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 and 71 without)

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

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

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

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

```
Read/parse next instruction (iadd), call handler
iadd handler:
   pop tos into variable x
   pop tos into variable y
   z = x+y
   push z on tos
```

Interpreter/runtime maintains the 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
  - 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 (Python)

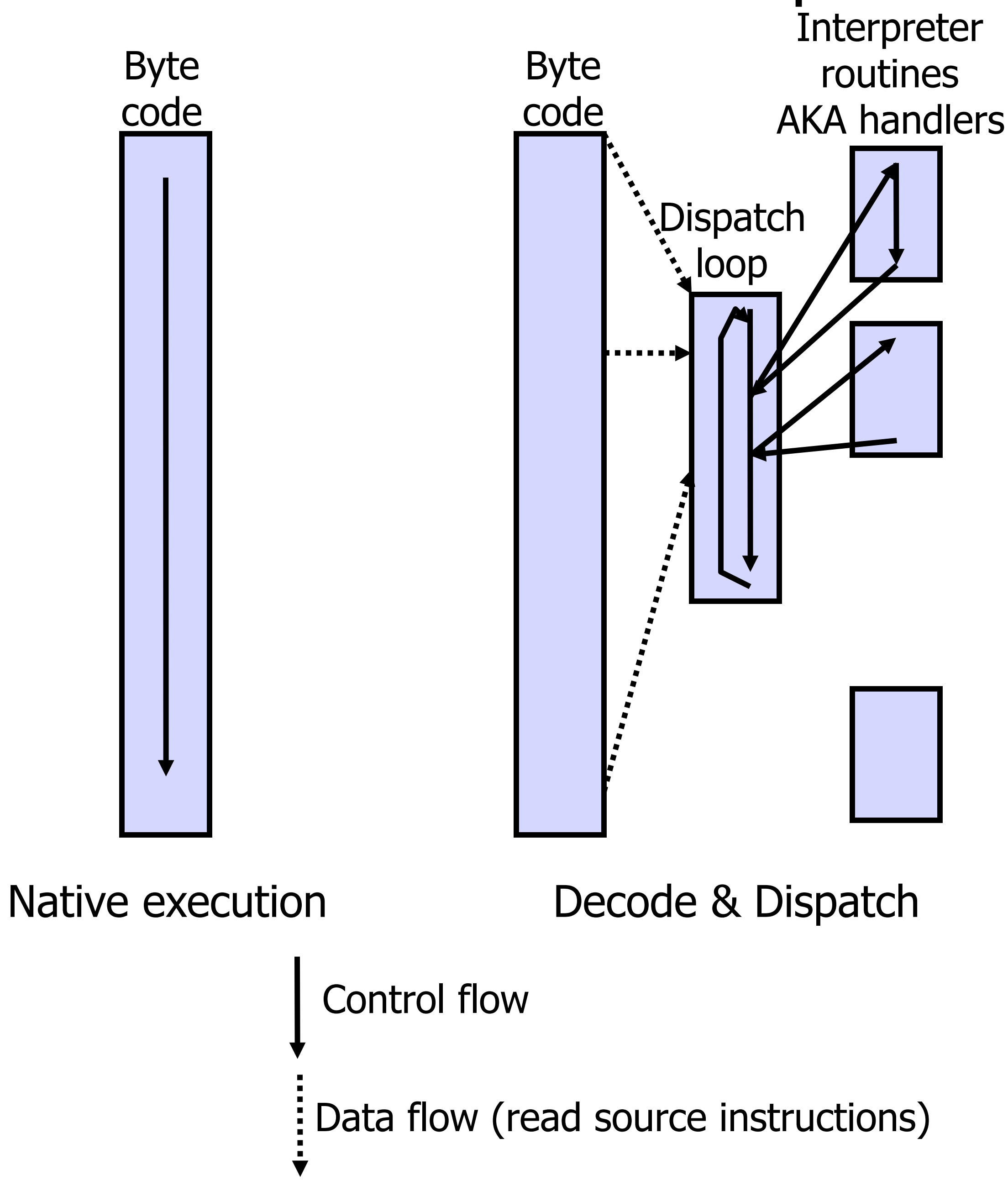
```
for(;;){
  //check for thread-switching/signals .. etc.
  ...
  //read next VM instruction from bytecode file, extract opcode
  opcode = NEXTOP();
  // opcode has an arg ?
  if (HAS_ARG(opcode))
          oparg = NEXTARG();

  switch (opcode) {
  case NOP: break;
  case LOAD_FAST: ... break;
  ...
  }
}
```

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)

