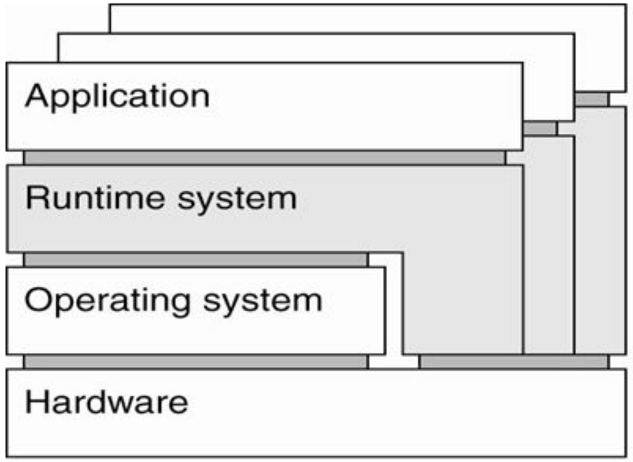
CS 206

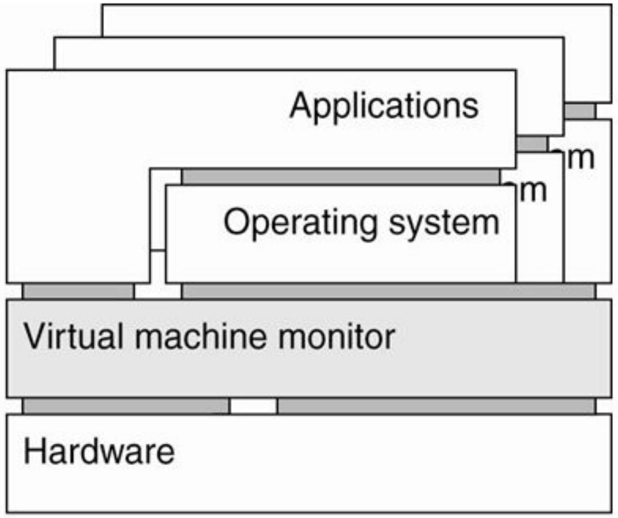
Lecture 22 – Run time Systems, JIT

Virtual Machines

Two Ways to Virtualize



(a)



(b)

<u>Process Virtual Machine</u>: program is compiled to intermediate code, executed by a runtime system

<u>Virtual Machine Monitor:</u> software layer mimics the instruction set; supports an OS and its applications

https://slideplayer.com/slide/4346200/

Java Virtual Machine

JVM Storage Management Storage allocation mechanisms in the JVM mirror those of the Java language:

- Global constant pool
- Set of registers
- Stack for each thread
- Method area to hold executable byte code
- Heap for dynamically allocated objects

x+y*z+u

```
push x
push y
push z
multiply
add
push u
add
```

Stack Machine

Code Conversion

```
class Hello {
    public static void main(String args[]) {
        System.out.println("Hello, world!");
    }
};
```

Code Conversion

- Contents of the JVM's constant pool
- 28 entries in the constant (global) pool

structures elsewhere in the class file; by pointing to these entries, the structures can be self-descriptive

```
const #1 = Method   #6.#15;          //  java/lang/Object."<init>":()V
const #2 = Field    #16.#17;         //  java/lang/System.out:Ljava/io/PrintStream;
const #3 = String   #18;             //  Hello, world!
const #4 = Method   #19.#20;         //  java/io/PrintStream.println:(Ljava/lang/String;)V
const #5 = class    #21;             //  Hello
const #6 = class    #22;             //  java/lang/Object
const #7 = Asciz    <init>;
const #8 = Asciz    ()V;
const #9 = Asciz    Code;
const #10 = Asciz   LineNumberTable;
const #11 = Asciz   main;
const #12 = Asciz   ([Ljava/lang/String;)V;
const #13 = Asciz   SourceFile;
const #14 = Asciz   Hello.java;
const #15 = NameAndType #7:#8;       //  "<init>":()V
const #16 = class   #23;             //  java/lang/System
const #17 = NameAndType #24:#25;     //  out:Ljava/io/PrintStream;
const #18 = Asciz   Hello, world!;
const #19 = class   #26;             //  java/io/PrintStream
const #20 = NameAndType #27:#28;     //  println:(Ljava/lang/String;)V
const #21 = Asciz   Hello;
const #22 = Asciz   java/lang/Object;
const #23 = Asciz   java/lang/System;
const #24 = Asciz   out;
const #25 = Asciz   Ljava/io/PrintStream;;
const #26 = Asciz   java/io/PrintStream;
const #27 = Asciz   println;
const #28 = Asciz   (Ljava/lang/String;)V;
```

