a program – this is done at **interpreter build time**
      - So we have to guess how many copies of each dispatch routine to make
  - **Figuring this out:** Run a bunch of programs, profile them, collect data on the most important instructions and the number of different instances they are likely to have
Interpreter Optimization: Replication Observations

- Each instruction has its own dispatch body
  - Dynamic – make a copy for each instruction, flush icache *dynamically*
    - Performed as the program is run
  - Static – make multiple copies for each operation, reroute execution of instructions to different copies --- use a greedy algorithm for rerouting
    - Performed at interpreter build time

- Much more executable code
- Same number of dispatches (# of bytecode instructions aka operations)
- Same number of indirect branches
  - But more predictable
    - 1 target each so will hit on repeated execution (loops, repeated f/n invokes)
    - Assuming no conflict/capacity misses
Interpreter Optimization: Superinstructions

- Identify basic blocks
  - Straight-line code (one way in (the top) & one way out (the bottom))
  - That ends with some control flow
    - Typically branch, jump, or call
    - Exceptions are control flow but they occur in high-level languages for many
      instructions so, these instructions typically do not end basic blocks
      - If they did, there wouldn’t be any instructions to work with/combine

```
basic block
x = a * 5
y = z[x]
a = a + 1
```
Control-Flow Graph (CFG)

- **Organizing** of the intermediate code in a way that enables efficient analysis and modification
- A simplified representation of a program
  - Function-level
  - But then functions can be linked

- The graph consists of nodes
  - Basic blocks
    - Pieces of straight-line code
    - One entry into it at the top
    - One exit out of it at the bottom
    - No instructions that change control flow inside
  - And edges
    - Control flow edges that show how control can change

```
basic block
x = a * 5
y = z[x]
a = a + 1
```
```plaintext
x = 20;
while (x < 10) {
    x = x - 1;
    A[x] = 10;
    if (x == 4) x = x - 2;
}
y = x + 5;
```
Finding Basic Blocks

- Find set of **leaders**

  - 1) The first tuple of a method is a leader
  - 2) Tuple L is a leader if there is a tuple:
    
    \[
    \text{goto L} \quad \text{if } x \text{ relop } y \text{ goto L}
    \]
  - 3) Tuple M is a leader if it immediately follows a tuple:
    
    \[
    \text{goto L} \quad \text{if } x \text{ relop } y \text{ goto L}
    \]

- A basic block consists of a leader and all of the following tuples except the next leader

- relop = relational operator (conditional check)
Finding Basic Blocks

- Find set of **leaders**
  - 1) The first tuple of a method is a leader
  - 2) Tuple L is a leader if there is a tuple that jumps to L
  - 3) Tuple L is a leader if it immediately follows a tuple that branches (unconditionally or conditionally)

Source code

```
p = 0;
i = 1;
do {
    p += i;
    if (p>60){
        p = 0; i = 5;
    }
    i = i*2 + 1;
}
k = p*3;
```

Intermediate code (IR/IF)

```
1) p = 0
2) i = 1
3) p = p + i
4) if p <= 60 goto 7
5) p = 0
6) i = 5
7) t1 = i * 2
8) i = t1 + 1
9) if i <= 20 goto 3
10) k = p * 3
```
Finding Basic Blocks

- Find set of **leaders**
  - 1) The first tuple of a method is a leader
  - 2) Tuple L is a leader if there is a tuple that jumps to L
  - 3) Tuple L is a leader if it immediately follows a tuple that branches (unconditionally or conditionally)

Source code

```java
p = 0;
i = 1;
do {
    p += i;
    if (p>60){
        p = 0; i = 5;
    }
    i = i*2 + 1;
} 
k = p*3;
```

Intermediate code (IR/IF)

```
1) p = 0
2) i = 1
3) p = p + i
4) if p<=60 goto 7
5) p = 0
6) i = 5
7) t1 = i * 2
8) i = t1 + 1
9) if i<=20 goto 3
10) k = p * 3
```