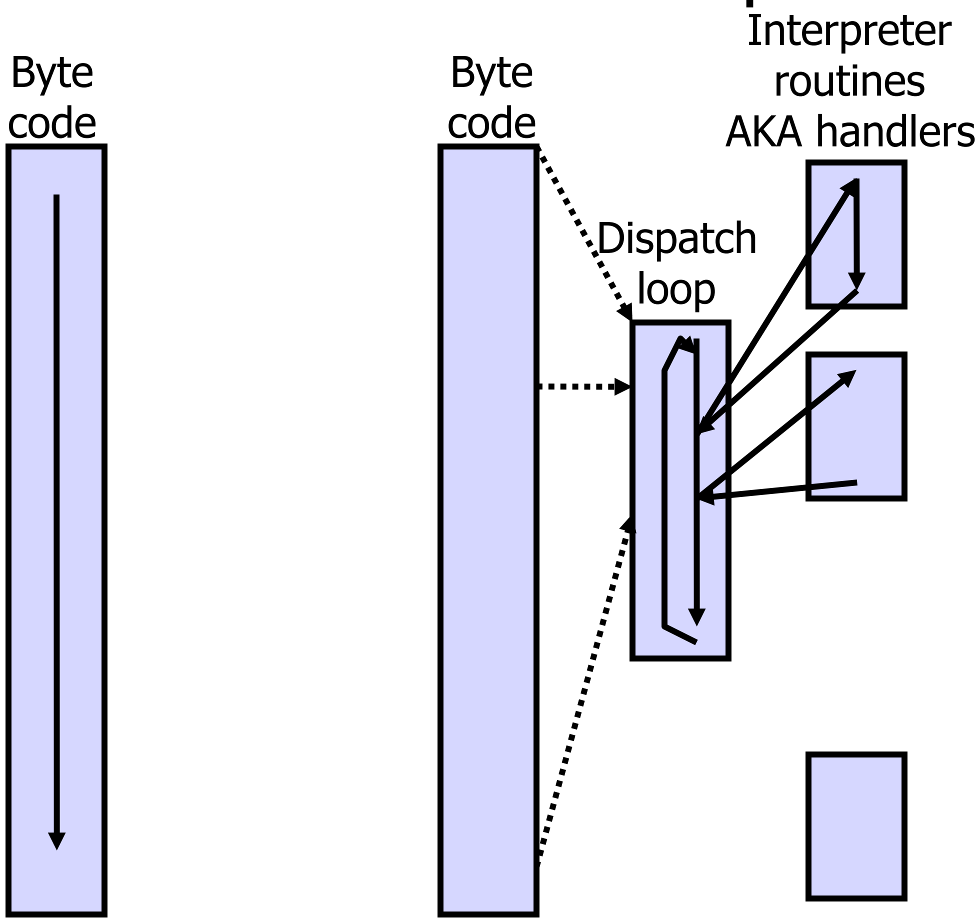
- All use operand stack for one or more of their operands
- Translator must translate this ISA to native code

Control and Data Flow Comparison



- Contains many branches (both direct and indirect)
  - Direct == target in instr
  - Indirect == target in register (need lookup)

Control and Data Flow Comparison



Native execution

Decode & Dispatch

Control flow

Data flow (read source instructions)

- Contains many branches (both direct and indirect)

CPU

Time	IF	DEC	EX	WB
1	i_1			
2	i_2	i_1		
3	i_3	i_2	i_1	
4	~~i_4~~	~~i_3~~	i_2	i_1
5	○	○	○	i_2
6	i_a	○	○	○
7	i_b	i_a	○	○

i_2 is a conditional branch

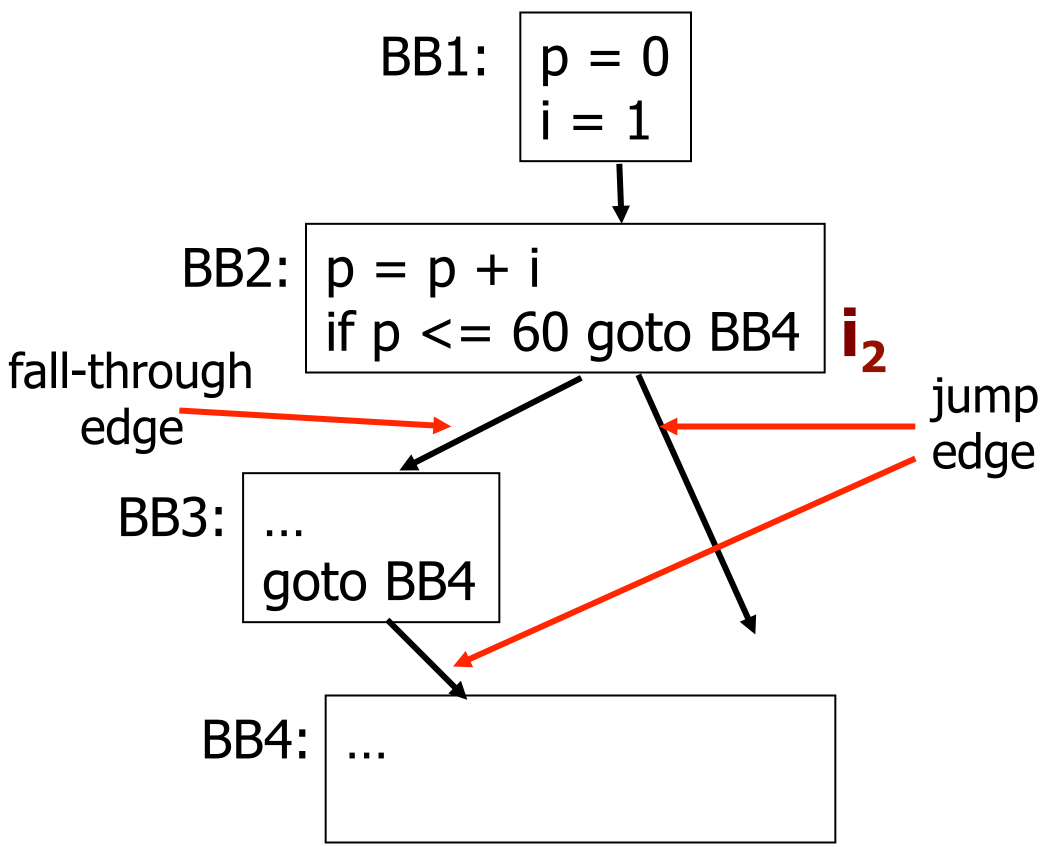
Hardware predicts its not taken, ie that the fallthrough instr i_3 is next