Java String

Type signatures

Input string text

files, classes, methods, and fields

JVM Class Files

- Class file is stored as a stream of bytes
- Typically, real file provided by the operating system, could just as easily be a record in a database
- Multiple class files may be combined into a Java archive (.jar) file

- class file has a well-defined hierarchical structure
  - begins with a “magic number” (0x_cafe_babe)
  - Major and minor version numbers of the JVM for which the file was created
  - The constant pool
  - Indices into the constant pool for the current class and its superclass
  - Tables describing the class’s interfaces, fields, and methods

Java Bytecode

- Instruction set of the Java virtual machine
  - Each bytecode is composed of one byte that represents the opcode, along with zero or more bytes for operands
- Stack oriented, operands and results of arithmetic and logic instructions are kept in the operand stack of the current method frame, rather than in registers
  - typical hardware performs arithmetic on values in named registers
  - byte code pops arguments from, and pushes result to, the operand stack of the current method frame
- Instruction set, version 2 categories:

  - load/store
  - arithmetic
  - type conversion
  - object management
  - operand stack management
  - control transfer
  - method calls
  - exceptions & monitors

Java Bytecode Example

C-pseudo	X86 ASM	Java ByteCode (Human-syntax)	Java ByteCode binary

int add
(int a, int b)
{
return a+b;
}
	mov eax, byte [ebp-4]	iload_1	0x1a

	mov edx, byte [ebp-8]	iload_2	0x1b
	add eax, edx	iadd	0x60
	ret	ireturn	0x3e

```
i = j + k;
if (i == 3)
    k = 0;
else
    j = j – 1;
```

```
        ILOAD j     // i = j + k
        ILOAD k
        IADD
        ISTORE i
        ILOAD i     // if (i < 3)
        BIPUSH 3
        IF_ICMPEQ L1
        ILOAD j     // j = j – 1
        BIPUSH 1
        ISUB
        ISTORE j
        GOTO L2
L1:     BIPUSH 0        // k = 0
        ISTORE k
L2:
```

Java Bytecode for List

```
public class LLset {
    node head;
    class node {
        int val;
        node next;
    };
    public LLset() {                // constructor
        head = new node();          // head node contains no real data
        head.next = null;
    }
    ...
}
```

Byte Code for List Insert

```
public void insert(int v) {
    node n = head;

    while (n.next != null
           && n.next.val < v) {

        n = n.next;

    }
    if (n.next == null
        || n.next.val > v) {

        node t = new node();

        t.val = v;

        t.next = n.next;

        n.next = t;

    } // else v already in set
}
```

```
Code:
 Stack=3, Locals=4, Args_size=2
 0:  aload_0                 // this
 1:  getfield       #4;  //Field head:LLLset$node;
 4:  astore_2
 5:  aload_2                 // n
 6:  getfield       #5;  //Field LLset$node.next:LLLset$node;
 9:  ifnull         31   // conditional branch
 12: aload_2
 13: getfield       #5;  //Field LLset$node.next:LLLset$node;
 16: getfield       #6;  //Field LLset$node.val:I
 19: iload_1                 // v
 20: if_icmpge      31
 23: aload_2
 24: getfield       #5;  //Field LLset$node.next:LLLset$node;
 27: astore_2
 28: goto           5
 31: aload_2
 32: getfield       #5;  //Field LLset$node.next:LLLset$node;
 35: ifnull         49
 38: aload_2
 39: getfield       #5;  //Field LLset$node.next:LLLset$node;
 42: getfield       #6;  //Field LLset$node.val:I
 45: iload_1
 46: if_icmple      76
 49: new            #2;  //class LLset$node
 52: dup
 53: aload_0
 54: invokespecial #3;  //Method LLset$node."<init>":(LLLset;)V
 57: astore_3
 58: aload_3                 // t
 59: iload_1
 60: putfield       #6;  //Field LLset$node.val:I
 63: aload_3
 64: aload_2
 65: getfield       #5;  //Field LLset$node.next:LLLset$node;
 68: putfield       #5;  //Field LLset$node.next:LLLset$node;
 71: aload_2
 72: aload_3
 73: putfield       #5;  //Field LLset$node.next:LLLset$node;
 76: return
```

Just-in-Time (JIT) and Dynamic Compilation

- JIT system compiles programs immediately prior to execution, can add significant delay to program start-up time
  - improves the performance of applications by compiling **bytecodes** to **native machine** code at **run time**

- Cost of JIT compilation is typically lessened by the existence of an earlier source-to-byte-code compiler e.g. Java byte code (JBC)

Java JIT

- Java programs consists of classes
  - =>platform-neutral bytecode
- There are two ways of executing bytecode
  - Interpret the byte code at run time in the JVM
  - Compile the byte code
    - Gives a chance for increasing performance
- Overheads?
- Pros?
- Cons?