Leader (rule 1)
Leader (rule 2)
Leader (rule 3)
Basic Blocks and Control Flow Example

BB1:

\[ p = 0 \]
\[ i = 1 \]

BB2:

\[ p = p + i \]
\[ \text{if } p \leq 60 \text{ goto BB4} \]

BB3:

\[ p = 0 \]
\[ i = 5 \]

BB4:

\[ t1 = i \times 2 \]
\[ i = t1 + 1 \]
\[ \text{if } i \leq 20 \text{ goto BB2} \]

BB5:

\[ k = p \times 3 \]
How It Looks in Bytecode

```java
static int f(int i) {
    int retn = i;
    if (i > 10) {
        retn += i*4;
    } else {
        retn += i+4;
    }
    return retn;
}
```

Conditional branches: if_* X
jump to X if condition is true
else fall thru to next instr

Unconditional jumps: goto X
jump to X

```
0 iload_0
1 istore_1
2 iload_0
3 bipush 10
5 if_icmple 17
8 iload_1
9 iload_0
10  iconst_4
11 imul
12 iadd
13 iload_1
14 goto 23
17 iload_1
18 iload_0
19  iconst_4
20 iadd
21 iadd
22 iload_1
23 iload_1
24 ireturn
```
How It Looks in Bytecode

```java
static int f(int i) {
    int retn = i;
    if (i > 10) {
        retn += i*4;
    } else {
        retn += i+4;
    }
    return retn;
}
```

```
0  iload_0
1  istore_1
2  iload_0
3  bipush 10
5  if_icmple 17
8  iload_1
9  iload_0
10  iconst_4
11  imul
12  iadd
13  istore_1
14  goto 23
17  iload_1
18  iload_0
19  iconst_4
20  iadd
21  iadd
22  istore_1
23  iload_1
24  ireturn
```

Fall-through edge

Jump edge
Interpreter Optimization: Superinstructions

- Identify basic blocks
  - Straight-line code
  - That ends with some control flow
    - Typically branch, jump, or call
    - Exceptions are control flow but they occur in high-level languages for many instructions so, these instructions typically do not end basic blocks
      - If they did, there wouldn’t be any instructions to work with/combine

- For each basic block
  - Make a dispatch body (superinstruction)
  - Remove dispatch code in between VM instructions within block
    - Increment VM program counter (PC)
    - Extract address from VM instruction, jump to address

```
iload_handler
iload_handler
iadd_handler
istore_handler
iload_iload_iadd_istore_handler
```

```
0 iload_0
1 istore_1
2 iload_0
3 bipush 10
5 if_icmple 17
```
Interpreter Optimization: Superinstructions

- **Identify basic blocks**
  - Straight-line code
  - That ends with some control flow
    - Typically branch, jump, or call
    - Exceptions are control flow but they occur in high-level languages for many many instructions so, these instructions typically do not end basic blocks
      - If they did, there wouldn’t be any instructions to work with/combine

- **For each basic block**
  - Make a dispatch body (superinstruction)
  - **Remove dispatch code** in between VM instructions within block
    - Increment VM program counter (PC)
    - Extract address from VM instruction, jump to address

- **For identical basic blocks**
  - Use same superinstruction (cost = less predictable branch into/out of)
  - Use replication in combination
Performance Results / Findings

- More benefit for GForth than for JVM
  - JVM has fewer dispatches to begin with for same amount of work
    - Bytecode instructions are “lower-level” – for GForth than for JVM
    - Instructions have types associated with them – for both

- Results
  - Many icache misses avoided, improves performance (up to 4.5X for GForth, 2.7X for JVM)
    - Compared to dynamic compilation: 3-5X for GForth; 9.5X for JVM
  - Dynamic is better
    - Static does ok for GForth but not JVM
  - Combination of replication & superinstructions is better

- Different architectures (w/ different BTBs studied)
  - Using hardware performance counters/monitors
  - Also simulation of different BTBs studied (another paper)