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



- Contains many branches (both direct and indirect)

CPU

Time	IF	DEC	EX	WB
1	i_1			
2	i_2	i_1		
3	i_3	i_2	i_1	
4	$\cancel{i_4}$	$\cancel{i_3}$	i_2	i_1
5	○	○	○	i_2
6	i_a	○	○	○
7	i_b	i_a	○	○

i_2 is a conditional branch

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

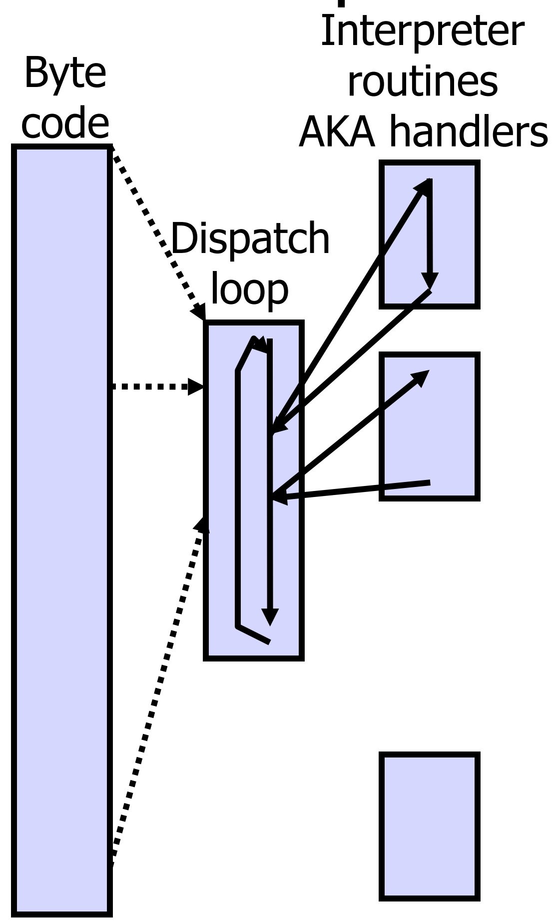
CPU computes branch target in EX
- and finds out that its TAKEN!
- i_3 and i_4 are mistakes! a MISS

Start correct instruction i_a

Flush i_3 and i_4 (bubble in pipeline)

Interpreter: Decode and Dispatch

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



Decode & Dispatch

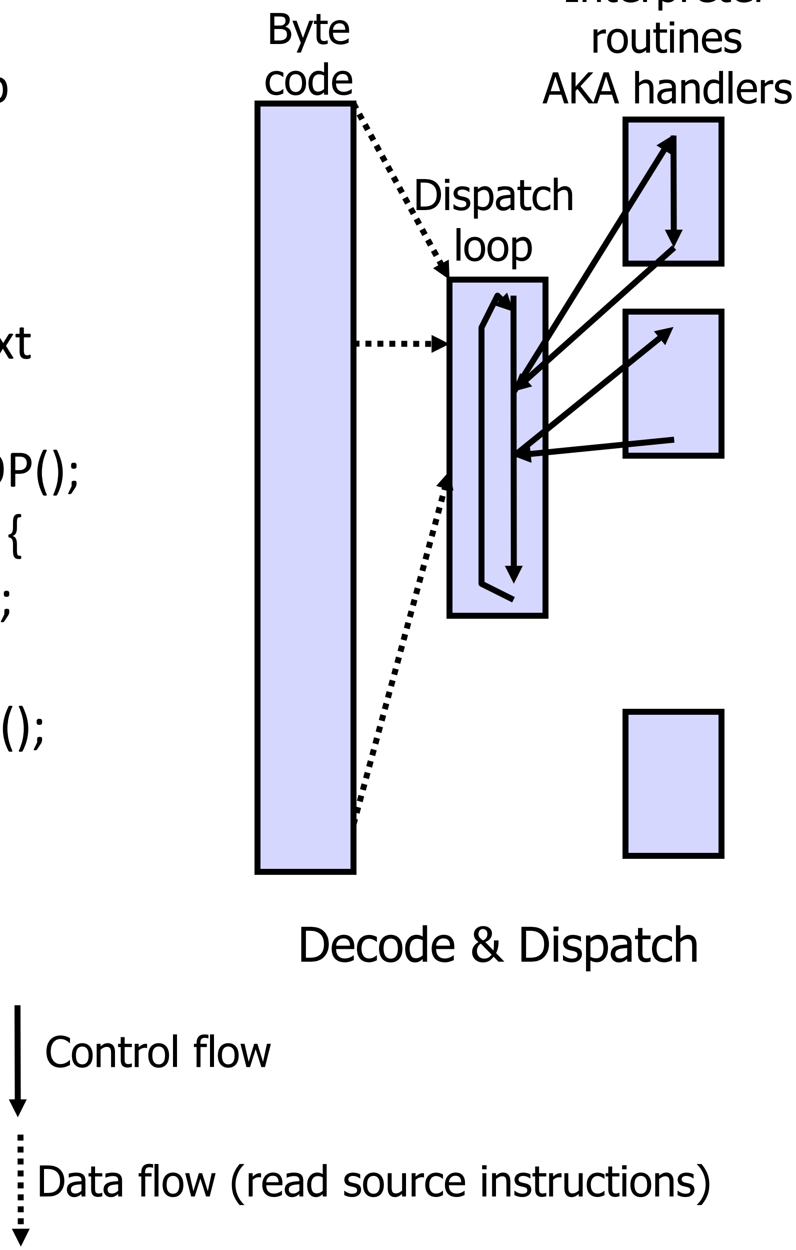
Control flow

Data flow (read source instructions)