More Java JIT

- When are methods actually compiled?
  - JVM maintains an invocation count for each method, starts high, decremented every time a method is called
  - Lower number => compilation might happen for often called methods
  - Higher number => these ones will always be interpreted
- Java JIT can compile at multiple optimization levels
  - **cold**, **warm**, **hot**, **veryHot**, or **scorching** (in increasing order)
  - Higher levels provide better performance, also have higher overhead
  - Method can be recompiled to higher optimization levels, depending on usage
- What happens if you disable the JIT?

Binary Translation

- Recompilation of object code
  - sequences of instructions are translated from a source instruction set to the target instruction set

- Allows already-compiled programs to be run on a machine with a different instruction set architecture

e.g. Apple's Rosetta system, which allows programs compiled for older PowerPC-based Macintosh computers to run on newer x86-based Macs

Binary Translation

- Static Translation
  - convert all of the code of an executable file into target code architecture **without** having to run the code first
  - difficult to do correctly, since not all the code can be discovered
  - some parts of the executable may be reachable only through indirect branches, whose value is known only at run-time
- Dynamic Translation
  - Inspect a short sequence of code (basic block)
  - translates BB, cache the resulting sequence
  - Code is translated as it is discovered
- Translation can be done in
  - Hardware
    - Transmeta, Softmachines Inc
    - Intel does it internally (CISC instruction -> micro-ops)
  - Software
    - Run-time engines

```
w = 0;
x = x + y;
if( x > z ){
   y = x;
   x++;
} else {
   y = z;
   z++;
}
w = x + z;
```

Code

```
w = 0;
x = x + y;
if( x > z ){
```

```
   y = x;
   x++;
```

```
   y = z;
   z++;
```