- 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

Interpreter: Decode and 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

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();
```

Optimization 1:

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

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

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

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

Elim… …andler address

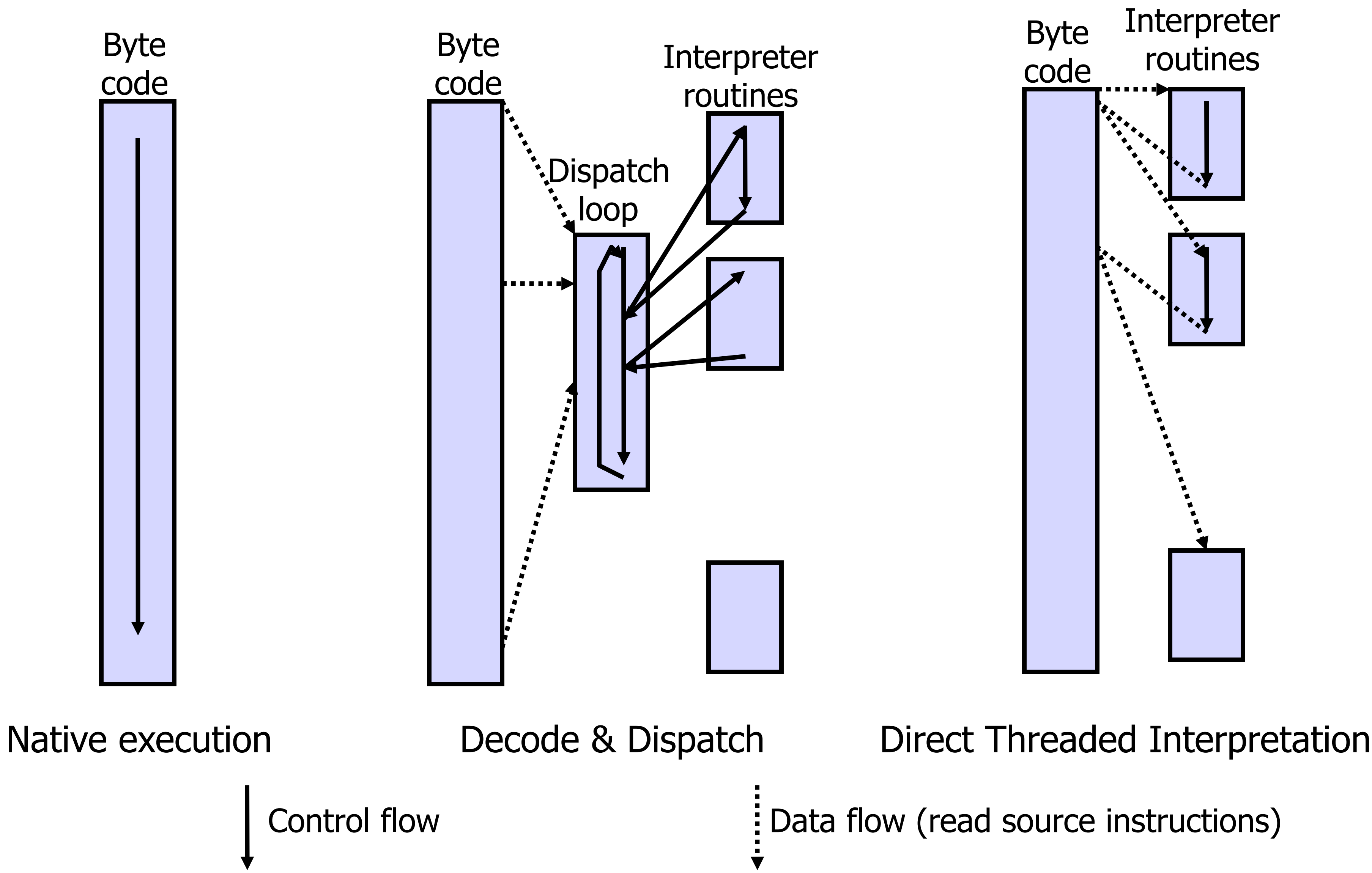
Gets: … as ITI

Adds: …lation of each instruction executed

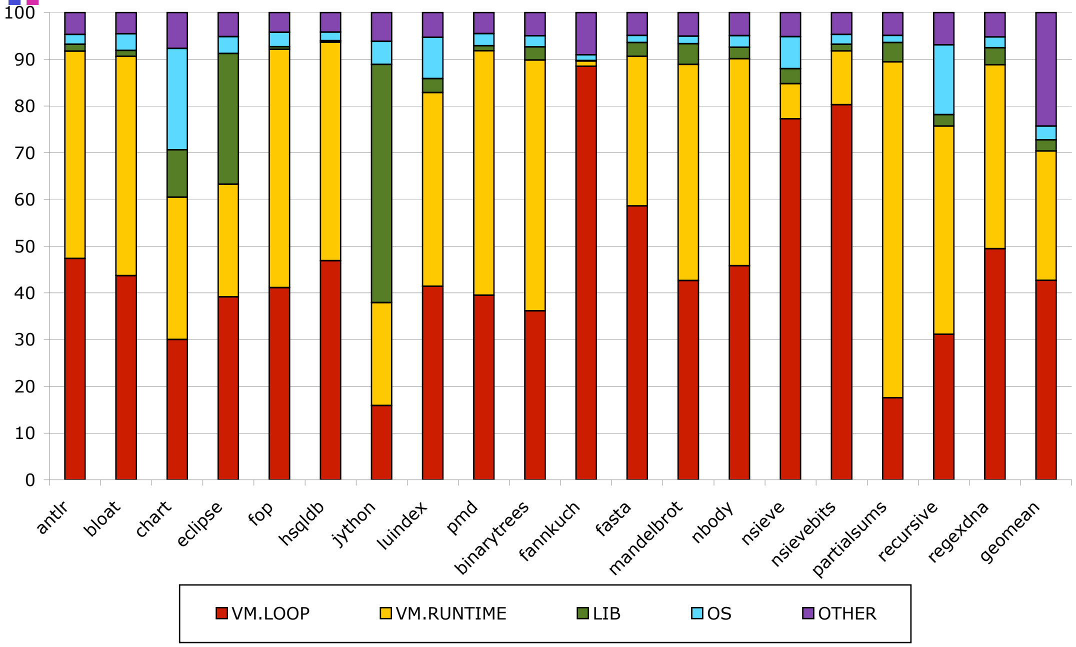
(once): opcode_operands -> handlerAddr_operands

-- necessarily increases the instruction size from 1 byte to 4 bytes

Control and Data Flow Comparison

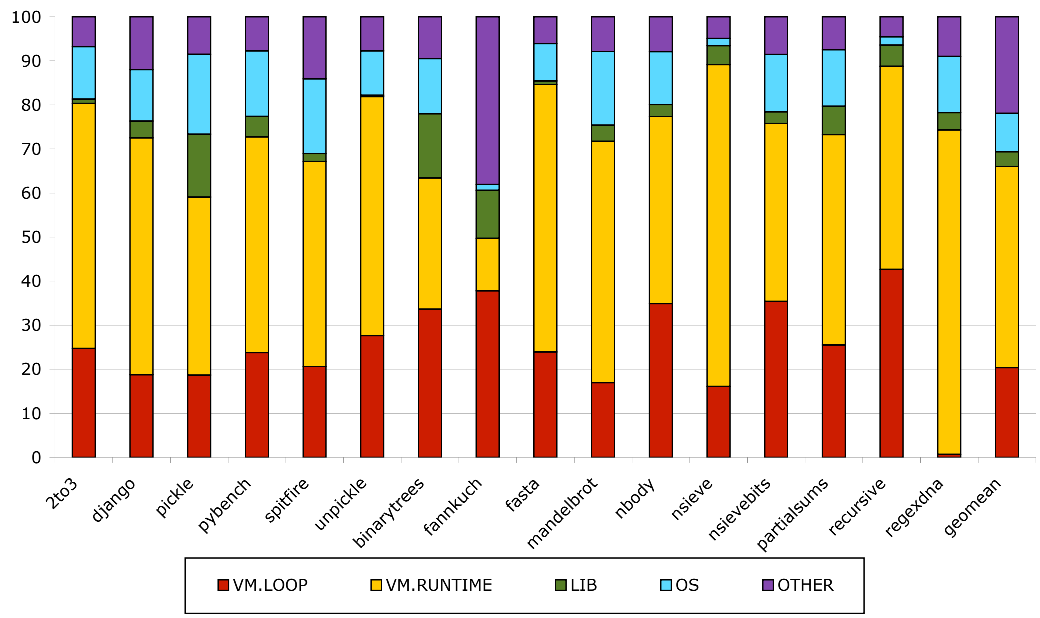


Interesting Interpreter Measurements: % Time Spent

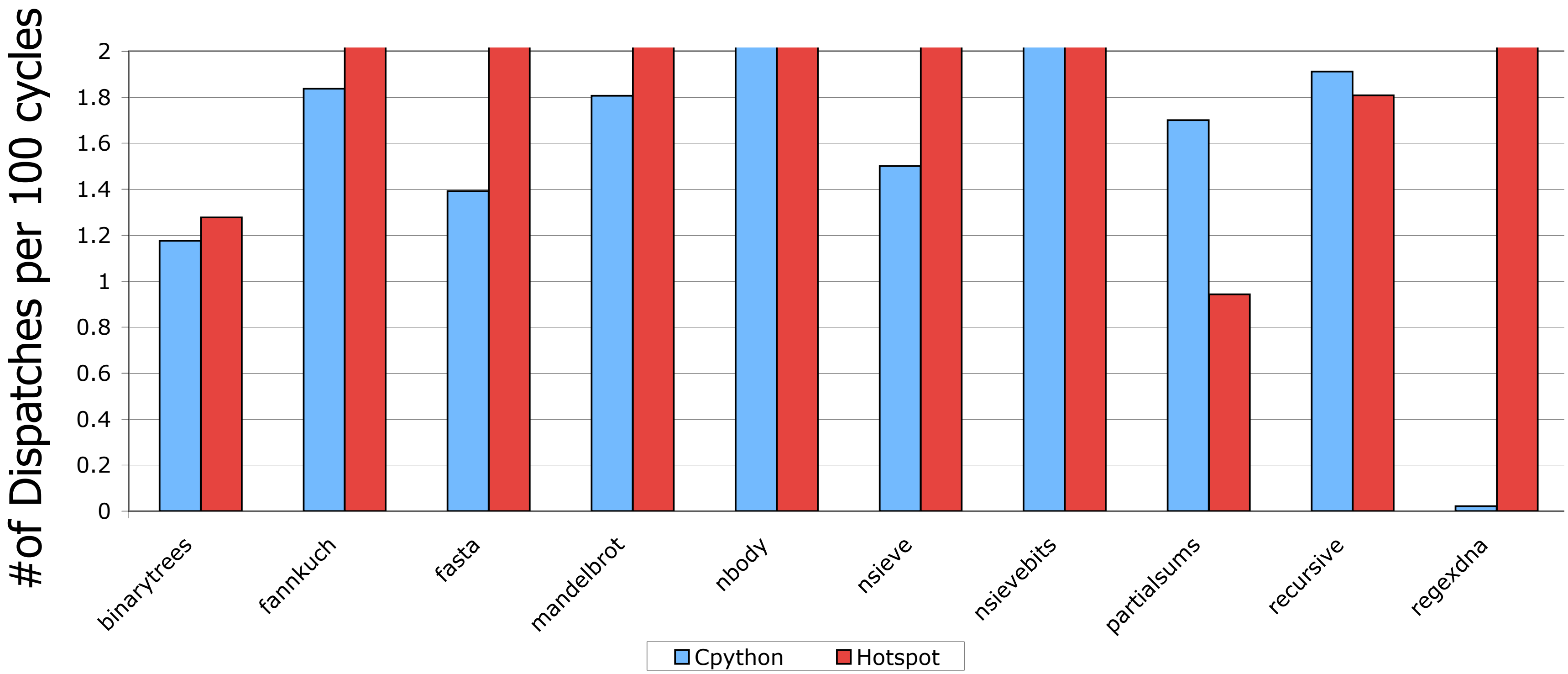


Java

Python



Interesting Interpreter Measurements: Dispatch Rate



- More cycles per dispatch for Python 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

Interpretation – Interesting points made in the paper

- Flat sequence layout of operations vs graph layout
  - Flat sequence is easier to manipulate – fast
  - VM instructions
- "Level" of operations
  - Amount of interpreter work per amount of useful work
    - Impacts the difference in performance between the interpreter and the equivalent native code execution
  - This work targets LOW LEVEL bytecodes
    - Those with high **dispatch-to-work ratios (dispatch rate)**
    - Note that the Python numbers presented earlier
      - Python has low dispatch rates, so interpreter overhead is in the noise
      - That is, these optimizations (that target the interpreter ovehead) aren't likely to have much impact

Interpretation – Interesting points made in the paper

- "Level" of operations
  - Amount of interpreter work per amount of useful work
    - Impacts the difference in performance between the interpreter and the equivalent native code execution
      - Large number of simple operations
        - Interpreters are slowest relative to native code execution
  - JVM vs GForth
    - Dispatch-to-real-work ratio of GForth is higher (simpler VM instructions)
      - JVM – fewer dispatches for same amount of work
    - JVM: more time outside of interpreter loop (GC, verification)
    - GForth caches topmost operand stack element in a register
    - 16.5% of retired machine instructions are ind. branches (6.1% for JVM)
      - Opts that reduce branch misses will benefit GForth more than JVM

Interpretation – Interesting points made in the paper (Continued)

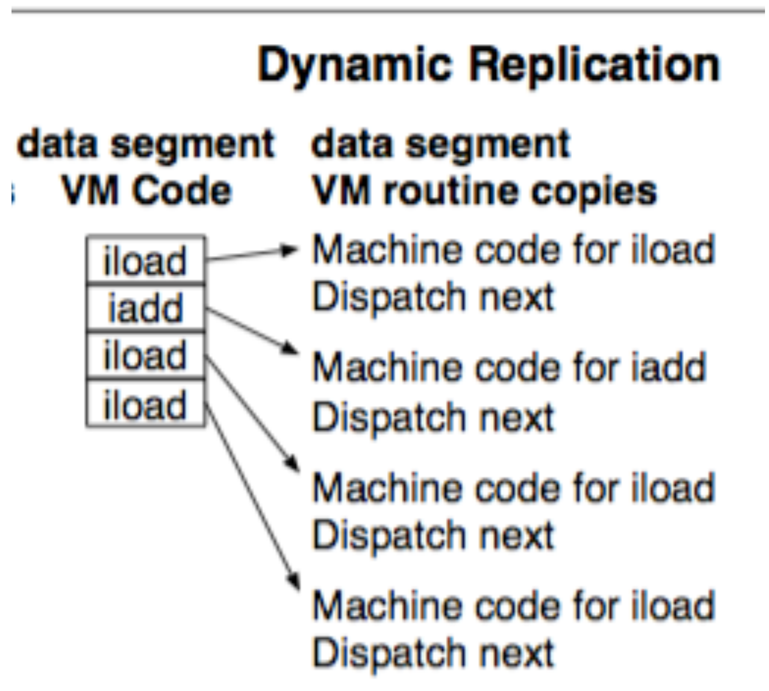
- The biggest problem with interpretation on performance
  - Branch mispredictions
  - The deeper the pipeline the worse the cost
  - **Again for bytecodes with high dispatch rates**

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

- Solutions: replication, superinstructions

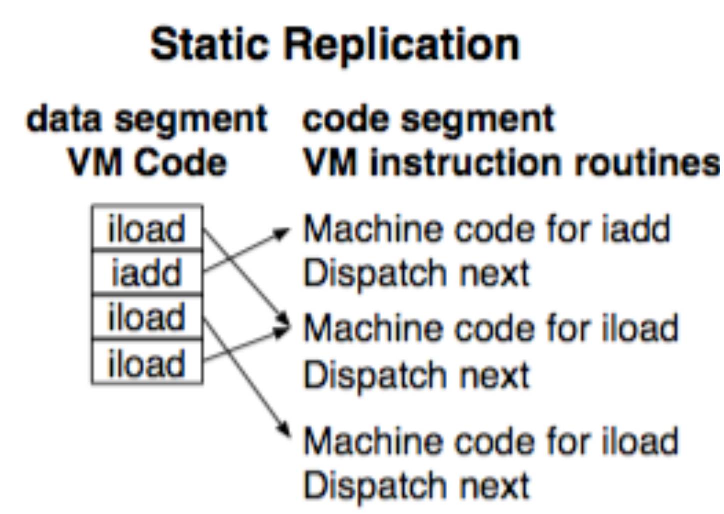
Interpreter Optimization: Dynamic Replication

- Each instruction has its own dispatch body
  - Dynamic – make a copy for each instruction, flush icache ***dynamically***
    - Concatenation of 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



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: 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
    - compiler can optimize across component instructions (keep stack items in registers, combine stack/pointer updates of components, instr. Scheduling