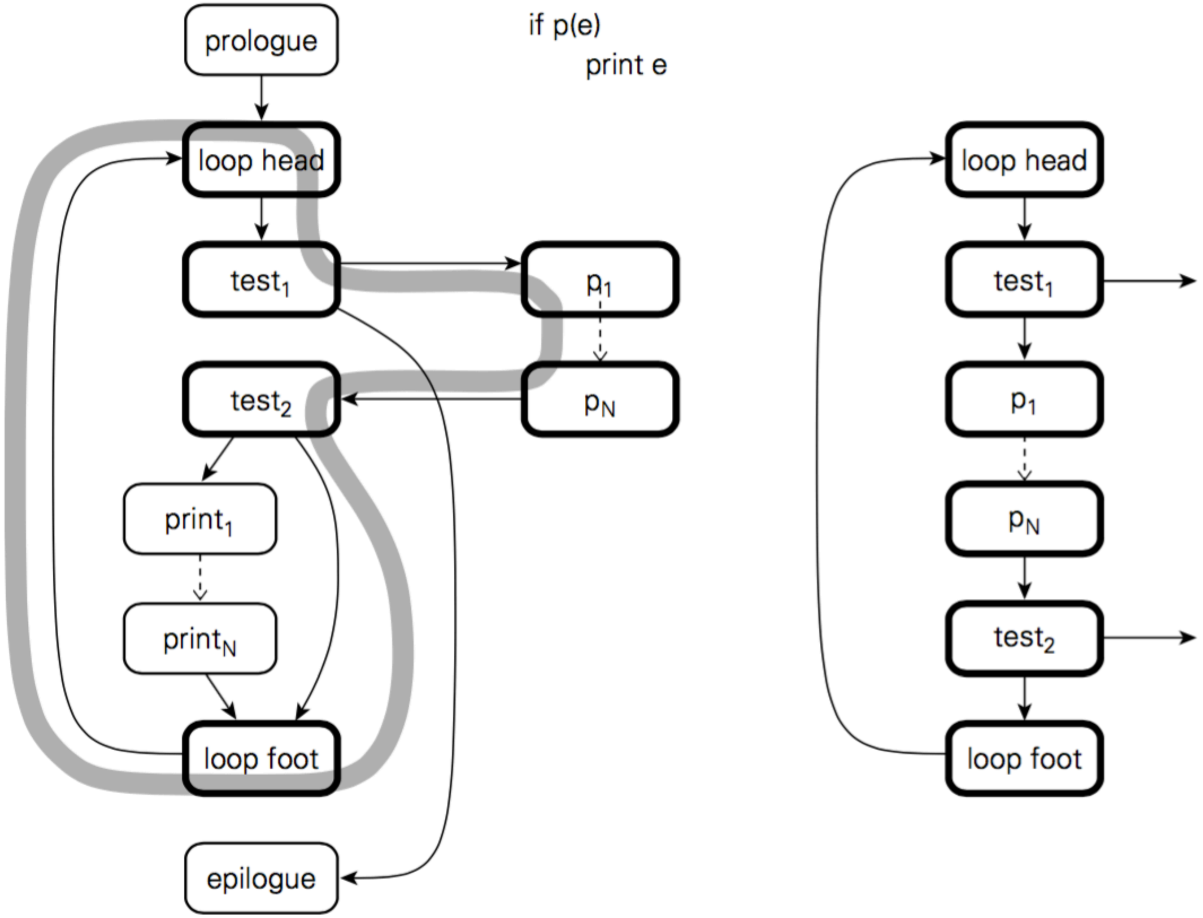
```
w = x + z;
```

Basic Blocks

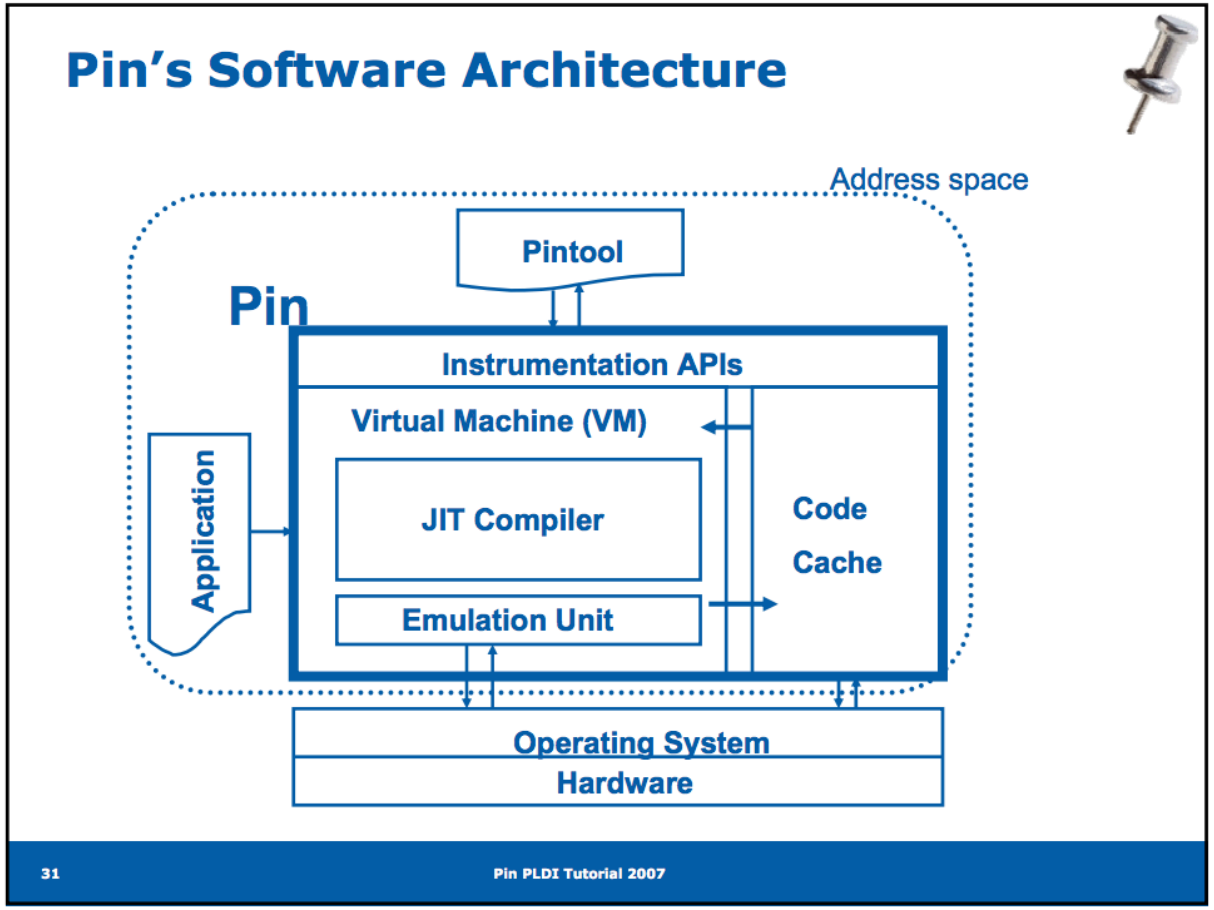
https://www.sra.uni-hannover.de/Theses/2018/BA-FI-approximation-using-basic-blocks.html

Dynamic Optimization

```
procedure print_matching(S : set, p : predicate)
    foreach e in S
        if p(e)
            print e
```



Binary Rewriting / Instrumentation



https://software.intel.com/sites/default/files/article/256675/pldi2007-pintutorial.pdf