    - Same replic/superinstr set across all programs/inputs (dynamic is customized for current program/input)
    - Note that this has no notion of a program – this is done at **interpreter build time**

Interpreter Optimization: Replication

- 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 VM instructions aka operations)
- Same number of indirect branches
  - But more predictable
    - 1 target each so will hit on repeated execution
    - Assuming no conflict/capacity misses

Interpreter Optimization: Superinstructions

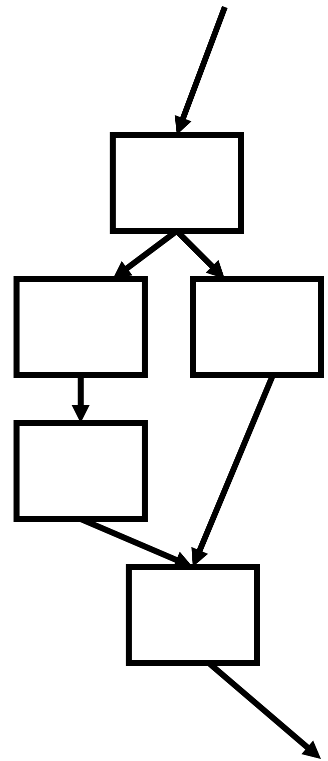
- Identify basic blocks
  - Straight-line code
  - That ends with some control flow
    - Typically branch, jump, or call
    - Exceptions are control flow but they occur in high-level languages for many many instructions so, these instructions typically do not end basic blocks
      - If they did, there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

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

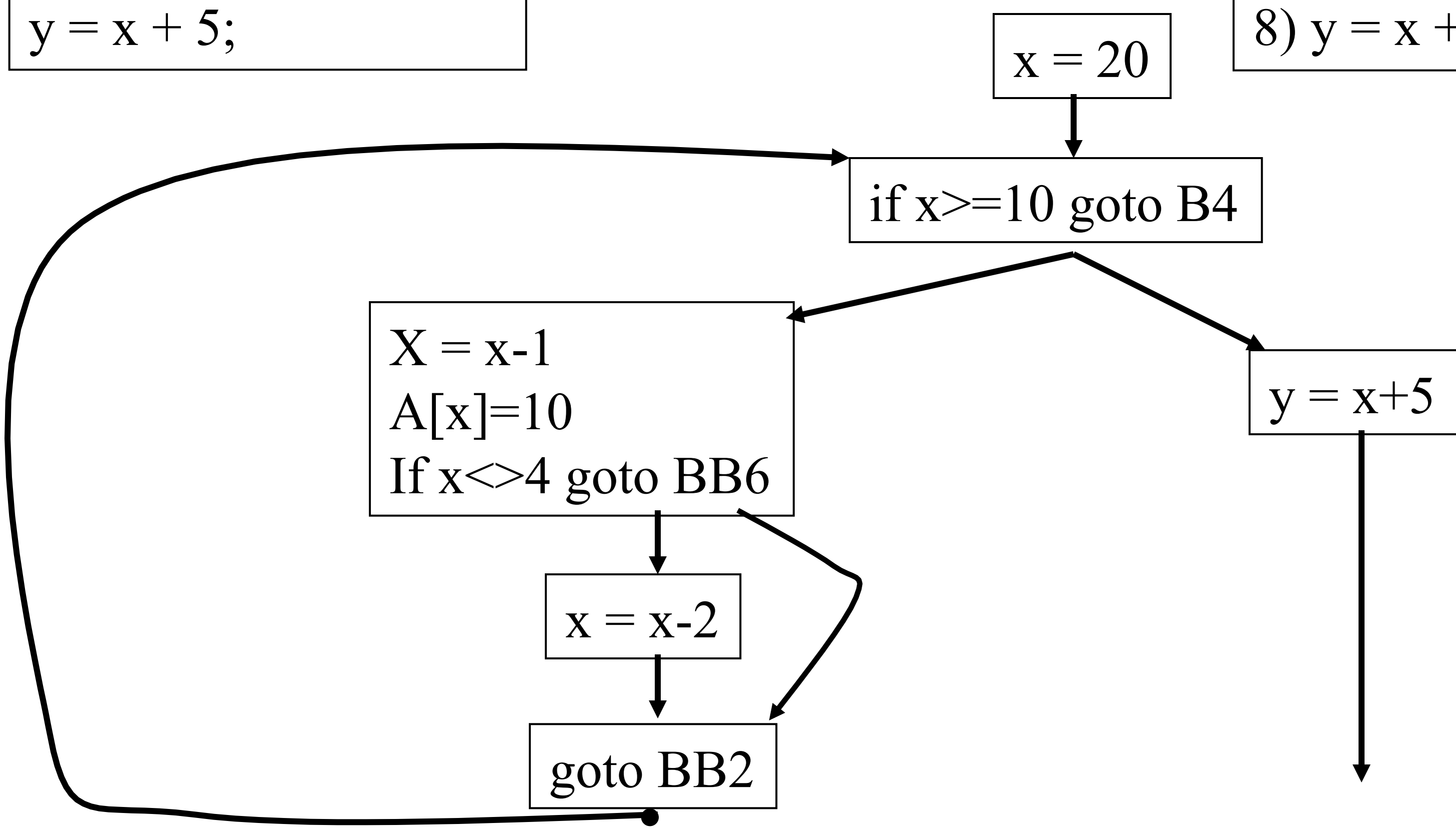
```
x = a * 5
y = z[x]
a = a + 1
```



Basic Blocks and Control Flow

```
x = 20;
while (x < 10) {
  x = x - 1;
  A[x] = 10;
  if (x == 4) x = x - 2;
}
y = x + 5;
```

```
1) x = 20
2) if x>=10 goto 8
3) x = x - 1
4) A[x] = 10
5) if x<>4 goto 7
6) x = x - 2
7) goto 2
8) y = x + 5
```



Finding Basic Blocks

- Find set of **leaders** **Here: *tuples are instructions***
  - 1) The first tuple of a method is a leader
  - 2) Tuple L is a leader if there is a tuple:

    goto L   if x relop y goto L

  - 3) Tuple M is a leader if it immediately follows a tuple:

    goto L   if x relop y goto L

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

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

```
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)

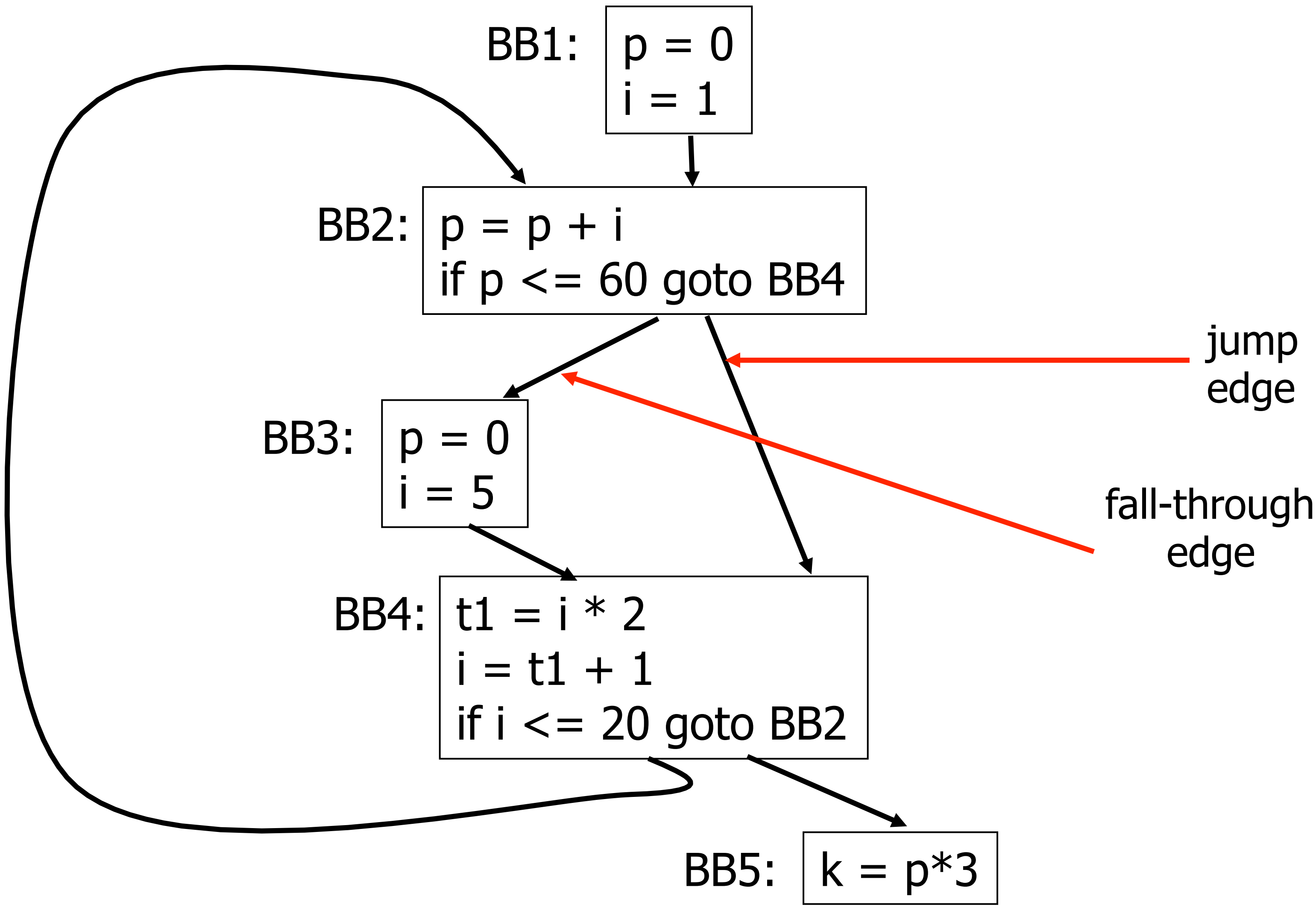
```
1) p = 0
2) i = 1
3) p = p + i
4) if p<=60 goto 7
5) p = 0
6) i = 5
7) t1 = i * 2
8) i = t1 + 1
9) if i<=20 goto 3
10)k = p * 3
```

Leader (rule 1)

Leader (rule 2)

Leader (rule 3)

Basic Blocks and Control Flow Example



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)